DEVELOPING ADOBE® AIR™ APPLICATIONS WITH ADOBE® FLEX™ 3

Contents

Part 1: Installation instructions

Chapter 1: Adobe AIR installation

Adobe® AIR™ allows you to run AIR applications on the desktop. You can install the runtime in the following ways:

- By installing the runtime separately (without also installing an AIR application)
- By installing an AIR application for the first time (you are prompted to also install the runtime)
- By setting up an AIR development environment such as the AIR SDK, Adobe® Flex™ Builder™ 3, or the Adobe Flex™ 3 SDK (which includes the AIR command line development tools)

The runtime only needs to be installed once per computer.

Contents

System requirements for Adobe AIR

The system requirements for running Adobe AIR are:

- For basic Adobe AIR applications:

	Windows	**Macintosh**
Processor	Intel® Pentium® 1.0 GHz or faster processor	PowerPC® G3 1.0 GHz or faster processor or Intel Core™ Duo 1.83 GHz or faster processor
Memory	256 MB RAM	256 MB RAM

OS	Windows 2000 Service Pack 4;
	Windows XP SP2;
	Vista
		Mac OS X 10.4.10 or 10.5.x (PowerPC);
		Mac OS X 10.4.x or 10.5.x (Intel)

- For Adobe AIR applications using full-screen video with hardware scaling:

	Windows	**Macintosh**
Processor	Intel® Pentium® 2.0 GHz or faster processor	PowerPC® G4 1.8GHz GHz or faster processor or Intel Core™ Duo 1.33GHz or faster processor
Memory	512 MB of RAM; 32 MB video RAM	256 MB RAM; 32 MB video RAM

OS	Windows 2000 Service Pack 4;
	Windows XP SP2;
	Vista
		Mac OS X v.10.4.10 or v.10.5 (Intel or PowerPC)
		
		NOTE: The codec used to display H.264 video requires an Intel processor

Installing Adobe AIR

Use the following instructions to download and install the Windows® and Mac OS X versions of AIR.

To update the runtime, a user must have administrative privileges for the computer.

Install the runtime on a Windows computer

1 Download the runtime installation file.

2 Double-click the runtime installation file.

3 In the installation window, follow the prompts to complete the installation.

Install the runtime on a Mac computer

1 Download the runtime installation file.

2 Double-click runtime installation file.

3 In the installation window, follow the prompts to complete the installation.

4 If the Installer displays an Authenticate window, enter your Mac OS user name and password.

Uninstalling Adobe AIR

Once you have installed the runtime, you can uninstall using the following procedures.

Uninstall the runtime on a Windows computer

1 In the Windows Start menu, select Settings > Control Panel.

2 Select the Add or Remove Programs control panel.

3 Select “Adobe AIR” to uninstall the runtime.

4 Click the Change/Remove button.

Uninstall the runtime on a Mac computer

- Double-click the “Adobe AIR Uninstaller”, which is located in the /Applications folder.

Installing and running the AIR sample applications

Some sample applications are available that demonstrate AIR features. You can access and install them using the following instructions:

1 Download and run the AIR sample applications. The compiled applications as well as the source code are available.

2 To download and run a sample application, click the sample application Install Now button. You are prompted to install and run the application.

3 If you choose to download sample applications and run them later, select the download links. You can run AIR applications at any time by:

- On Windows, double-clicking the application icon on the desktop or selecting it from the Windows Start menu.
- On Mac OS, double-clicking the application icon, which is installed in the Applications folder of your user directory (for example, in Macintosh HD/Users/JoeUser/Applications/) by default.

Note: *Check the AIR release notes for updates to these instructions, which are located here: http://www.adobe.com/go/learn_air_relnotes.*

Chapter 2: Setting up the Flex 3 SDK

To develop Adobe® AIR™ applications with Adobe® Flex™, you have the following options:

- You can download and install Adobe® Flex™ Builder™ 3, which provides integrated tools to create Adobe AIR projects and test, debug, and package your AIR applications. See "Developing AIR applications with Flex Builder" on page 20.
- You can download the Adobe® Flex™ 3 SDK and develop Flex AIR applications using your favorite text editor and the command-line tools.

Contents

About the AIR SDK command line tools

Each of the command-line tools you use to create an Adobe AIR application calls the corresponding tool used to build Flex applications:

- `amxmlc` calls `mxmlc` to compile application classes
- `acompc` calls `compc` to compile library and component classes

The only difference between the Flex and the AIR versions of the utilities is that the AIR versions load the configuration options from the `air-config.xml` file instead of the `flex-config.xml` file.

The Flex SDK tools and their command-line options are fully described in Building and Deploying Flex 3 Applications *(http://www.adobe.com/go/learn_flex3_building)* in the Flex 3 documentation library. The Flex SDK tools are described here at a basic level to help you get started and to point out the differences between building Flex applications and building AIR applications.

Install the Flex 3 SDK

Building AIR applications with the command-line tools requires that Java is installed on your computer. You can use the Java virtual machine from either the JRE or the JDK (version 1.4.2 or newer). The Java JRE and JDK are available at http://java.sun.com.

Note: *Java is not required for end users to run AIR applications.*

The Flex 3 SDK provides you with the AIR API and command-line tools that you use to package, compile, and debug your AIR applications.

1 If you haven't already done so, download the Flex 3 SDK.

2 Place the contents of the SDK into a folder (for example, Flex 3 SDK).

3 The command line utilities are located in the `bin` folder.

Compiler setup

You typically specify compilation options both on the command line and with one or more configuration files. The global Flex SDK configuration file contains default values that are used whenever the compilers are run. You can edit this file to suit your own development environment. There are two global Flex configuration files located in the frameworks directory of your Flex 3 SDK installation. The `air-config.xml` file is used when you run the `amxmlc` compiler. This file configures the compiler for AIR by including the AIR libraries. The `flex-config.xml` file is used when you run `mxmlc`.

The default configuration values are suitable for discovering how Flex and AIR work, but when you embark on a full-scale project examine the available options more closely. You can supply project-specific values for the compiler options in a local configuration file that takes precedence over the global values for a given project. For a full list of the compilation options and for the syntax of the configuration files, see *Flex SDK Configuration* in Building and Deploying Flex 3 Applications *(http://www.adobe.com/go/learn_flex3_building)* in the Flex 3 documentation library.

Note: *No compilation options are used specifically for AIR applications, but you must reference the AIR libraries when compiling an AIR application. Typically, these libraries are referenced in a project-level configuration file, in a tool for a build tool such as Ant, or directly on the command line.*

Debugger setup

AIR supports debugging directly, so you do not need a debug version of the runtime (as you would with Adobe® Flash® Player). To conduct command-line debugging, you use the Flash Debugger and the AIR Debug Launcher.

The Flash Debugger is distributed in the Flex 3 SDK directory. The native versions, for example fdb.exe on Windows, are in the bin subdirectory. The Java version is in the lib subdirectory. The AIR Debug Launcher (ADL), adl.exe, is in the bin directory of your Flex SDK installation. (There is no separate Java version).

Note: *You cannot start an AIR application directly with FDB, because FDB attempts to launch it with Flash Player. Instead, let the AIR application connect to a running FDB session.*

Application packager setup

The AIR Developer Tool (ADT), which packages your application into an installable AIR file, is a Java program. No setup is required other than setting up your environment so that you can conveniently run the utility.

The SDK includes a script file in the SDK bin directory for executing ADT as a command. You can also run ADT as a Java program, which could be convenient when using build tools such as Apache Ant.

See also

- “Creating your first AIR application with the Flex SDK” on page 14
- “Creating an AIR application using the command line tools” on page 23

- “Using the AIR Debug Launcher (ADL)” on page 26
- “Packaging an AIR installation file using the AIR Developer Tool (ADT)” on page 28

Part 2: Getting started

Chapter 3: Introducing Adobe AIR

Adobe® AIR™ is a cross-operating system runtime that allows you to leverage your existing web development skills (Adobe® Flash® CS3 Professional, Adobe® Flex™, HTML, JavaScript®, Ajax) to build and deploy Rich Internet Applications (RIAs) to the desktop.

AIR enables you to work in familiar environments, to leverage the tools and approaches you find most comfortable, and by supporting Flash, Flex, HTML, JavaScript, and Ajax, to build the best possible experience that meets your needs.

For example, applications can be developed using one or a combination of the following technologies:

- Flash / Flex / ActionScript
- HTML / JavaScript / CSS / Ajax
- PDF can be leveraged with any application

As a result, AIR applications can be:

- Based on Flash or Flex: Application whose root content is Flash/Flex (SWF)
- Based on Flash or Flex with HTML or PDF. Applications whose root content is Flash/Flex (SWF) with HTML (HTML, JS, CSS) or PDF content included
- HTML-based. Application whose root content is HTML, JS, CSS
- HTML-based with Flash/Flex or PDF. Applications whose root content is HTML with Flash/Flex (SWF) or PDF content included

Users interact with AIR applications in the same way that they interact with native desktop applications. The runtime is installed once on the user's computer, and then AIR applications are installed and run just like any other desktop application.

The runtime provides a consistent cross-operating system platform and framework for deploying applications and therefore eliminates cross-browser testing by ensuring consistent functionality and interactions across desktops. Instead of developing for a specific operating system, you target the runtime, which has the following benefits:

- Applications developed for AIR run across multiple operating systems without any additional work by you. The runtime ensures consistent and predictable presentation and interactions across all the operating systems supported by AIR.
- Applications can be built faster by enabling you to leverage existing web technologies and design patterns and extend your web based applications to the desktop without learning traditional desktop development technologies or the complexity of native code. Easier than using lower level languages such as C and C++, developing applications in AIR does away with the need to learn complex low-level APIs specific to each operating system.

When developing applications for AIR, you can leverage a rich set of frameworks and APIs:

- APIs specific to AIR provided by the runtime and the AIR framework
- ActionScript APIs used in SWF files and Flex framework (as well as other ActionScript based libraries and frameworks)

AIR delivers a new paradigm that dramatically changes how applications can be created, deployed, and experienced. You gain more creative control and can extend your Flash, Flex, HTML, and Ajax-based applications to the desktop, without learning traditional desktop development technologies.

Chapter 4: Finding AIR Resources

For more information on developing AIR applications, see the following resources:

Source	**Location**
Developing Adobe AIR applications with Adobe Flex 3	http://www.adobe.com/go/learn_air_flex3_en
Adobe AIR Quick Starts for Flex	http://www.adobe.com/go/learn_air_flex3_qs_en
Flex 3 Language Reference (including the combined Flex, Flash Player, and AIR APIs)	http://www.adobe.com/go/learn_flex3_aslr_en
Programming ActionScript 3.0	http://www.adobe.com/go/learn_flex3_progas30_en
Flex 3 Developer's Guide	http://www.adobe.com/go/learn_flex3_en
Building and Deploying Flex 3 Applications	http://www.adobe.com/go/learn_flex3_building_en
Creating and Extending Flex 3 Components	http://www.adobe.com/go/learn_flex3_components_en

You can find articles, samples and presentations by both Adobe and community experts on the Adobe AIR Developer Center at http://www.adobe.com/devnet/air/. You can also download Adobe AIR and related software from there.

You can find a section specifically for Flex developers at http://www.adobe.com/devnet/air/flex/.

Visit the Adobe Support website, at http://www.adobe.com/support/, to find troubleshooting information for your product and to learn about free and paid technical support options. Follow the Training link for access to Adobe Press books, a variety of training resources, Adobe software certification programs, and more.

Chapter 5: Creating your first Flex AIR application in Flex Builder

For a quick, hands-on illustration of how Adobe® AIR™ works, use these instructions to create and package a simple SWF file-based AIR "Hello World" application using Adobe® Flex™ Builder™ 3.

If you haven't already done so, download and install Flex Builder 3. For more information, see "Setting up the Flex 3 SDK" on page 5.

Contents

Run Flex Builder and create an AIR project

Flex Builder 3 includes the tools you need to develop and package AIR applications. You begin to create AIR applications in Flex Builder in the same way that you create other Flex-based application projects: by defining a new project.

1 Open Flex Builder 3.

2 Select File > New > Flex Project.

3 Enter the project name as AIRHelloWorld.

4 In Flex, AIR applications are considered an application type. You have two type options: a Flex application that runs on the Web in Adobe® Flash® Player and an AIR application that runs on the desktop in Adobe AIR. Select Desktop Application as the application type.

5 You won't be using a server technology, so select None, and then click Next.

6 Select the folder in which you want to place your compiled application. The default is the bin folder. Click Finish to create the project.

AIR projects initially consist of two files: the main MXML file and an application XML file (referred to as the application descriptor file). The latter file specifies parameters for identifying, installing, and launching AIR applications. There will be occasions when you will want to manually edit this file. For now, be aware that it exists.

Write the AIR application code

To write the "Hello World" application code, you edit the application MXML file (AIRHelloWorld.mxml), which is open in the editor. If it isn't, use the Project Navigator to open the file.

Flex AIR applications are contained within the MXML WindowedApplication tag. The MXML WindowedApplication tag creates a simple window that includes basic window controls such as a title bar and close button.

1 Add a `title` attribute to the WindowedApplication component, and assign it the value `"Hello World"`:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Hello World">

</mx:WindowedApplication>
```

2 Add a Label component to the application (place it inside the WindowedApplication tag), set the `text` property of the Label component to `"Hello AIR"`, and set the layout constraints to keep it centered, as shown here:

```
<?xml version="1.0" encoding="utf-8"?><?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Hello World">
    <mx:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

3 Add the following style block immediately after the opening WindowedApplication tag and before the label component tag you just entered:

```
<mx:Style>
    WindowedApplication
    {
        background-color:"0x999999";
        background-alpha:"0.5";
    }
</mx:Style>
```

These style settings apply to the entire application and render the window background a slightly transparent gray.

The application code now looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Hello World">
    <mx:Style>
        WIndowedApplication
        {
            background-color:"0x999999";
            background-alpha:"0.5";
        }
    </mx:Style>
    <mx:Label text="Hello AIR" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

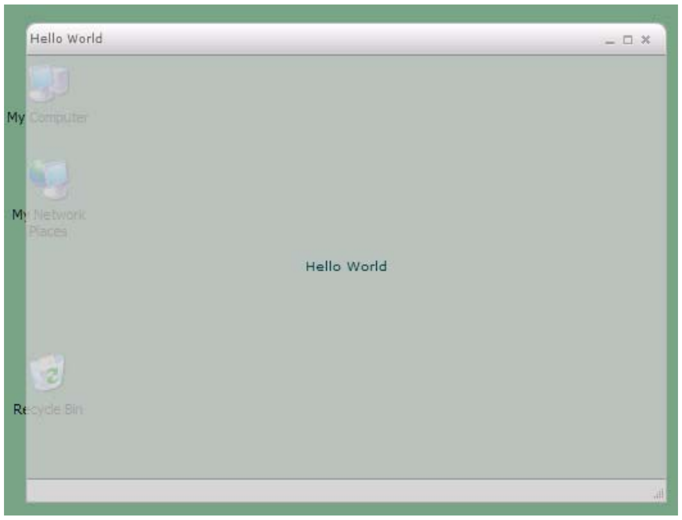
Test the AIR application

To test the application code that you've written, run it in debug mode.

1 Click the Debug button in the main Flex Builder toolbar.

You can also select the Run > Debug > AIRHelloWorld command.

The resulting AIR application should look like the following example (the green background is the desktop):



2 Using the horizontalCenter and verrticalCenter properties of the Label control, the text is placed in the center of the window. Move or resize the window as you would any other desktop application.

***Note:** If the application does not compile, fix any syntax or spelling errors that you inadvertently entered into the code. Errors and warnings are displayed in the Problems view in Flex Builder.*

Package, sign, and run your AIR application

You are now ready to use Flex Builder to package the "Hello World" application into an AIR file for distribution. An AIR file is an archive file that contains the application files, which are all of the files contained in the project's bin folder. In this simple example, those files are the SWF and application XML files. You distribute the AIR package to users who then use it to install the application. A required step in this process is to digitally sign it.

1 Ensure that the application has no compilation errors and runs as expected.

2 Select Project > Export Release Version.

3 If you have multiple projects and applications open in Flex Builder, you must select the specific AIR project you want to package.

4 Select the Export and Sign an AIR File with a Digital Certificate option.

5 If you have an existing digital certificate, click Browse to locate and select it.

6 If you must create a new self-signed digital certificate, select Create.

7 Enter the required information and click OK.

8 Click Finish to generate the AIR package, which is named AIRHelloWorld.air.

You can now run the application from the Project Navigator in Flex Builder or from the file system by double-clicking the AIR file.

See also

- "Developing AIR applications with Flex Builder" on page 20
- "Using the Flex AIR components" on page 38

Chapter 6: Creating your first AIR application with the Flex SDK

For a quick, hands-on illustration of how Adobe® AIR™ works, use these instructions to create a simple SWF-based AIR "Hello World" application using the Flex SDK. You will learn how to compile, test, and package an AIR application with the command-line tools provided with the SDK.

To begin, you must have installed the runtime and set up Adobe® Flex™ 3. You will use the *AMXMLC* compiler, the *AIR Debug Launcher* (ADL), and the *AIR Developer Tool* (ADT) in this tutorial. These programs can be found in the `bin` directory of the Flex 3 SDK (see “Setting up the Flex 3 SDK” on page 5).

Contents

Create the AIR application descriptor file

This section describes how to create the application descriptor, which is an XML file with the following structure:

```
<application>
    <id>...</id>
    <version>...</version>
    <filename>…</filename>
    <initialWindow>
        <content>…</content>
        <visible>…</visible>
        <systemChrome>…</systemChrome>
        <transparent>…</transparent>
        <width>…</width>
        <height>…</height>
    </initialWindow>
</application>
```

1 Create an XML file named `HelloWorld-app.xml` and save it in the project directory.

2 Add the `<application>` element, including the AIR namespace attribute:

   `<application xmlns="http://ns.adobe.com/air/application/1.0">` The last segment of the namespace, “1.0,” specifies the version of the runtime required by the application.

3 Add the `<id>` element:

   `<id>samples.flex.HelloWorld</id>` The application ID uniquely identifies your application along with the publisher ID (which AIR derives from the certificate used to sign the application package). The recommended form is a dot-delimited, reverse-DNS-style string, such as `"com.company.AppName"`. The application ID is used for installation, access to the private application file-system storage directory, access to private encrypted storage, and interapplication communication.

4 Add the `<version>` element:

`<version>0.1</version>` Helps users to determine which version of your application they are installing.

5 Add the `<filename>` element:

`<filename>HelloWorld</filename>` The name used for the application executable, install directory, and similar for references in the operating system.

6 Add the `<initialWindow>` element containing the following child elements to specify the properties for your initial application window:

`<content>HelloWorld.swf</content>` Identifies the root HTML file for AIR to load.

`<visible>true</visible>` Makes the window visible immediately.

`<width>400</width>` Sets the window width (in pixels).

`<height>200</height>` Sets the window height.

7 Save the file. Your complete application descriptor file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/1.0">
    <id>samples.flex.HelloWorld</id>
    <version>0.1</version>
    <filename>HelloWorld</filename>
    <initialWindow>
        <content>HelloWorld.swf</content>
        <visible>true</visible>
        <systemChrome>none</systemChrome>
        <transparent>true</transparent>
        <width>400</width>
        <height>200</height>
    </initialWindow>
</application>
```

This example only sets a few of the possible application properties. For the full set of application properties, which allow you to specify such things as window chrome, window size, transparency, default installation directory, associated file types, and application icons, see “Setting AIR application properties” on page 88

Write the application code

Note: *SWF-based AIR applications can use a main class defined either with MXML or with ActionScript™. This example uses an MXML file to define its main class. The process for creating an AIR application with a main ActionScript class is similar. Instead of compiling an MXML file into the SWF, you compile the ActionScript class file. When using ActionScript, the main class must extend flash.display.Sprite.*

Like all Flex applications, AIR applications built with the Flex framework contain a main MXML file. AIR applications, however, use the *WindowedApplication* component as the root element instead of the Application component. The WindowedApplication component provides properties, methods, and events for controlling your application and its initial window.

The following procedure creates the Hello World application:

1 Using a text editor, create a file named `HelloWorld.mxml` and add the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" title="Hello World">
</mx:WindowedApplication>
```

2 Next, add a Label component to the application (place it inside the WindowedApplication tag).

3 Set the `text` property of the Label component to `"Hello AIR"`.

4 Set the layout constraints to always keep it centered.

The following example shows the code so far:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
    title="Hello World">
        <mx:Label text="Hello World" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

5 Add the following style block:

```
<mx:Style>
    WindowedApplication
    {
        background-color:"0x999999";
        background-alpha:"0.5";
    }
</mx:Style>
```

These styles apply to the entire application and set the window background to be a slightly transparent gray.

The entire application code now looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Hello World">
    <mx:Style>
        WindowedApplication
        {
            background-color:"0x999999";
            background-alpha:"0.5";
        }
    </mx:Style>
    <mx:Label text="Hello World" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

Compile the application

Before you can run and debug the application, you must compile the MXML code into a SWF file using the amxmlc compiler. The amxmlc compiler can be found in the `bin` directory of the Flex 3 SDK. If desired, you can set the path environment of your computer to include the Flex 3 SDK bin directory to make it easier to run the utilities on the command line.

1 Open a command shell or a terminal and navigate to the project folder of your AIR application.

2 Enter the following command:

```
amxmlc HelloWorld.mxml
```

Running `amxmlc` produces `HelloWorld.swf`, which contains the compiled code of the application.

***Note:** If the application does not compile, fix syntax or spelling errors. Errors and warnings are displayed in the console window used to run the amxmlc compiler.*

For more information, see “Compiling an AIR application with the amxmlc compiler” on page 23.

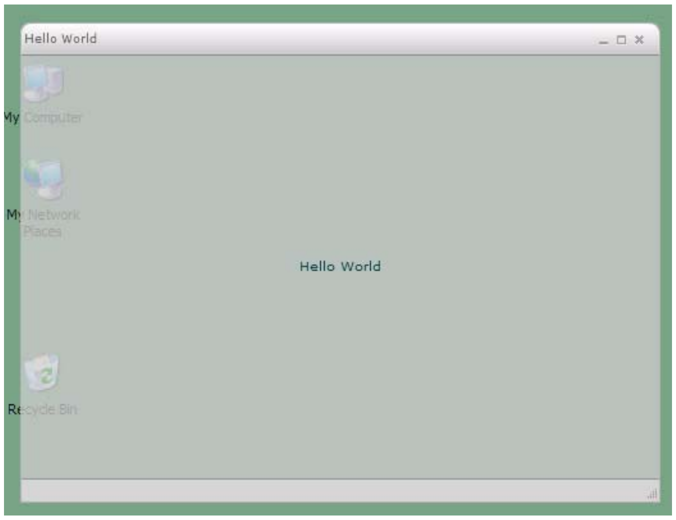
Test the application

To run and test the application from the command line, use the AIR Debug Launcher (ADL) to launch the application using its application descriptor file. (ADL can be found in the bin directory of the Flex 3 SDK.)

❖ From the command prompt, enter the following command:

```
adl HelloWorld-app.xml
```

The resulting AIR application looks something like this (the green background is the user's desktop):



Using the horizontalCenter and verticalCenter properties of the Label control, the text is placed in the center of the window. Move or resize the window as you would any other desktop application.

For more information, see "Using the AIR Debug Launcher (ADL)" on page 26.

Create the AIR installation file

When your application runs successfully, you can use the ADT utility to package the application into an AIR installation file. An AIR installation file is an archive file that contains all the application files, which you can distribute to your users. You must install Adobe AIR before installing a packaged AIR file.

To ensure application security, all AIR installation files must be digitally signed. For development purposes, you can generate a basic, self-signed certificate with ADT or another certificate generation tool. You can also buy a commercial code-signing certificate from a commercial certificate authority such as VeriSign or Thawte. When users install a self-signed AIR file, the publisher is displayed as "unknown" during the installation process. This is because a self-signed certificate only guarantees that the AIR file has not been changed since it was created. There is nothing to prevent someone from self-signing a masquerade AIR file and presenting it as your application. For publicly released AIR files, a verifiable, commercial certificate is strongly recommended. For an overview of AIR security issues, see "AIR security" on page 69.

Generate a self-signed certificate and key pair

❖ From the command prompt, enter the following command (the ADT executable can be found in the `bin` directory of the Flex 3 SDK):

```
adt -certificate -cn SelfSigned 1024-RSA sampleCert.pfx samplePassword
```

This example uses the minimum number of attributes that can be set for a certificate. You can use any values for the parameters in *italics*. The key type must be either *1024-RSA* or *2048-RSA* (see “Digitally signing an AIR file” on page 339).

Create the AIR installation file

❖ From the command prompt, enter the following command (on a single line):

```
adt -package -storetype pkcs12 -keystore sampleCert.pfx HelloWorld.air
HelloWorld-app.xml HelloWorld.swf
```

You will be prompted for the keystore file password.

The `HelloWorld.air` argument is the AIR file that ADT produces. `HelloWorld-app.xml` is the application descriptor file. The subsequent arguments are the files used by your application. This example only uses three files, but you can include any number of files and directories.

After the AIR package is created, you can install and run the application by double-clicking the package file. You can also type the AIR filename as a command in a shell or command window.

See also

- “Setting up the Flex 3 SDK” on page 5.
- “Compiling an AIR application with the amxmlc compiler” on page 23
- “Using the AIR Debug Launcher (ADL)” on page 26
- “Digitally signing an AIR file” on page 339
- “Packaging an AIR installation file using the AIR Developer Tool (ADT)” on page 28

Part 3: AIR development tools

Chapter 7: Developing AIR applications with Flex Builder

Adobe® Flex™ Builder™ 3 provides you with the tools to create Adobe® AIR™ projects, work with the Flex AIR components, and debug and package Adobe AIR applications. The workflow for developing AIR applications in Flex Builder is similar to that for developing most Flex applications.

Contents

Creating AIR projects with Flex Builder

If you have not already done so, install AIR and Flex Builder 3.

1 Open Flex Builder 3.

2 Select File > New > Flex Project.

3 Enter the project name.

4 In Flex, AIR applications are considered an application type. You have two type options: a Flex application that runs on the Web in Adobe® Flash® Player and an AIR application that runs on the desktop in Adobe AIR. Select Desktop Application as the application type.

5 Select the server technology (if any) that you want to use with your AIR application. If you're not using a server technology, select None and then click Next.

6 Select the folder in which you want to place your application. The default is the bin folder. Click Next.

7 Modify the source and library paths as needed and then click Finish to create your AIR project.

Debugging AIR applications with Flex Builder

Flex Builder provides full debugging support for AIR applications. For more information about the debugging capabilities of Flex Builder, refer to Flex Builder Help.

1 Open a source file for the application (such as an MXML file) in Flex Builder.

2 Click the Debug button on the main toolbar .

You can also select Run > Debug.

The application launches and runs in the ADL application (the AIR Debugger Launcher). The Flex Builder debugger catches any breakpoints or runtime errors and you can debug the application like any other Flex application.

You can also debug an application from the command line, using the AIR Debug Launcher command-line tool. For more information, see "Using the AIR Debug Launcher (ADL)" on page 26.

Packaging AIR applications with Flex Builder

When your application is complete and ready to be distributed (or tested running from the desktop), you package it into an AIR file. Packaging consists of the following steps:

- Selecting the AIR application you want to publish
- Optionally allowing users to view the source code and then selecting which of your application files to include
- Digitally signing your AIR application using a Verisign or Thwate digital certificate or by creating and applying a self-signed signature
- Optionally choosing to create an intermediate AIR file, which can be signed at a later time

Package an AIR application

1 Open the project and ensure that the application has no compilation errors and runs as expected.

2 Select Project > Export Release Build.

3 If you have multiple projects and applications open in Flex Builder, select the specific AIR project you want to package.

4 Optionally select Enable View Source if you want users to be able to see the source code when they run the application. You can select individual files to exclude by selecting Choose Source Files. By default all the source files are selected. For more information about publishing source files in Flex Builder, see the Flex Builder Help.

5 You can also optionally change the name of the AIR file that is generated. When you're ready to continue, click Next to digitally sign your application.

Digitally signing your AIR applications

Before continuing with the Export Release Version, decide how you want to digitally sign your AIR application. You have several options. You can sign the application using a Verisign or Thwate digital certificate, you can create and use a self-signed digital certificate, or you can choose to package the application now and sign it later.

Digital certificates by VeriSign and Thwate assure your users of your identity as a publisher and verify that the installation file has not been altered since you signed it. Self-signed digital certificates serve the same purpose but they do not provide validation by a third party.

You also have the option of packaging your AIR application without a digital signature by creating an intermediate AIR file (.airi). An intermediate AIR file is not valid in that it cannot be installed. It is instead used for testing (by the developer) and can be launched using the AIR ADT command line tool. AIR provides this capability because in some development environments a particular developer or team handles signing. This practice insures an additional level of security in managing digital certificates.

For more information about signing applications, see "Digitally signing an AIR file" on page 339.

Digitally sign your AIR application

1 You can digitally sign your AIR application by selecting an existing digital certificate or by creating a new self-signed certificate. Select the Export and Sign an AIR File with a Digital Certificate option.

2 If you have an existing digital certificate, click Browse to locate and select it.

3 To create a new self-signed digital certificate, select Create.

4 Enter the required information and click OK.

5 Click Next to optionally select files to exclude from the exported AIR file. By default, all the files are included.

6 Click Finish to generate the AIR file.

Create an intermediate AIR file

- Select Export an Intermediate AIRI File that will be Exported Later option. Click Finish to generate the intermediate file.

After you have generated an intermediate AIR file, it can be signed using the ADT command line tool (see "Signing an AIR intermediate file with ADT" on page 32).

Create an AIR Library project

To create an AIR code library for multiple AIR projects, create an AIR library project using the standard Flex library project wizard.

1. Select File > New > Flex Library Project.
2. Specify a project name.
3. Select the Add Adobe AIR Libraries and then click Next.

   Note: *The Flex SDK version you select must support AIR. The Flex 2.0.1 SDK does not.*
4. Modify the build path as needed and then click Finish. For more information about creating library projects, see "About library projects" in the Flex Builder Help.

See also

- "The application descriptor file structure" on page 88
- "Using the AIR Debug Launcher (ADL)" on page 26
- "Packaging an AIR installation file using the AIR Developer Tool (ADT)" on page 28

Chapter 8: Creating an AIR application using the command line tools

The Adobe® AIR™ command line tools provide an alternative to Adobe® Flex™ Builder™ for compiling, debugging, and packaging Adobe AIR applications. You can also use these tools in automated build processes. The command line tools are included in both the Flex and AIR SDKs.

Contents

Compiling an AIR application with the amxmlc compiler

You can compile the ActionScript and MXML assets of your AIR application with the command line MXML compiler (amxmlc):

```
amxmlc [compiler options] -- MyAIRApp.mxml
```

where `[compiler options]` specifies the command line options used to compile your AIR application.

The amxmlc command invokes the standard Flex mxmlc compiler with an additional parameter, `+configname=air`. This parameter instructs the compiler to use the air-config.xml file instead of the flex-config.xml file. Using amxmlc is otherwise identical to using mxmlc. The mxmlc compiler and the configuration file format are described in Building and Deploying Flex 3 Applications in the Flex 3 documentation library.

The compiler loads the air-config.xml configuration file specifying the AIR and Flex libraries typically required to compile an AIR application. You can also use a local, project-level configuration file to override or add additional options to the global configuration. Typically, the easiest way to create a local configuration file is to edit a copy of the global version. You can load the local file with the `-load-config` option:

`-load-config=project-config.xml` Overrides global options.

`-load-config+=project-config.xml` Adds additional values to those global options that take more than value, such as the `-library-path` option. Global options that only take a single value are overridden.

If you use a special naming convention for the local configuration file, the amxmlc compiler loads the local file automatically. For example, if the main MXML file is `RunningMan.mxml`, then name the local configuration file: `RunningMan-config.xml`. Now, to compile the application, you only have to type:

```
amxmlc RunningMan.mxml
```

`RunningMan-config.xml` is loaded automatically since its filename matches that of the compiled MXML file.

amxmlc examples

The following examples demonstrate use of the amxmlc compiler. (Only the ActionScript and MXML assets of your application must be compiled.)

Compile an AIR MXML file:

```
amxmlc myApp.mxml
```

Compile and set the output name:

```
amxmlc –output anApp.swf -- myApp.mxml
```

Compile an AIR ActionScript file:

```
amxmlc myApp.as
```

Specify a compiler configuration file:

```
amxmlc –load-config config.xml -- myApp.mxml
```

Add additional options from another configuration file:

```
amxmlc –load-config+=moreConfig.xml -- myApp.mxml
```

Add libraries on the command line (in addition to the libraries already in the configuration file):

```
amxmlc –library-path+=/libs/libOne.swc,/libs/libTwo.swc  -- myApp.mxml
```

Compile an AIR MXML file without using a configuration file (Win):

```
mxmlc -library-path [AIR SDK]/frameworks/libs/air/airframework.swc, ^
[AIR SDK]/frameworks/libs/air/airframework.swc, ^
-library-path [Flex 3 SDK]/frameworks/libs/framework.swc ^
-- myApp.mxml
```

Compile an AIR MXML file without using a configuration file (Mac OS X):

```
mxmlc -library-path [AIR SDK]/frameworks/libs/air/airframework.swc, \
[AIR SDK]/frameworks/libs/air/airframework.swc, \
-library-path [Flex 3 SDK]/frameworks/libs/framework.swc \
-- myApp.mxml
```

Compile an AIR MXML file to use a runtime-shared library:

```
amxmlc -external-library-path+=../lib/myLib.swc -runtime-shared-libraries=myrsl.swf --
myApp.mxml
```

Compiling from Java (with the class path set to include `mxmlc.jar`):

```
java flex2.tools.Compiler +flexlib [Flex SDK 3]/frameworks +configname=air [additional
compiler options] -- myApp.mxml
```

The flexlib option identifies the location of your Flex SDK frameworks directory, enabling the compiler to locate the flex_config.xml file.

Compiling from Java (without the class path set):

```
java -jar [Flex SDK 2]/lib/mxmlc.jar +flexlib [Flex SDK 3]/frameworks +configname=air
[additional compiler options] -- myApp.mxml
```

Compiling an AIR component or library with the acompc compiler

Use the component compiler, acompc, to compile AIR libraries and independent components. The acompc component compiler behaves like the amxmlc compiler, with the following exceptions:

- You must specify which classes within the code base to include in the library or component.
- acompc does not look for a local configuration file automatically. To use a project configuration file, you must use the –load-config option.

The acompc command invokes the standard Flex compc component compiler, but loads its configuration options from the `air-config.xml` file instead of the `flex-config.xml` file.

Contents

Component compiler configuration file

Use a local configuration file to avoid typing (and perhaps incorrectly typing) the source path and class names on the command line. Add the -load-config option to the acompc command line to load the local configuration file.

The following example illustrates a configuration for building a library with two classes, ParticleManager and Particle, both in the package: `com.adobe.samples.particles`. The class files are located in the `source/com/adobe/samples/particles` folder.

```
<flex-config>
    <compiler>
        <source-path>
            <path-element>source</path-element>
        </source-path>
    </compiler>
    <include-classes>
        <class>com.adobe.samples.particles.ParticleManager</class>
        <class>com.adobe.samples.particles.Particle</class>
    </include-classes>
</flex-config>
```

To compile the library using the configuration file, named `ParticleLib-config.xml`, type:

```
acompc -load-config ParticleLib-config.xml -output ParticleLib.swc
```

To run the same command entirely on the command line, type:

```
acompc -source-path source -include-classes com.adobe.samples.particles.Particle 
com.adobe.samples.particles.ParticleManager -output ParticleLib.swc
```

(Type the entire command on one line, or use the line continuation character for your command shell.)

acompc examples

These examples assume that you are using a configuration file named `myLib-config.xml`.

Compile an AIR component or library:

```
acompc -load-config myLib-config.xml -output lib/myLib.swc
```

Compile a runtime-shared library:

```
acompc -load-config myLib-config.xml -directory -output lib
```

(Note, the folder lib must exist and be empty before running the command.)

Reference a runtime-shared library:

```
acompc -load-config myLib-config.xml -output lib/myLib.swc
```

Using the AIR Debug Launcher (ADL)

Use the AIR Debug Launcher (ADL) to run both Flex-based and HTML-based applications during development. Using ADL, you can run an application without first packaging and installing it. By default, ADL uses a runtime included with the SDK, which means you do not have to install the runtime separately to use ADL.

ADL prints trace statements and run-time errors to the standard output, but does not support breakpoints or other debugging features. If you're developing a SWF-based application, use the Flash Debugger (or Flex Builder) for complex debugging issues.

Contents

Launching an application with ADL

Use the following syntax:

```
adl [-runtime runtime-directory] [-pubid publisher-id] [-nodebug] application.xml [root-
directory] [-- arguments]
```

`-runtime runtime-directory` Specifies the directory containing the runtime to use. If not specified, the runtime directory in the same SDK as the ADL program is used. If you move ADL out of its SDK folder, then you must specify the runtime directory. On Windows, specify the directory containing the `Adobe AIR` directory. On Mac OS X, specify the directory containing `Adobe AIR.framework`.

`-pubid publisher-id` Assigns the specified value as the publisher ID of the AIR application for this run. Specifying a temporary publisher ID allows you to test features of an AIR application, such as communicating over a local connection, that use the publisher ID to help uniquely identify an application. The final publisher ID is determined by the digital certificate used to sign the AIR installation file.

`-nodebug` Turns off debugging support. If used, the application process cannot connect to the Flash debugger and dialogs for unhandled exceptions are suppressed. Trace statements still print to the console window. Turning off debugging allows your application to run a little faster and also emulates the execution mode of an installed application more closely.

`application.xml` The application descriptor file. See "Setting AIR application properties" on page 88.

`root-directory` Specifies the root directory of the application to run. If not specified, the directory containing the application descriptor file is used.

`-- arguments` Any character strings appearing after "--" are passed to the application as command line arguments.

***Note:** When you launch an AIR application that is already running, a new instance of that application is not started. Instead, an `invoke` event is dispatched to the running instance.*

Printing trace statements

To print trace statements to the console used to run ADL, add trace statements to your code with the `trace()` function:

```
trace("debug message");
```

ADL Examples

Run an application in the current directory:

```
adl myApp-app.xml
```

Run an application in a subdirectory of the current directory:

```
adl source/myApp-app.xml release
```

Run an application and pass in two command line arguments, "tick" and "tock":

```
adl myApp-app.xml -- tick tock
```

Run an application using a specific runtime:

```
adl -runtime /AIRSDK/runtime myApp-app.xml
```

Connecting to the Flash Debugger (FDB)

To debug a SWF-based AIR application with the Flash Debugger, start an FDB session and then launch a debug version of your application. The debug version of a SWF file automatically connects to a listening FDB session.

1 Start FDB. The FDB program can be found in the `bin` directory of your Flex SDK folder.

  The console displays the FDB prompt: `<fdb>`

2 Execute the Run command: `<fdb>run [Enter]`

3 In a different command or shell console, start a debug version of your application:

  ```
  adl myApp-debug.xml
  ```

4 Using the FDB commands, set breakpoints as desired.

5 Type: `continue [Enter]`

ADL exit and error codes

The following table describes the exit codes printed by ADL:

Exit code	Description
0	Successful launch. ADL exits after the AIR application exits.
1	Successful invocation of an already running AIR application. ADL exits immediately.
2	Usage error. The arguments supplied to ADL are incorrect.
3	The runtime cannot be found.
4	The runtime cannot be started. Often, this occurs because the version or patch level specified in the application does not match the version or patch level of the runtime.
5	An error of unknown cause occurred.
6	The application descriptor file cannot be found.
7	The contents of the application descriptor are not valid. This error usually indicates that the XML is not well formed.
8	The main application content file (specified in the <content> element of the application descriptor file) cannot be found.
9	The main application content file is not a valid SWF or HTML file.

Packaging an AIR installation file using the AIR Developer Tool (ADT)

You create an AIR installation file for both your SWF-based and HTML-based AIR applications using the AIR Developer Tool (ADT). (If you are using Flex Builder to create your application, you can also use the Flex Builder Export wizard to build the AIR file package. See "Developing AIR applications with Flex Builder" on page 20.)

ADT is a Java program that you can run from the command line or a build tool such as Ant. The SDK includes command line scripts that execute the Java program for you. For information on configuring your system to run the ADT tool, see "Setting up the Flex 3 SDK" on page 5.

Contents

Packaging an AIR installation file

Every AIR application must, at a minimum, have an application descriptor file and a main SWF or HTML file. Any other installed application assets must be packaged in the AIR file as well.

All AIR installer files must be signed using a digital certificate. The AIR installer uses the signature to verify that your application file has not been altered since you signed it. You can use a code signing certificate from a certificate authority, such as VeriSign or Thawte, or a self-signed certificate. A certificate issued by a trusted certificate authority provides users of your application some assurance of your identity as publisher. A self-signed certificate cannot be used to verify your identity as the signer (which also weakens the assurance that the package hasn't been altered, because a legitimate installation file could be substituted with a forgery before it reaches the user).

You can package and sign an AIR file in a single step using the ADT `-package` command. You can also create an intermediate, unsigned package with the `-prepare` command, and sign the intermediate package with the `-sign` command in a separate step.

When signing the installation package, ADT automatically contacts a time-stamp authority server to verify the time. The timestamp information is included in the AIR file. An AIR file that includes a verified time-stamp can be installed at any point in the future as long as the certificate was valid at the time of signing. If ADT cannot connect to the time-stamp server, then packaging is canceled. You can override the timestamping option, but without a timestamp, an AIR application ceases to be installable after the certificate used to sign the installation file expires.

If you are creating a package to update an existing AIR application, the package must be signed with the same certificate used to sign the original application or with a certificate that has the same identity. To have the same identity, two certificates must have the same distinguished name (all the informational fields match) and the same certificate chain to the root certificate. Therefore, you can renew a certificate from a certificate authority as long as you do not change any of the identifying information.

***Note:** The settings in the application descriptor file determine the identity of an AIR application and its default installation path. See "The application descriptor file structure" on page 88.*

Package and sign an AIR file in one step

❖ Use the `-package` command with the following syntax (on a single command line):

```
adt -package SIGNING_OPTIONS air_file app_xml [file_or_dir | -C dir file_or_dir | -e file
dir ...] ...
```

`SIGNING_OPTIONS` The signing options identify the keystore containing the private key and certificate used to sign the AIR file. To sign an AIR application with a self-signed certificate generated by ADT, the options to use are:

```
-storetype pkcs12 -keystore certificate.p12
```

In this example, *`certificate.p12`* is the name of the keystore file. The signing options are fully described in "ADT command line signing options" on page 30.

`air_file` The name of the AIR file that is created.

`app_xml` The path to the application descriptor file. The path can be specified relative to the current directory or as an absolute path. (The application descriptor file is renamed as "application.xml" in the AIR file.)

`file_or_dir` The files and directories to package in the AIR file. Any number of files and directories can be specified, delimited by whitespace. If you list a directory, all files and subdirectories within, except hidden files, are added to the package. (In addition, if the application descriptor file is specified, either directly, or through wildcard or directory expansion, it is ignored and not added to the package a second time.) Files and directories specified must be in the current directory or one of its subdirectories. Use the `-C` option to change the current directory.

Important: *Wildcards cannot be used in the `file_or_dir` arguments following the `-C` option. (Command shells expand the wildcards before passing the arguments to ADT, which causes ADT to look for files in the wrong location.) You can, however, still use the dot character, ".", to stand for the current directory in order to package the entire contents of the directory specified in the -C option. In other words, "`-C assets .`" copies everything in the assets directory, including any subdirectories, to the root level of the application package.*

`-C dir` Changes the working directory to the value of *`dir`* before processing subsequent files and directories added to the application package. The files or directories are added to the root of the application package. The `-C` option can be used any number of times to include files from multiple points in the file system. If a relative path is specified for `dir`, the path is always resolved from the original working directory.

As ADT processes the files and directories included in the package, the relative paths between the current directory and the target files are stored. These paths are expanded into the application directory structure when the package is installed. Therefore, specifying `-C release/bin lib/feature.swf` places the file `release/bin/lib/feature.swf` in the `lib` subdirectory of the root application folder.

`-e file dir` Places the specified file into the specified package directory.

Note: *The `<content>` element of the application descriptor file must specify the final location of the main application file within the application package directory tree. This may be an issue if the main application file is not located in the current directory when you run ADT.*

ADT Examples

Package specific application files in the current directory:

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.swf
components.swc
```

Package all files and subdirectories in the current working directory:

```
adt -package -storetype pkcs12 -keystore ../cert.p12 myApp.air myApp.xml .
```

Note: *The keystore file contains the private key used to sign your application. Never include the signing certificate inside the AIR package! If you use wildcards in the ADT command, place the keystore file in a different location so that it is not included in the package. In this example the keystore file, cert.p12, resides in the parent directory.*

Package only the main files and an images subdirectory:

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.swf images
```

Package the application.xml file and main SWF located in a working directory (release\bin):

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air release\bin\myApp.xml -C
release\bin myApp.swf
```

Package assets from more than one place in your build file system. In this example, the application assets are located in the following folders before packaging:

```
/devRoot
    /myApp
        /release
            /bin
                myApp.xml
                myApp.swf
    /artwork
        /myApp
            /images
                image-1.png
                ...
                image-n.png
    /libraries
        /release
            /libs
                lib-1.swf
                ...
                lib-n.swf
```

Running the following ADT command from the /devRoot/myApp directory:

```
adt –package -storetype pkcs12 -keystore cert.p12 myApp.air release/bin/myApp.xml
    –C release/bin myApp.swf
    –C ../artwork/myApp images
    –C ../libraries/release libs
```

Results in the following package structure:

```
/myAppRoot
    /META-INF
        /AIR
            application.xml
            hash
    myApp.swf
    mimetype
    /images
        image-1.png
        ...
        image-n.png
    /libs
        lib-1.swf
        ...
        lib-n.swf
```

Run ADT as a Java program (without setting the classpath):

```
java –jar {AIRSDK}\lib\ADT.jar –package -storetype pkcs12 -keystore cert.p12 myApp.air
myApp.xml myApp.swf
```

Run ADT as a Java program (with the Java classpath set to include the ADT.jar package):

```
java com.adobe.air.ADT –package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml
myApp.swf
```

ADT command line signing options

ADT uses the Java Cryptography Architecture (JCA) to access private keys and certificates for signing AIR applications. The signing options identify the keystore and the private key and certificate within that keystore.

The keystore must include both the private key and the associated certificate chain. The certificate chain is used to establish the publisher ID for the application. If the signing certificate chains to a trusted certificate on a computer, then the common name of the certificate is displayed as the publisher name on the AIR installation dialog.

ADT requires that the certificate conform to the x509v3 standard (RFC3280) and include the Extended Key Usage extension with the proper values for code signing. Constraints within the certificate are respected and could preclude the use of some certificates for signing AIR applications.

***Note:** ADT uses the Java runtime environment proxy settings, when appropriate, for connecting to Internet resources for checking certificate revocation lists and obtaining time-stamps. If you encounter problems connecting to Internet resources when using ADT and your network requires specific proxy settings, you may need to configure the JRE.*

Specifying AIR signing options

❖ To specify the ADT signing options for the `-package` and `-prepare` commands, use the following syntax:

```
[-alias aliasName] [-storetype type] [-keystore path] [-storepass password1] [-keypass
password2] [-providerName className] [-tsa url]
```

`-alias aliasName` The alias of a key in the keystore. Specifying an alias is not necessary when a keystore only contains a single certificate. If no alias is specified, ADT uses the first key in the keystore.

Not all keystore management applications allow an alias to be assigned to certificates. When using the Windows system keystore for example, use the distinguished name of the certificate as the alias. You can use the Java Keytool utility to list the available certificates so that you can determine the alias. For example, running the command:

```
keytool -list -storetype Windows-MY
```

produces output like the following for a certificate:

```
CN=TestingCert,OU=QE,O=Adobe,C=US, PrivateKeyEntry,
Certificate fingerprint (MD5): 73:D5:21:E9:8A:28:0A:AB:FD:1D:11:EA:BB:A7:55:88
```

To reference this certificate on the ADT command line, set the alias to:

```
CN=TestingCert,OU=QE,O=Adobe,C=US
```

On Mac OS X, the alias of a certificate in the Keychain is the name displayed in the Keychain Access application.

`-storetype type` The type of keystore, determined by the keystore implementation. The default keystore implementation included with most installations of Java supports the `JKS` and `PKCS12` types. Java 5.0 includes support for the `PKCS11` type, for accessing keystores on hardware tokens, and `Keychain` type, for accessing the Mac OS X keychain. Java 6.0 includes support for the `MSCAPI` type (on Windows). If other JCA providers have been installed and configured, additional keystore types might be available. If no keystore type is specified, the default type for the default JCA provider is used.

Store type	Keystore format	Minimum Java version
JKS	Java keystore file (.keystore)	1.2
PKCS12	PKCS12 file (.p12 or .pfx)	1.4
PKCS11	Hardware token	1.5
KeychainStore	Mac OS X Keychain	1.5
Windows-MY or Windows-ROOT	MSCAPI	1.6

`-keystore path` The path to the keystore file for file-based store types.

`-storepass password1` The password required to access the keystore. If not specified, ADT prompts for the password.

`-keypass password2` The password required to access the private key that is used to sign the AIR application. If not specified, ADT prompts for the password.

`-providerName className` The JCA provider for the specified keystore type. If not specified, then ADT uses the default provider for that type of keystore.

`-tsa url` Specifies the URL of an RFC3161-compliant timestamp server to time-stamp the digital signature. If no URL is specified, a default time-stamp server provided by Geotrust is used. When the signature of an AIR application is time-stamped, the application can still be installed after the signing certificate expires, because the timestamp verifies that the certificate was valid at the time of signing.

If ADT cannot connect to the time-stamp server, then signing is canceled and no package is produced. Specify `-tsa none` to disable time-stamping. However, an AIR application packaged without a timestamp ceases to be installable after the signing certificate expires.

***Note:** The signing options are like the equivalent options of the Java Keytool utility. You can use the Keytool utility to examine and manage keystores on Windows. The Apple® security utility can also be used for this purpose on Mac OS X.*

Signing option examples

Signing with a .p12 file:

```
-storetype pkcs12 -keystore cert.p12
```

Signing with the default Java keystore:

```
-alias AIRcert -storetype jks
```

Signing with a specific Java keystore:

```
-alias AIRcert -storetype jks -keystore certStore.keystore
```

Signing with the Mac OS X keychain:

```
-alias AIRcert -storetype KeychainStore -providerName Apple
```

Signing with the Windows system keystore:

```
-alias cn=AIRCert -storeype Windows-MY
```

Signing with a hardware token (refer to the token manufacturer’s instructions on configuring Java to use the token and for the correct `providerName` value):

```
-alias AIRCert -storetype pkcs11 -providerName tokenProviderName
```

Signing without embedding a timestamp:

```
-storetype pkcs12 -keystore cert.p12 -tsa none
```

Creating an unsigned AIR intermediate file with ADT

Use the `-prepare` command to create an unsigned AIR intermediate file. An AIR intermediate file must be signed with the ADT `-sign` command to produce a valid AIR installation file.

The `-prepare` command takes the same flags and parameters as the `-package` command (except for the signing options). The only difference is that the output file is not signed. The intermediate file is generated with the filename extension: `airi`.

To sign an AIR intermediate file, use the ADT `-sign` command. (See Signing an AIR intermediate file with ADT.)

ADT example

```
adt -prepare unsignedMyApp.airi myApp.xml myApp.swf components.swc
```

Signing an AIR intermediate file with ADT

To sign an AIR intermediate file with ADT, use the `-sign` command. The sign command only works with AIR intermediate files (extension `airi`). An AIR file cannot be signed a second time.

To create an AIR intermediate file, use the adt `-prepare` command. (See "Creating an unsigned AIR intermediate file with ADT" on page 32.)

Sign an AIRI file

❖ Use the ADT `-sign` command with following syntax:

```
adt -sign SIGNING_OPTIONS airi_file air_file
```

`SIGNING_OPTIONS` The signing options identify the private key and certificate with which to sign the AIR file. These options are described in "ADT command line signing options" on page 30.

`airi_file` The path to the unsigned AIR intermediate file to be signed.

`air_file` The name of the AIR file to be created.

ADT Example

```
adt -sign -storetype pkcs12 -keystore cert.p12 unsignedMyApp.airi myApp.air
```

For more information, see "Digitally signing an AIR file" on page 339.

Creating a self-signed certificate with ADT

Self-signed certificates allow you to produce a valid AIR installation file, but only provide limited security assurances to your users since the authenticity of self-signed certificates cannot be verified. When a self-signed AIR file is installed, the publisher information is displayed to the user as Unknown. A certificate generated by ADT is valid for five years.

If you create an update for an AIR application that was signed with a self-generated certificate, you must use the same certificate to sign both the original and update AIR files. The certificates that ADT produces are always unique, even if the same parameters are used. Thus, if you want to self-sign updates with an ADT-generated certificate, preserve the original certificate in a safe location. In addition, you will be unable to produce an updated AIR file after the original ADT-generated certificate expires. (You can publish new applications with a different certificate, but not new versions of the same application.)

Important: *Because of the limitations of self-signed certificates, Adobe strongly recommends using a commercial certificate from a reputable certificate authority, such as VeriSign or Thawte, for signing publicly released AIR applications.*

The certificate and associated private key generated by ADT are stored in a PKCS12-type keystore file. The password specified is set on the key itself, not the keystore.

Generating a digital ID certificate for self-signing AIR files

❖ Use the ADT `-certificate` command (on a single command line):

```
adt -certificate -cn name [-ou org_unit][-o org_name][-c country] key_type pfx_file password
```

`-cn name` The string assigned as the common name of the new certificate.

`-ou org_unit` A string assigned as the organizational unit issuing the certificate. (Optional.)

`-o org_name` A string assigned as the organization issuing the certificate. (Optional.)

`-c country` A two-letter ISO-3166 country code. A certificate is not generated if an invalid code is supplied. (Optional.)

`key_type` The type of key to use for the certificate, either "1024-RSA" or "2048-RSA".

`pfx_file` The path for the certificate file to be generated.

`password` The password for the new certificate. The password is required when signing AIR files with this certificate.

Certificate generation examples

```
adt -certificate -cn SelfSign -ou QE -o "Example, Co" -c US 2048-RSA newcert.p12 39#wnetx3tl
adt -certificate -cn ADigitalID 1024-RSA SigningCert.p12 39#wnetx3tl
```

To use these certificates to sign AIR files, you use the following signing options with the ADT `-package` or `-prepare` commands:

```
-storetype pkcs12 -keystore newcert.p12 -keypass 39#wnetx3tl
-storetype pkcs12 -keystore SigningCert.p12 -keypass 39#wnetx3tl
```

Using Apache Ant with the SDK tools

This topic provides examples of using the Apache Ant build tool to compile, test, and package AIR applications.

***Note:** This discussion does not attempt to provide a comprehensive outline of Apache Ant. For Ant documentation, see http://Ant.Apache.org.*

Contents

Using Ant for simple projects

This example illustrates building an AIR application using Ant and the AIR command line tools. A simple project structure is used with all files stored in a single directory.

***Note:** This example assumes that you are using the command line tools in the Flex 3 SDK rather than Flex Builder. The tools and configuration files in the SDKs included with Flex Builder are stored in a different directory structure.*

To make it easier to reuse the build script, these examples use several defined properties. One set of properties identifies the installed locations of the command line tools:

```
    <property name="SDK_HOME" value="C:/Flex3SDK"/>
    <property name="MXMLC.JAR" value="${SDK_HOME}/lib/mxmlc.jar"/>
    <property name="ADL" value="${SDK_HOME}/bin/adl.exe"/>
    <property name="ADT.JAR" value="${SDK_HOME}/lib/adt.jar"/>
```

The second set of properties is project specific. These properties assume a naming convention in which the application descriptor and AIR files are named based on the root source file. Other conventions are easily supported. The properties also define the MXMLC debug parameter as true (by default).

```
    <property name="APP_NAME" value="ExampleApplication"/>
    <property name="APP_ROOT" value="."/>
    <property name="MAIN_SOURCE_FILE" value="${APP_ROOT}/${APP_NAME}.mxml"/>
    <property name="APP_DESCRIPTOR" value="${APP_ROOT}/${APP_NAME}-app.xml"/>
    <property name="AIR_NAME" value="${APP_NAME}.air"/>
    <property name="DEBUG" value="true"/>
    <property name="STORETYPE" value="pkcs12"/>
    <property name="KEYSTORE" value="ExampleCert.p12"/>
```

Invoking the compiler

To invoke the compiler, the example uses a Java task to run mxmlc.jar:

```
<target name="compile">
    <java jar="${MXMLC.JAR}" fork="true" failonerror="true">
        <arg value="-debug=${DEBUG}"/>
        <arg value="+flexlib=${SDK_HOME}/frameworks"/>
        <arg value="+configname=air"/>
        <arg value="-file-specs=${MAIN_SOURCE_FILE}"/>
    </java>
</target>
```

When invoking mxmlc using Java, you must specify the `+flexlib` parameter. The `+configname=air` parameter instructs mxmlc to load the supplied AIR configuration file along with the normal Flex config file.

Invoking ADL to test an application

To run the application with ADL, use an exec task:

```
<target name="test" depends="compile">
    <exec executable="${ADL}">
        <arg value="${APP_DESCRIPTOR}"/>
    </exec>
</target>
```

Invoking ADT to package an application

To package the application use a Java task to run the adt.jar tool:

```
<target name="package" depends="compile">
    <java jar="${ADT.JAR}" fork="true" failonerror="true">
        <arg value="-package"/>
        <arg value="-storetype"/>
        <arg value="${STORETYPE}"/>
        <arg value="-keystore"/>
        <arg value="${KEYSTORE}"/>
        <arg value="${AIR_NAME}"/>
        <arg value="${APP_DESCRIPTOR}"/>
        <arg value="${APP_NAME}.swf"/>
        <arg value="*.png"/>
    </java>
</target>
```

If your application has more files to package, you can add additional `<arg>` elements.

Using Ant for more complex projects

The directory structure of a typical application is more complex than a single directory. The following example illustrates a build file used to compile, test, and package an AIR application which has a more practical project directory structure.

This sample project stores the application source files and other assets like icon files within a src directory. The build script creates the following working directories:

build Stores the release (non-debug) versions of compiled SWF files.

debug Stores an unpackaged debug version of the application, including any compiled SWFs and asset files. The ADL utility runs the application from this directory.

release Stores the final AIR package

The AIR tools require the use of some additional options when operating on files outside the current working directory:

Compiling The -output option of the mxmlc compiler allows you to specify where to place the compiled file; in this case, in the build or debug subdirectories. To specify the output file, the line:

```
<arg value="-output=${debug}/${APP_ROOT_FILE}"/>
```

is added to the compilation task.

Testing The second argument passed to ADL specifies the root directory of the AIR application. To specify the application root directory, the following line is added to the testing task:

```
<arg value="${debug}"/>
```

Packaging Packaging files from subdirectories that should not be part of the final package structure requires using the -C directive to change the ADT working directory. When you use the -C directive, files and directories in the new working directory are copied to the root level of the AIR package file. Thus, `-C build file.png` copies `file.png` to the root of the application directory. Likewise, `-C assets icons` copies the icon folder to the root level, and copies all the files and directories within the icons folder as well. For example, the following sequence of lines in the package task adds the icons directory directly to the root level of the application package file:

```
<arg value="-C"/>
<arg value="${assets}"/>
<arg value="icons"/>
```

***Note:** If you need to move many resources and assets into different relative locations, it is typically easier to marshall them into a temporary directory using Ant tasks than it is to build a complex argument list for ADT. Once your resources are organized, a simple ADT argument list can be used to package them.*

```
<?xml version="1.0" ?>
<project>
    <!-- SDK properties -->
    <property name="SDK_HOME" value="C:/Flex3SDK"/>
    <property name="MXMLC.JAR" value="${SDK_HOME}/lib/mxmlc.jar"/>
    <property name="ADL" value="${SDK_HOME}/bin/adl.exe"/>
    <property name="ADT.JAR" value="${SDK_HOME}/lib/adt.jar"/>

    <!-- Project properties -->
    <property name="APP_NAME" value="ExampleApplication"/>
    <property name="APP_ROOT_DIR" value="."/>
    <property name="MAIN_SOURCE_FILE" value="${APP_ROOT_DIR}/src/${APP_NAME}.mxml"/>
    <property name="APP_ROOT_FILE" value="${APP_NAME}.swf"/>
    <property name="APP_DESCRIPTOR" value="${APP_ROOT_DIR}/${APP_NAME}-app.xml"/>
    <property name="AIR_NAME" value="${APP_NAME}.air"/>
    <property name="build" location="${APP_ROOT}/build"/>
    <property name="debug"  location="${APP_ROOT_DIR}/debug"/>
    <property name="release"  location="${APP_ROOT_DIR}/release"/>
    <property name="assets"  location="${APP_ROOT_DIR}/src/assets"/>
    <property name="STORETYPE" value="pkcs12"/>
    <property name="KEYSTORE" value="ExampleCert.p12"/>

    <target name="init" depends="clean">
        <mkdir dir="${build}"/>
        <mkdir dir="${debug}"/>
        <mkdir dir="${release}"/>
    </target>

    <target name="debugcompile" depends="init">
        <java jar="${MXMLC.JAR}" fork="true" failonerror="true">
            <arg value="-debug=true"/>
            <arg value="+flexlib=${SDK_HOME}/frameworks"/>
            <arg value="+configname=air"/>
            <arg value="-file-specs=${MAIN_SOURCE}"/>
            <arg value="-output=${debug}/${APP_ROOT_FILE}"/>
        </java>
        <copy todir="${debug}">
            <fileset dir="${assets}"/>
        </copy>
    </target>
    <target name="releasecompile" depends="init">
        <java jar="${MXMLC.JAR}" fork="true" failonerror="true">
            <arg value="-debug=false"/>
            <arg value="+flexlib=${SDK_HOME}/frameworks"/>
            <arg value="+configname=air"/>
            <arg value="-file-specs=${MAIN_SOURCE_FILE}"/>
            <arg value="-output=${build}/${APP_ROOT_FILE}"/>
        </java>
    </target>

    <target name="test" depends="debugcompile">
        <exec executable="${ADL}">
            <arg value="${APP_DESCRIPTOR}"/>
            <arg value="${debug}"/>
        </exec>
    </target>

    <target name="package" depends="releasecompile">
        <java jar="${ADT.JAR}" fork="true" failonerror="true">
            <arg value="-package"/>
            <arg value="-storetype"/>
            <arg value="${STORETYPE}"/>
            <arg value="-keystore"/>
            <arg value="${KEYSTORE}"/>
            <arg value="${release}/${AIR_NAME}"/>
            <arg value="${APP_DESCRIPTOR}"/>
            <arg value="-C"/>
            <arg value="${build}"/>
            <arg value="${APP_ROOT_FILE}"/>
            <arg value="-C"/>
            <arg value="${assets}"/>
            <arg value="icons"/>
        </java>
    </target>

    <target name="clean" description="clean up">
        <delete dir="${build}"/>
        <delete dir="${debug}"/>
        <delete dir="${release}"/>
    </target>
</project>
```

Chapter 9: Using the Flex AIR components

When building an AIR application in Flex, you can use any of the controls and other components that are part of Flex. In addition, Flex includes a set of components that are specifically for AIR applications. The Flex AIR components can be divided into the following groups:

File system controls

The file system controls are a set of user-interface controls that provide information about and tools to interact with the file system of the local computer on which the AIR application is running. These include controls for displaying lists of files in tree or grid format, controls for choosing directories or files from a list or combo box, and so on.

- "About file system controls" on page 39
- "FileSystemComboBox control" on page 39
- "FileSystemTree control" on page 41
- "FileSystemList control" on page 41
- "FileSystemDataGrid control" on page 42
- "FileSystemHistoryButton control" on page 43
- "Example: Displaying a directory structure with Flex AIR" on page 45

HTML control

The HTML control is used to display an HTML web page within a Flex application. For example, you could use it to combine HTML and Flex content in a single application.

- "About the HTML control" on page 45

FlexNativeMenu control

The FlexNativeMenu control provides the ability to use MXML to declaratively define the structure of a native menu. You can use it to define an application menu (on OS X), a window's menu (on Windows), a context menu, and so forth.

- "About the FlexNativeMenu control" on page 46

Window containers

The window containers are two components that can be used as containers for defining the layout of windows in applications. There are two window containers: the ApplicationWindow, a substitute for the Application container to use as the main or initial window of an AIR application; and the Window, for application windows that are opened after the initial window of the application.

- "About window containers" on page 63.
- "WindowedApplication container" on page 64
- "Window container" on page 65

For more information about these components, see the Flex 3 Language Reference.

About file system controls

The Flex file system components combine the functionality of other Flex controls, such as Tree, DataGrid, ComboBox, and so forth, with pre-built awareness of the file system on the application user's computer. These controls duplicate the functionality of user interface controls that are commonly used in desktop applications for browsing and selecting files and directories. You can use one or two of them to directly include file-related functionality in a screen of your application. Or you can combine several of them together to create a full-featured file browsing or selection dialog box.

Each of the Flex file system controls, except the FileSystemHistoryButton control, displays a view of the contents of a particular directory in the computer's file system. For instance, the FileSystemTree displays the directory's contents in a hierarchical tree (using the Flex Tree control) and the FileSystemComboBox displays the directory and its parent directories in the menu of a ComboBox control.

For any of the Flex file system controls except the FileSystemHistoryButton control, you use the control's `directory` property to change the currently selected directory for a control. You can also use the `directory` property to retrieve the current directory, such as if the user selects a directory in the control.

Contents

FileSystemComboBox control

A FileSystemComboBox defines a combo box control for selecting a location in a file system. The control always displays the selected directory in the combo box's text field. When the combo box's drop-down list is displayed, it shows the hierarchy of directories that contain the selected directory, up to the computer root directory. The user can select a higher-level directory from the list. In this sense, the FileSystemComboBox control's behavior is different from the FileSystemTree, FileSystemList, and FileSystemDataGrid controls that display the directories and files that are contained by the current directory.

For more information on the FileSystemComboBox control, see the Flex 3 Language Reference.

Contents

Creating a FileSystemComboBox control

You use the `<mx:FileSystemComboBox>` tag to define a FileSystemComboBox control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. The `directory` property can be set in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property, the data provider of the underlying combo box is automatically populated. By default the `directory` property is set to the root "Computer" directory, which has no ancestor directories and hence shows no selectable directories in the combo box's drop-down list.

The following example shows four variations on the basic FileSystemComboBox. Each combo box is initially set to the user's desktop directory, in the application's `creationComplete` handler. The distinct characteristics of the combo boxes are as follows:

- The first combo box simply displays the selected directory.
- The second combo box's `showIcons` property is set to `false`, so no icon appears next to the items in the combo box's list.
- The third combo box's `indent` property is set to 20, which is larger than the default. As a result, the items in the combo box's list are more indented than normal.
- The fourth combo box has an event handler defined for the `directoryChange` event. When the selected directory in the combo box changes, it calls the `setOutput()` method, which writes the selected directory's path to a TextArea control named `output`.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
creationComplete="init();">
    <mx:Script>
        <![CDATA[
            import flash.filesystem.File;

            private function init():void
            {
                fcb.directory = File.desktopDirectory;
                fcbIndent.directory = File.desktopDirectory;
                fcbNoIcons.directory = File.desktopDirectory;
                fcbChange.directory = File.desktopDirectory;
            }

            private function setOutput():void
            {
                output.text = fcbChange.directory.nativePath;
            }
        ]]>
    </mx:Script>
    <mx:FileSystemComboBox id="fcb"/>
    <mx:FileSystemComboBox id="fcbNoIcons" showIcons="false"/>
    <mx:FileSystemComboBox id="fcbIndent" indent="20"/>
    <mx:FileSystemComboBox id="fcbChange" directoryChange="setOutput();"/>
    <mx:TextArea id="output" width="200" height="50"/>
</mx:WindowedApplication>
```

FileSystemComboBox user interaction

The FileSystemComboBox supports the same user interaction as a standard combo box control. The control displays a directory in its selection field. The user clicks the button (or uses the keyboard) to open a drop-down list containing the names of the hierarchy of directories that contain the selected directory. The user can then select one of the directories, which causes the drop-down list to close and the selected directory to become the current directory. When the user selects a directory, the control dispatches the `directoryChange` event, and its `directory` property changes to the newly selected directory.

FileSystemTree control

A FileSystemTree control displays the contents of a file system directory as a tree. The tree can display the directory's files, its subdirectories, or both. For files, file names can be displayed with or without extensions.

For more information on the FileSystemTree control, see the Flex 3 Language Reference.

Contents

Creating a FileSystemTree control

You use the `<mx:FileSystemComboBox>` tag to define a FileSystemComboBox control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. You can set the directory property in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property, the data provider of the underlying tree control is automatically populated. The specified directory isn't displayed in the tree—its child files or directories are shown as the top-level nodes of the tree. By default the `directory` property is set to the root "Computer" directory. Consequently, its children (the drive or drives attached to the computer) are displayed as the top branches of the tree.

The following example demonstrates creating a FileSystemTree control that displays all files and folders, which is the default. In addition, hidden files are shown by setting the `showHidden` property to `true`.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:FileSystemTree showHidden="true"/>
</mx:WindowedApplication>
```

FileSystemTree user interaction

The FileSystemTree control supports the same types of user interaction as the standard Flex Tree control. If the user double-clicks a closed directory node or clicks its disclosure icon, the control dispatches a `directoryOpening` event. If the user double-clicks an open directory node or clicks its disclosure icon, the control dispatches a `directoryClosing` event. If the user double-clicks a file node, the control dispatches a `select` event.

FileSystemList control

A FileSystemList control displays the contents of a file system directory as selectable items in a scrolling list (a Flex List control). The displayed contents can include subdirectories and files, with additional filtering options as well. A FileSystemList control can be linked to a FileSystemHistoryButton control, meaning that the button can be used to move to a previously displayed directory.

For more information on the FileSystemList control, see the Flex 3 Language Reference.

Contents

Creating a FileSystemList control

You use the `<mx:FileSystemList>` tag to define a FileSystemList control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. The directory property can be set in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property, the data provider of the underlying list control is automatically populated. The specified directory isn't displayed in the list—its child files or directories are shown as the items in the list. By default the `directory` property is set to the root "Computer" directory. In that case its children, which are the drive or drives attached to the computer, are displayed as the items in the list.

The following example demonstrates creating a FileSystemList control that displays all files and folders (the default). The sample also includes a button for navigating up one level in the directory hierarchy. The button is enabled if the currently displayed directory has a parent directory because the button's `enabled` property is bound to the FileSystemList control's `canNavigateUp` property. The button navigates up one level by calling the FileSystemList control's `navigateUp()` method when it is clicked.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Button label="Up" click="fileList.navigateUp();"
        enabled="{fileList.canNavigateUp}"/>
    <mx:FileSystemList id="fileList"/>
</mx:WindowedApplication>
```

FileSystemList user interaction

The FilesSystemList control provides standard scrolling list functionality for files: a user can scroll through the list of files and select one or multiple files or directories. When the user double-clicks a directory, the FileSystemList control automatically sets that directory as the control's `directory` property. It then becomes the directory whose contents are displayed in the list.

FileSystemDataGrid control

A FileSystemDataGrid displays file information in a data-grid format. The file information displayed includes the file name, creation date, modification date, type, and size. Data grid columns displaying this information are automatically created in the underlying DataGrid control, and can be removed or customized in the same way that you customize DataGrid columns. The displayed contents can include subdirectories and files, with additional filtering options as well. A FileSystemList control can be linked to a FileSystemHistoryButton control, meaning that the button can be used to move to a previously displayed directory.

For more information on the FileSystemDataGrid control, see the Flex 3 Language Reference.

Contents

Creating a FileSystemDataGrid control

You use the `<mx:FileSystemDataGrid>` tag to define a FileSystemDataGrid control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the currently displayed directory using the control's `directory` property. You can set the directory property in MXML by binding the value to a property or variable, or by setting the property in ActionScript. When you set the `directory` property the data provider of the underlying data grid control is automatically populated. The specified directory isn't displayed in the grid—its child files or directories are shown as the rows in the grid. By default the `directory` property is set to the root "Computer" directory. In that case its children, the drive or drives attached to the computer, are displayed as the items in the grid.

The following example demonstrates creating a FileSystemDataGrid control that displays all files and folders (the default). The sample also includes a button for navigating up one level in the directory hierarchy. The button is enabled if the currently displayed directory has a parent directory because the button's `enabled` property is bound to the FileSystemDataGrid control's `canNavigateUp` property. The button navigates up one level by calling the FileSystemDataGrid control's `navigateUp()` method when it is clicked.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Button label="Up" click="fileGrid.navigateUp();"
        enabled="{fileGrid.canNavigateUp}"/>
    <mx:FileSystemDataGrid id="fileGrid"/>
</mx:WindowedApplication>
```

FileSystemDataGrid user interaction

The FileSystemDataGrid control includes standard DataGrid functionality such as scrolling through the grid, selecting grid rows, reordering grid columns, and sorting grid data by clicking the grid headers. In addition, the FileSystemDataGrid provides some file-specific functionality. A FileSystemDataGrid allows a user to navigate to other directories using the mouse or keyboard. The user can change the directory by double-clicking a subdirectory, by pressing Enter or Ctrl-Down when a subdirectory is selected, by pressing Ctrl-Up when the control isn't displaying the COMPUTER directory, by pressing Ctrl-Left when there is a "previous" directory to navigate back to, or by pressing Ctrl-Right when there is a "next" directory to navigate forward to. If the user attempts to change the directory being displayed, the control dispatches a cancelable `directoryChanging` event. If the event isn't canceled, the control displays the contents of the new directory and its `directory` property changes. Whenever the `directory` property changes for any reason, the controls dispatches a `directoryChange` event.

FileSystemHistoryButton control

The FileSystemHistoryButton control lets the user move backwards or forwards through the navigation history of another control. It works in conjunction with a FileSystemList or FileSystemDataGrid control, or any similar control with a property containing an array of File objects. The FileSystemHistoryButton is a PopUpMenuButton. It has a button for navigating back or forward one step in the history. It also has a list of history steps from which one step can be chosen.

To link a FileSystemHistoryButton to a control, bind the button's `dataProvider` property to one of the control's properties. The property must contain an array of File objects representing a sequence of directories in a file system browsing history. For instance, you can bind the `dataProvider` property to the `forwardHistory` or `backHistory` property of a FileSystemList or FileSystemDataGrid control. The button can then be used to navigate the display history of that control if you set the `click` and `itemClick` event handlers of the button to call the `navigateForward()` or `navigateBack()` method of the control.

For more information on the FileSystemHistoryButton control, see the Flex 3 Language Reference.

Contents

Creating a FileSystemHistoryButton control

You use the `<mx:FileSystemHistoryButton>` tag to define a FileSystemHistoryButton control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the property to which the button is bound by setting the `dataProvider` property.

The following example demonstrates creating two FileSystemHistoryButton controls that are linked to the display history of a FileSystemList control. Each button's `enabled` property is bound to the FileSystemList control's `canNavigateBack` or `canNavigateForward` property. As a result, the button is enabled if the currently displayed directory can navigate in the appropriate direction. When the user clicks a button, its event listener calls the FileSystemList control's `navigateBack()` or `navigateForward()` method. This causes the FileSystemList control to navigate to the previous or next directory.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:HBox>
        <mx:FileSystemHistoryButton label="Back"
            dataProvider="{fileList.backHistory}"
            enabled="{fileList.canNavigateBack}"
            click="fileList.navigateBack();"
            itemClick="fileList.navigateBack(event.index)"/>
        <mx:FileSystemHistoryButton label="Forward"
            dataProvider="{fileList.forwardHistory}"
            enabled="{fileList.canNavigateForward}"
            click="fileList.navigateForward();"
            itemClick="fileList.navigateForward(event.index)"/>
    </mx:HBox>
    <mx:FileSystemList id="fileList"/>
</mx:WindowedApplication>
```

FileSystemHistoryButton user interaction

The FileSystemHistoryButton is based on the Flex PopUpMenuButton, so their core functionality is the same. When the user clicks the main button the click event is dispatched (normally moving backward or forward one step in the history). In addition, by clicking the pull-down menu button, a list of the history steps is displayed. This allows the user to navigate directly to a specific step in the history.

Example: Displaying a directory structure with Flex AIR

The following example uses the WindowedApplication container and the FileSystemTree and FileSystemDataGrid controls. In this example, the FileSystemTree control displays a directory structure. Clicking a directory name in the FileSystemTree control causes the FileSystemDataGrid control to display information about the files in the selected directory:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:HDividedBox>
    <mx:FileSystemTree id="tree"
        width="200" height="100%"
        directory="{new File('C:\\')}"
        enumerationMode="directoriesOnly"
        change="dataGrid.directory = File(tree.selectedItem);"/>
    <mx:FileSystemDataGrid id="dataGrid"
        width="100%" height="100%"
        directory="{new File('C:\\')}"/>
    </mx:HDividedBox>
</mx:WindowedApplication>
```

About the HTML control

An HTML control displays HTML web pages in your application. It is designed to be used to render specific external HTML content within your AIR application. It offers functionality like a lightweight web browser, including loading HTML pages, navigation history, and the ability to access the raw HTML content. The HTML control is not designed or intended to be used as a replacement for the Text or TextArea controls. Those controls are more appropriate for displaying formatted text or for use as an item renderer for displaying short runs of text.

Contents

Creating an HTML control

You use the `<mx:HTML>` tag to define an HTML control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the location of the HTML page to display by setting the `location` property.

The following example demonstrates the use of an HTML control in a simple application. The HTML control's `location` property is set to "http://labs.adobe.com/", so that URL is opened in the control when it loads. In addition, when the "back" and "forward" are clicked they call the control's `historyBack()` and `historyForward()` methods. A TextInput control allows the user to enter a URL location. When a third "go" button is clicked, the HTML control's `location` property is set to the `text` property of the input text field.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:ControlBar width="100%">
        <mx:Button label="&lt; Back" click="content.historyBack();"/>
        <mx:Button label="Forward &gt;" click="content.historyForward();"/>
        <mx:TextInput id="address" text="{content.location}" width="100%"/>
        <mx:Button label="Go!" click="content.location = address.text"/>
    </mx:ControlBar>
    <mx:Canvas width="100%" height="100%">
        <mx:HTML id="content" location="http://labs.adobe.com/"/>
    </mx:Canvas>
</mx:WindowedApplication>
```

HTML control user interaction

For a user interacting with an HTML control, the experience is like using a web browser with only the content window and no menu bar or navigation buttons. The HTML page content displays in the control. The user can interact with the content through form fields and buttons and by clicking hyperlinks. Some of these interactions, such as clicking a link or submitting a form, would normally cause a browser to load a new page. These actions cause the HTML control to display the content of the new page and also change the value of the control's `location` property.

About the FlexNativeMenu control

A FlexNativeMenu component is a Flex wrapper for the NativeMenu class. The FlexNativeMenu allows you to use MXML and a data provider to define the structure of a menu. The FlexNativeMenu component does not have any visual representation that is rendered by Flex. Instead, a FlexNativeMenu instance defines a native operating system menu such as an application menu (OS X), a window menu (Windows), a context menu, or any other native menu that can be created in AIR. For a complete list of the ways a native menu can be used in AIR, see "AIR menu basics" on page 127.

The FlexNativeMenu component is designed to be like the Flex Menu and MenuBar components. Developers who have worked with those components should find the FlexNativeMenu familiar.

For more information on the FlexNativeMenu control, see the Flex 3 Language Reference.

Contents

Creating a FlexNativeMenu control

You define a FlexNativeMenu control in MXML by using the `<mx:FlexNativeMenu>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML application, either in another tag or in an ActionScript block.

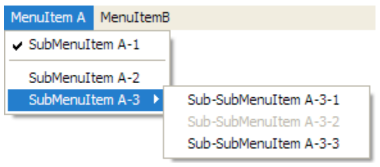
You specify the data for the FlexNativeMenu control by using the `dataProvider` property. The FlexNativeMenu control uses the same types of data providers as does the ManuBar control and the Menu control. Several of the XML attributes or object property names have meaning to the FlexNativeMenu control. For more information on structuring FlexNativeMenu data providers, see "Defining FlexNativeMenu menu structure and data" on page 51.

You can assign any name to node tags in the XML data. In subsequent examples, each node is named with the generic `<menuitem>` tag, but you can use `<node>`, `<subNode>`, `<person>`, `<address>`, and so on.

Contents

Creating an application or window menu

When you create an application or window menu using the FlexNativeMenu control, the top-level objects or nodes in the data provider correspond to the top-level menu items. In other words, they define the items that display in the menu bar itself. Items nested inside one of those top-level items define the items within the menu. Likewise, those menu items can contain items, in which case the menu item is a submenu. When the user selects the menu item it expands its own menu items. For example, the following screenshot displays a window menu with three menu items (plus an additional separator menu item). The item with the label "SubMenuItem A-3" in turn contains three menu items, so SubMenuItem A-3 is treated as a submenu. (The code to create this menu is provided later.)



For an MXML application using the Flex WindowedApplication container as the root MXML node, you can assign a FlexNativeMenu to the WindowedApplication instance's `menu` property. The menu is used as the application menu on OS X and the window menu of the initial window on Windows. Likewise, to specify a window menu for an additional window defined using the Flex Window container, assign a FlexNativeMenu to the Window instance's `menu` property. In that case the menu displays on Windows only and is ignored on OS X.

Note: *Mac OS X defines a menu containing standard items for every application. Assigning a FlexNativeMenu object to the* `menu` *property of the WindowedApplication component replaces the standard menu rather than adding additional menus to it.*

The following application defines a FlexNativeMenu as the `menu` property of a WindowedApplication container. Consequently, the specified menu is used as the application menu on OS X and the window menu of the initial window on Windows. This code creates the menu shown in the previous screenshot:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
backgroundColor="#ffffff" layout="absolute">
    <mx:menu>
        <mx:FlexNativeMenu dataProvider="{myMenuData}"
            labelField="@label"
            showRoot="false"/>
    </mx:menu>
    <mx:XML format="e4x" id="myMenuData">
        <root>
            <menuitem label="MenuItem A">
                <menuitem label="SubMenuItem A-1" type="check" toggled="true"/>
                <menuitem type="separator"/>
                <menuitem label="SubMenuItem A-2"/>
                <menuitem label="SubMenuItem A-3">
                    <menuitem label="Sub-SubMenuItem A-3-1"/>
                    <menuitem label="Sub-SubMenuItem A-3-2" enabled="false"/>
                    <menuitem label="Sub-SubMenuItem A-3-3"/>
                </menuitem>
            </menuitem>
            <menuitem label="MenuItemB">
```

```
                <menuitem label="SubMenuItem B-1"/>
                <menuitem label="SubMenuItem B-2"/>
            </menuitem>
        </root>
    </mx:XML>
</mx:WindowedApplication>
```

Creating a context menu

Creating a context menu in a Flex AIR application involves two steps. You create the FlexNativeMenu instance that defines the menu structure. You then assign that menu as the context menu for its associated control. Because a context menu consists of a single menu, the top-level menu items serve as the items in the single menu. Any menu item that contains child menu items defines a submenu within the single context menu.

The FlexNativeMenu is a replacement for the context menu that you use with browser-based Flex applications (the flash.ui.ContextMenu class). You can use one type of menu or the other, but you can't specify both types for a single component.

To assign a FlexNativeMenu component as the context menu for a visual Flex control, call the FlexNativeMenu instance's `setContextMenu()` method, passing the visual control as the `component` parameter (the only parameter):

```
menu.setContextMenu(someComponent);
```

The same FlexNativeMenu can be used as the context menu for more than one object, by calling `setContextMenu()` multiple times using different `component` parameter values. You can also reverse the process (that is, remove an assigned context menu) using the `unsetContextMenu()` method.

The following example demonstrates creating a FlexNativeMenu component and setting it as the context menu for a Label control:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="init();">

    <mx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void
            {
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;
                myMenu.setContextMenu(lbl);
            }
        ]]>
    </mx:Script>

    <!-- The XML data provider -->
    <mx:XML format="e4x" id="menuData">
        <root>
            <menuitem label="MenuItem A"/>
            <menuitem label="MenuItem B"/>
            <menuitem label="MenuItem C"/>
        </root>
    </mx:XML>

    <mx:Label id="lbl" x="100" y="10"
```

```
        text="Right-click here to open menu"/>
</mx:WindowedApplication>
```

In addition to context menus for visual components within an application window, an AIR application supports two other special context menus: dock icon menus (OS X) and system tray icon menus (Windows). To set either of these menus, you define the menu's structure using the FlexNativeMenu component, then you assign the FlexNativeMenu instance to the WindowedApplication container's `dockIconMenu` or `systemTrayIconMenu` property.

Before setting the `dockIconMenu` or `systemTrayIconMenu` property you may want to determine whether the user's operating system supports a dock icon or a system tray icon, using the NativeApplication class's static `supportsDockIcon` and `supportsSystemTrayIcon` properties. Doing so isn't necessary, but can be useful. For instance, you might want to customize a menu depending on whether it is used as the context menu for a dock icon or for a system tray icon.

Finally, while a dock icon exists automatically for an application, you must explicitly specify a system tray icon in order for the icon to appear. (Naturally, the icon must exist in order for the user to be able to right-click the icon to activate the context menu).

The following example defines a FlexNativeMenu that is used as a context menu. If the user's operating system supports a system tray icon, the code creates an icon and displays it in the system tray. The code then assigns the FlexNativeMenu instance as the context menu for the system tray and dock icon menus.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    creationComplete="init();">

    <mx:Script>
        <![CDATA[
            import flash.desktop.DockIcon;
            import flash.desktop.InteractiveIcon;
            import flash.desktop.NativeApplication;
            import flash.desktop.SystemTrayIcon;
            import flash.display.Shape;
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void
            {
                // Create the menu
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;

                var icon:InteractiveIcon;
                icon = NativeApplication.nativeApplication.icon;

                // If we need a system tray icon, create one and display it
                if (NativeApplication.supportsSystemTrayIcon)
                {
                    var iconData:BitmapData = createSystemTrayIcon();
                    SystemTrayIcon(icon).bitmaps = new Array(iconData);
                }

                // Use this approach if you want to assign the same menu
                // to the dock icon and system tray icon
                this.systemTrayIconMenu = this.dockIconMenu = myMenu;
                // Use this approach if you want to assign separate menus
//              if (NativeApplication.supportsDockIcon)
//              {
//                  this.dockIconMenu = myMenu;
//              }
//              else if (NativeApplication.supportsSystemTrayIcon)
//              {
//                  this.systemTrayIconMenu = myMenu;
//              }
            }

            private function createSystemTrayIcon():BitmapData
            {
                // Draw the icon in a Graphic
                var canvas:Shape = new Shape();
                canvas.graphics.beginFill(0xffff00);
                canvas.graphics.drawCircle(24, 24, 24);
                canvas.graphics.endFill();
                canvas.graphics.beginFill(0x000000);
                canvas.graphics.drawEllipse(13, 13, 9, 12);
                canvas.graphics.drawEllipse(27, 13, 9, 12);
                canvas.graphics.endFill();
                canvas.graphics.lineStyle(3, 0x000000);
                canvas.graphics.moveTo(11, 32);
                canvas.graphics.curveTo(24, 46, 37, 32);

                var result:BitmapData = new BitmapData(48, 48, true, 0x00000000);
                result.draw(canvas);

                return result;
            }
        ]]>
    </mx:Script>

    <!-- The XML data provider -->
    <mx:XML format="e4x" id="menuData">
        <root>
            <menuitem label="MenuItem A"/>
            <menuitem label="MenuItem B"/>
            <menuitem label="MenuItem C"/>
        </root>
    </mx:XML>

    <mx:Text text="Right-click on the dock icon (Mac OS X) or system tray icon (Windows)"/>
</mx:WindowedApplication>
```

***Note:** Mac OS X defines a standard menu for the application dock icon. When you assign a FlexNativeMenu as the dock icon's menu, the items in that menu are displayed above the standard items. You cannot remove, access, or modify the standard menu items.*

Creating a pop-up menu

A pop-up menu is like a context menu, but the pop-up menu isn't necessarily associated with a particular Flex component. To open a pop-up menu, create a FlexNativeMenu instance and set its `dataProvider` property to populate the menu. To open the menu on the screen, call its `display()` method:

```
myMenu.display(this.stage, 10, 10);
```

The `display()` method has three required parameters: the Stage instance that defines the coordinates within which the menu is placed, the x coordinate where the menu is placed, and the y coordinate for the menu. For an example of using the `display()` method to create a pop-up menu, see "Example: An Array FlexNativeMenu data provider" on page 54.

One important thing to keep in mind is that the `display()` method operates immediately when it's called. Several property changes cause the FlexNativeMenu's data provider to invalidate (such as changes to the data provider, changing the `labelField`, and so forth). When the `display()` method is called immediately after making such changes, those changes aren't reflected in the menu that appears on the screen. For example, in the following code listing when the button is clicked no menu appears because the menu is created and the data provider is specified in the same block of code in which the `display()` method is called:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">

    <mx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private function createAndShow():void
            {
                var myMenu:FlexNativeMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;
                // calling display() here has no result, because the data provider
                // has been set but the underlying NativeMenu hasn't been created yet.
                myMenu.display(this.stage, 10, 10);
            }
        ]]>
    </mx:Script>

    <!-- The XML data provider -->
    <mx:XML format="e4x" id="menuData">
        <root>
            <menuitem label="MenuItem A"/>
            <menuitem label="MenuItem B"/>
            <menuitem label="MenuItem C"/>
        </root>
    </mx:XML>

    <!-- Button control to create and open the menu. -->
    <mx:Button x="300" y="10"
        label="Open Menu"
        click="createAndShow();"/>
</mx:WindowedApplication>
```

Defining FlexNativeMenu menu structure and data

The techniques for defining structure and data for a FlexNativeMenu are like the techniques for structuring all Flex menu controls. Consequently, this section does not provide comprehensive information on structuring Flex menus, but instead focuses on differences between FlexNativeMenu structure versus other Flex menu components. For more information on structuring data providers for all Flex menu controls, see "Defining menu structure and data" on page 348. For more information on hierarchical data providers including data descriptors, see "Hierarchical data objects" on page 167.

The `dataProvider` property of a FlexNativeMenu defines the structure of the menu. To change a menu's structure at runtime, change the data provider and the menu updates itself accordingly. Menus typically use a hierarchical data provider such as nested arrays or XML. However, a simple menu may consist of a single flat structure of menu items.

A FlexNativeMenu instance uses a data descriptor to parse and manipulate the data provider's contents. By default, a FlexNativeMenu control uses a DefaultDataDescriptor instance as its descriptor. However, you can customize menu data parsing by creating your own data descriptor class and setting it as the FlexNativeMenu control's `dataDescriptor` property. The DefaultDataDescriptor supports a data provider that is an XML object or XMLList object, an array of objects, an object with a `children` property containing an array of objects, or a collection that implements the ICollectionView interface such as an ArrayCollection or XMLListCollection instance.

Contents

Specifying and using menu entry information

Information in a FlexNativeMenu control's data provider determines how each menu entry appears and is used. To access or change the menu contents, you modify the contents of the data provider.

The FlexNativeMenu class uses the methods of the IMenuDataDescriptor interface to access and manipulate information in the data provider that defines the menu behavior and contents. Flex includes the DefaultDataDescriptor class that implements this interface. A FlexNativeMenu control uses the DefaultDataDescriptor class if you do not specify another class in the `dataDescriptor` property.

Menu entry types

Each data provider entry can specify an item type and type-specific information about the menu item. Menu-based classes support the following item types (`type` field values):

Menu item type	**Description**
normal	The default type. Selecting an item with the `normal` type triggers an `itemClick` event. Alternatively, if the item has children, the menu dispatches a `menuShow` event and opens a submenu.
check	Selecting an item with the `check` type toggles the menu item's `toggled` property between `true` and `false` values and triggers an `itemClick` event. When the menu item is in the `true` state, it displays a check mark in the menu next to the item's label.
separator	Items with the `separator` type provide a simple horizontal line that divides the items in a menu into different visual groups.

Unlike other Flex menu controls, the FlexNativeMenu component does not support radio-button menu items (type `radio`).

Menu attributes

Menu items can specify several attributes that determine how the item is displayed and behaves. The following table lists the attributes you can specify, their data types, their purposes, and how the data provider must represent them if the menu uses the DefaultDataDescriptor class to parse the data provider:

Attribute	Type	Description
`altKey`	Boolean	Specifies whether the Alt key is required as part of the key equivalent for the item.
`cmdKey`	Boolean	Specifies whether the Command key is required as part of the key equivalent for the item.
`ctrlKey`	Boolean	Specifies whether the Control key is required as part of the key equivalent for the item.
`enabled`	Boolean	Specifies whether the user can select the menu item (`true`), or not (`false`). If not specified, Flex treats the item as if the value were `true`. If you use the default data descriptor, data providers must use an `enabled` XML attribute or object field to specify this characteristic.
`keyEquivalent`	String	Specifies a keyboard character which, when pressed, triggers an event as though the menu item was selected. The menu's `keyEquivalentField` or `keyEquivalentFunction` property determines the name of the field in the data that specifies the key equivalent, or a function for determining the key equivalents. (If the data provider is in E4X XML format, you must specify one of these properties to assign a key equivalent.)
`label`	String	Specifies the text that appears in the control. This item is used for all menu item types except `separator` . The menu's `labelField` or `labelFunction` property determines the name of the field in the data that specifies the label, or a function for determining the labels. (If the data provider is in E4X XML format, you must specify one of these properties to display a label.) If the data provider is an array of strings, Flex uses the string value as the label.
`mnemonicIndex`	Integer	Specifies the index position of the character in the label that is used as the mnemonic for the menu item. The menu's `mnemonicIndexField` or `mnemonicIndexFunction` property determines the name of the field in the data that specifies the mnemonic index, or a function for determining mnemonic index. (If the data provider is in E4X XML format, you must specify one of these properties to specify a mnemonic index in the data.) Alternatively, you can indicate that a character in the label is the menu item's mnemonic by including an underscore immediately to the left of that character.
`shiftKey`	String	Specifies whether the Shift key is required as part of the key equivalent for the item.
`toggled`	Boolean	Specifies whether a `check` item is selected. If not specified, Flex treats the item as if the value were `false` and the item is not selected. If you use the default data descriptor, data providers must use a `toggled` XML attribute or object field to specify this characteristic.
`type`	String	Specifies the type of menu item. Meaningful values are `separator` and `check`. Flex treats all other values, or nodes with no type entry, as normal menu entries. If you use the default data descriptor, data providers must use a `type` XML attribute or object field to specify this characteristic.

Unlike other Flex menu controls, the FlexNativeMenu component does not support the `groupName` or `icon` attributes. In addition, it supports the additional attribute `keyEquivalent` and the key equivalent modifier attributes `altKey`, `cmdKey`, `ctrlKey`, and `shiftKey`.

The FlexNativeMenu component ignores all other object fields or XML attributes, so you can use them for application-specific data.

Considerations for XML-based FlexNativeMenu data providers

In a simple case for creating a single menu or menu bar using the FlexNativeMenu control, you might use an `<mx:XML>` or `<mx:XMLList>` tag and standard XML node syntax to define the menu data provider. When you use an XML-based data provider, keep the following rules in mind:

- With the `<mx:XML>` tag you must have a single root node, and you set the `showRoot` property of the FlexNativeMenu control to `false`. (Otherwise, your FlexNativeMenu would have only the root node as a menu item.) With the `<mx:XMLList>` tag you define a list of XML nodes, and the top-level nodes define the top-level menu items.
- If your data provider has `label`, `keyEquivalent`, or `mnemonicIndex` attributes, the default attribute names are not recognized by the DefaultDataDescriptor class. Set the FlexNativeMenu control's `labelField` , `keyEquivalentField`, or `mnemonicIndexField` property and use the E4X @ notation to specify the attribute name, such as:

```
labelField="@label"
keyEquivalentField="@keyEquivalent"
mnemonicIndexField="@mnemonicIndex"
```

Example: An Array FlexNativeMenu data provider

The following example uses a FlexNativeMenu component to display a popup menu. It demonstrates how to define the menu structure using an Array of plain objects as a data provider. For an application that specifies an identical menu structure in XML, see “Example: An XML FlexNativeMenu data provider” on page 55.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="init();">

    <mx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void
            {
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.showRoot = false;
            }

            // Method to show the menu.
            private function show():void
            {
                myMenu.display(this.stage, 10, 10);
            }

            // The Array data provider
            [Bindable]
            public var menuData:Array = [
                {label: "MenuItem A"},
                {label: "MenuItem B", type: "check", toggled: true},
                {label: "MenuItem C", enabled: false},
                {type: "separator"},
                {label: "MenuItem D", children: [
                    {label: "SubMenuItem D-1"},
                    {label: "SubMenuItem D-2"},
                    {label: "SubMenuItem D-3"}
                    ]}
                ];
        ]]>
    </mx:Script>

    <!-- Button control to create and open the menu. -->
    <mx:Button x="300" y="10"
        label="Open Menu"
        click="show();"/>
</mx:WindowedApplication>
```

Example: An XML FlexNativeMenu data provider

The following example displays a popup menu using a FlexNativeMenu component. It shows how to define the menu structure using XML as a data provider. For an application that specifies an identical menu structure using an Array of objects as a data provider, see "Example: An Array FlexNativeMenu data provider" on page 54.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="init();">

    <mx:Script>
        <![CDATA[
            import mx.controls.FlexNativeMenu;

            private var myMenu:FlexNativeMenu;

            private function init():void
            {
                myMenu = new FlexNativeMenu();
                myMenu.dataProvider = menuData;
                myMenu.labelField = "@label";
                myMenu.showRoot = false;
            }

            // Method to show the menu.
            private function show():void
            {
                myMenu.display(this.stage, 10, 10);
            }
        ]]>
    </mx:Script>

    <!-- The XML data provider -->
    <mx:XML format="e4x" id="menuData">
        <root>
            <menuitem label="MenuItem A"/>
            <menuitem label="MenuItem B" type="check" toggled="true"/>
            <menuitem label="MenuItem C" enabled="false"/>
            <menuitem type="separator"/>
            <menuitem label="MenuItem D">
                <menuitem label="SubMenuItem D-1"/>
                <menuitem label="SubMenuItem D-2"/>
                <menuitem label="SubMenuItem D-3"/>
            </menuitem>
        </root>
    </mx:XML>

    <!-- Button control to create and open the menu. -->
    <mx:Button x="300" y="10"
        label="Open Menu"
        click="show();"/>
</mx:WindowedApplication>
```

Specifying menu item keyboard equivalents

You can specify a key equivalent (sometimes called an accelerator) for a menu command. When the key or key combination is pressed the FlexNativeMenu dispatches an `itemClick` event, as though the user had selected the menu item. The key equivalent string is automatically displayed beside the menu item name in the menu. The format depends on the user's operating system and system preferences. In order for the command to be invoked, the menu containing the command must be part of the application menu (OS X) or the window menu of the active window (Windows).

***Note:** Key equivalents are only triggered for application and window menus. If you add a key equivalent to a context or pop-up menu, the key equivalent is displayed in the menu label but the associated menu command is never invoked.*

Contents

About key equivalents

A key equivalent consists of two parts:

Primary key A string containing the character that serves as the key equivalent. If a data provider object has a `keyEquivalent` field, the DefaultDataDescriptor automatically uses that value as the key equivalent. You can specify an alternative data provider field by setting the FlexNativeMenu component's `keyEquivalentField` property. You can specify a function to use to determine the key equivalent by setting the FlexNativeMenu component's `keyEquivalentFunction` property.

Modifier keys One or more modifier keys that are also part of the key equivalent combination, such as the control key, shift key, command key, and so forth. If a data provider item includes an `altKey`, `cmdKey`, `ctrlKey`, or `shiftKey` object field or XML attribute set to `true`, the specified key or keys become part of the key equivalent combination, and the entire key combination must be pressed to trigger the command. Alternatively, you can specify a function for the FlexNativeMenu component's `keyEquivalentModifiersFunction`, and that function is called to determine the key equivalent modifiers for each data provider item.

If you specify more than one key equivalent modifier, all the specified modifiers must be pressed in order to trigger the command. For instance, for the menu item generated from the following XML the key equivalent combination is Control+Shift+A (rather than Control+A OR Shift+A):

```
<menuitem label="Select All" keyEquivalent="a" ctrlKey="true" shiftKey="true"/>
```

Note that this can result in impossible key combinations if the menu item specifies multiple modifiers that are only available on one operating system. For example, the following item results in a menu item with the key equivalent Command-Shift-G:

```
<menuitem label="Ungroup" keyEquivalent="g" cmdKey="true" shiftKey="true"/>
```

On Mac OS X, this command works as expected. On Windows, the key equivalent Shift+G is displayed in the menu. However, pressing Shift+G does not trigger the command because the Command key is still considered a required part of the command, even though that key doesn't exist in Windows.

To use different key combinations for the same menu item on different platforms, you can specify a `keyEquivalentModifiersFunction` function for the FlexNativeMenu instance. This function can provide alternative logic for processing the menu item data. For an example using the `keyEquivalentModifiersFunction`, see "Example: Using custom logic for multi-platform key equivalent menu commands" on page 57.

Example: FlexNativeMenu key equivalent commands

The following example uses a FlexNativeMenu to define a menu that includes keyboard equivalents for the menu commands. Note that while this example only uses keys and modifier keys that are available on Windows and Mac OS X, it uses the Control key as a modifier on both platforms. However, the Command key would be more in line with the common convention on Mac OS X. For an example that uses the `keyEquivalentModifiersFunction` property to create menus that use the common cross-platform conventions, see “Example: Using custom logic for multi-platform key equivalent menu commands” on page 57.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
    <mx:menu>
        <mx:FlexNativeMenu dataProvider="{menuData}"
            labelField="@label"
            keyEquivalentField="@keyEquivalent"
            showRoot="false"
            itemClick="trace('click:', event.label);"/>
    </mx:menu>
    <mx:XML format="e4x" id="menuData">
        <root>
            <menuitem label="File">
                <menuitem label="New" keyEquivalent="n" ctrlKey="true"/>
                <menuitem label="Open" keyEquivalent="o" ctrlKey="true"/>
                <menuitem label="Save" keyEquivalent="s" ctrlKey="true"/>
                <menuitem label="Save As..." keyEquivalent="s" ctrlKey="true"
shiftKey="true"/>
                <menuitem label="Close" keyEquivalent="w" ctrlKey="true"/>
            </menuitem>
            <menuitem label="Edit">
                <menuitem label="Cut" keyEquivalent="x" ctrlKey="true"/>
                <menuitem label="Copy" keyEquivalent="c" ctrlKey="true"/>
                <menuitem label="Paste" keyEquivalent="v" ctrlKey="true"/>
            </menuitem>
        </root>
    </mx:XML>
</mx:WindowedApplication>
```

Example: Using custom logic for multi-platform key equivalent menu commands

The following example creates the same menu structure as the previous example. However, instead of using the same keyboard combination (for example, Control-O) regardless of the user’s operating system, in this example a `keyEquivalentModifiersFunction` function is defined for the FlexNativeMenu. The function is used to create keyboard equivalents that follow platform conventions by using the Control key on Windows but substituting the Command key on Mac OS X. In the data provider data, the `ctrlKey="true"` attribute is still used. The function that determines the key equivalent modifiers uses the value of the `ctrlKey` field or XML attribute to specify the Control key on Windows and the Command key on OS X, and if the `ctrlKey` attribute is `false` (or not specified) then neither modifier is applied.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    initialize="init();">

    <mx:menu>
        <mx:FlexNativeMenu dataProvider="{menuData}"
            labelField="@label"
            keyEquivalentField="@keyEquivalent"
            keyEquivalentModifiersFunction="keyEquivalentModifiers"
            showRoot="false"
            itemClick="trace('click:', event.label);"/>
```

```
</mx:menu>
<mx:Script>
    <![CDATA[
        import flash.system.Capabilities;
        import flash.ui.Keyboard;

        private var isWin:Boolean;
        private var isMac:Boolean;

        private function init():void
        {
            isWin = (Capabilities.os.indexOf("Windows") >= 0);
            isMac = (Capabilities.os.indexOf("Mac OS") >= 0);
        }

        private function keyEquivalentModifiers(item:Object):Array
        {
            var result:Array = new Array();

            var keyEquivField:String = menu.keyEquivalentField;
            var altKeyField:String;
            var ctrlKeyField:String;
            var shiftKeyField:String;
            if (item is XML)
            {
                altKeyField = "@altKey";
                ctrlKeyField = "@ctrlKey";
                shiftKeyField = "@shiftKey";
            }
            else if (item is Object)
            {
                altKeyField = "altKey";
                ctrlKeyField = "ctrlKey";
                shiftKeyField = "shiftKey";
            }

            if (item[keyEquivField] == null || item[keyEquivField].length == 0)
            {
                return result;
            }

            if (item[altKeyField] != null && item[altKeyField] == true)
            {
                if (isWin)
                {
                    result.push(Keyboard.ALTERNATE);
                }
            }

            if (item[ctrlKeyField] != null && item[ctrlKeyField] == true)
            {
                if (isWin)
                {
                    result.push(Keyboard.CONTROL);
                }
                else if (isMac)
                {
                    result.push(Keyboard.COMMAND);
                }
            }
```

```
                if (item[shiftKeyField] != null && item[shiftKeyField] == true)
                {
                    result.push(Keyboard.SHIFT);
                }

                return result;
            }
        ]]>
    </mx:Script>
    <mx:XML format="e4x" id="menuData">
        <root>
            <menuitem label="File">
                <menuitem label="New" keyEquivalent="n" ctrlKey="true"/>
                <menuitem label="Open" keyEquivalent="o" ctrlKey="true"/>
                <menuitem label="Save" keyEquivalent="s" ctrlKey="true"/>
                <menuitem label="Save As..."
                    keyEquivalent="s"
                    ctrlKey="true"
                    shiftKey="true"/>
                <menuitem label="Close" keyEquivalent="w" ctrlKey="true"/>
            </menuitem>
            <menuitem label="Edit">
                <menuitem label="Cut" keyEquivalent="x" ctrlKey="true"/>
                <menuitem label="Copy" keyEquivalent="c" ctrlKey="true"/>
                <menuitem label="Paste" keyEquivalent="v" ctrlKey="true"/>
            </menuitem>
        </root>
    </mx:XML>
</mx:WindowedApplication>
```

Specifying menu item mnemonics

A menu item mnemonic is a key associated with a menu item which, when pressed while the menu is displayed, is equivalent to selecting that menu item with the mouse. Typically, the operating system indicates a menu item's mnemonic by underlining that character in the name of the menu item. Mnemonics for menu items are supported in Windows. In Mac OS X, when a menu is activated a user types the first letter or letters of a menu item's label, then presses return to select the item. For more details about the behavior of mnemonics, see "Mnemonics" on page 131.

The simplest way to specify a mnemonic for a menu item in a FlexNativeMenu component is to include an underscore character ("_") in the menu item's label field, immediately to the left of the letter that serves as the mnemonic for that menu item. For instance, if the following XML node is used in a data provider for a FlexNativeMenu, the mnemonic for the command is the first character of the second word (the letter "A"):

```
<menuitem label="Save _As"/>
```

When the native menu is created, the underscore character is not included in the label. Instead, the character following the underscore becomes the mnemonic for the menu item. To include a literal underscore character in a menu item's name, use two underscore characters ("__"). This sequence is converted to one underscore in the menu item label.

As an alternative to using underscore characters in label names, you can provide an integer index position for the mnemonic character in a `mnemonicIndex` field in the data provider objects. You can also use another Object property or XML attribute by setting the FlexNativeMenu component's `mnemonicIndexField` property. To use complex logic for assigning mnemonics, you can specify a function for the FlexNativeMenu component's `mnemonicIndexFunction` property. Each of these properties provides a mechanism to define an integer (zero-based) index position for the menu items' mnemonics.

Handling FlexNativeMenu control events

User interaction with a FlexNativeMenu is event-driven. When the user selects a menu item or opens a menu or submenu, the menu dispatches an event. You can register event listeners to define the actions that are carried out in response to the user's selection. Event handling with the FlexNativeMenu component shares similarities with other Flex menu components, but also has key differences. For information about Flex menu component events, see “Menu-based control events” on page 353. For detailed information on events and how to use them, see “Using Events” on page 61.

The FlexNativeMenu component defines two specific events, both of which dispatch event objects that are instances of the FlexNativeMenuEvent class:

Event	Description
itemClick	(FlexNativeMenuEvent.ITEM_CLICK) Dispatched when a user selects an enabled menu item of type normal or check. This event is not dispatched when a user selects a menu item that opens a submenu or a disabled menu item.
menuShow	(FlexNativeMenuEvent.MENU_SHOW) Dispatched when the entire menu or a submenu opens (including a top-level menu of an application or window menu).

The event object passed to the event listener is of type FlexNativeMenuEvent and contains the following menu-specific properties:

Property	Description
`index`	The index of the item in the menu or submenu that contains it. Only available for the `itemClick` event.
`item`	The item in the data provider for the menu item associated with the event. Only available for the `itemClick` event.
`label`	The label of the item. Only available for the `itemClick` event.
`nativeMenu`	A reference to the underlying NativeMenu object where the event occurred.
`nativeMenuItem`	A reference to the underlying NativeMenuItem object that triggered the event. Only available for the `itemClick` event.

To access properties of an object-based menu item, you specify the `item` property of the event object, as follows:

```
ta1.text = event.item.extraData;
```

To access attributes of an E4X XML-based menu item, you specify the menu item attribute name in E4X syntax, as follows:

```
ta1.text = event.item.@extraData;
```

***Note:** If you set an event listener on a submenu of a FlexNativeMenu component, and the menu data provider's structure changes (for example, if an element is removed), the event listener might no longer exist. To ensure that the event listener is available when the data provider structure changes, use the events of the FlexNativeMenu control, not a submenu.*

The standard approach to handling FlexNativeMenu events is to register an event listener with the FlexNativeMenu component. Any time an individual menu item is selected or submenu is opened, the FlexNativeMenu dispatches the appropriate event. Your listener code can use the event object's `item` property or other properties to determine on which menu item the interaction occurred, and perform actions in response.

The following example lets you experiment with FlexNativeMenu control events. It lets you display two menus, one with an XML data provider and one with an Array data provider. A TextArea control displays information about each event as a user opens the menus, opens submenus, and selects menu items. The example shows some of the differences in how you handle XML and object-based menus. It also indicates some of the types of information that are available about each FlexNativeMenu event.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">

    <mx:Script>
        <![CDATA[
            import mx.events.FlexNativeMenuEvent;

            import mx.controls.FlexNativeMenu;
            import mx.events.FlexNativeMenuEvent;

            // The event listener that opens the menu with an XML data
            // provider and adds event listeners for the menu.
            private function createAndShow():void
            {
                ta1.text="";
                xmlBasedMenu.addEventListener(FlexNativeMenuEvent.ITEM_CLICK,
menuShowInfo);
                xmlBasedMenu.addEventListener(FlexNativeMenuEvent.MENU_SHOW,
menuShowInfo);
                xmlBasedMenu.display(stage, 225, 10);
            }

            // The event listener for the xml-based menu events.
            // Retain information on all events for a menu instance.
            private function menuShowInfo(event:FlexNativeMenuEvent):void
            {
                ta1.text = "event.type: " + event.type;

                // The label field is null for menuShow events.
                ta1.text += "\nevent.label: " + event.label;

                // The index value is -1 for menuShow events.
                ta1.text+="\nevent.index: " + event.index;

                // The item field is null for menuShow events.
                if (event.item != null)
                {
                    ta1.text += "\nItem label: " + event.item.@label
                    ta1.text += "\nItem toggled: " + event.item.@toggled;
                    ta1.text += "\nItem type: " + event.item.@type;
                }
            }

            // The event listener that creates an object-based menu
            // and adds event listeners for the menu.
            private function createAndShow2():void
            {
                ta1.text="";
                objectBasedMenu.addEventListener(FlexNativeMenuEvent.ITEM_CLICK,
menuShowInfo2);
                objectBasedMenu.addEventListener(FlexNativeMenuEvent.MENU_SHOW,
menuShowInfo2);
                objectBasedMenu.display(stage, 225, 10);
            }

            // The event listener for the object-based Menu events.
            private function menuShowInfo2(event:FlexNativeMenuEvent):void
            {
                ta1.text = "event.type: " + event.type;
                // The label field is null for menuShow events.
                ta1.text += "\nevent.label: " + event.label;

                // The index value is -1 for menuShow events.
                ta1.text += "\nevent.index: " + event.index;

                // The item field is null for menuShow events.
                if (event.item)
                {
                    ta1.text += "\nItem label: " + event.item.label
                    ta1.text += "\nItem toggled: " + event.item.toggled;
                    ta1.text += "\ntype: " + event.item.type;
                }
            }

            // The object-based data provider, an Array of objects.
            // Its contents are identical to that of the XML data provider.
            [Bindable]
            public var objMenuData:Array = [
                {label: "MenuItem A", children: [
                    {label: "SubMenuItem A-1", enabled: false},
                    {label: "SubMenuItem A-2"}
                ]},
                {label: "MenuItem B", type: "check", toggled: true},
                {label: "MenuItem C", type: "check", toggled: false},
                {type: "separator"},
                {label: "MenuItem D", children: [
                    {label: "SubMenuItem D-1"},
                    {label: "SubMenuItem D-2"},
                    {label: "SubMenuItem D-3"}
                ]}
            ];

        ]]>
    </mx:Script>

    <!-- The XML-based menu data provider.
    The <mx:XML tag requires a single root. -->
    <mx:XML id="xmlMenuData">
        <xmlRoot>
            <menuitem label="MenuItem A" >
                <menuitem label="SubMenuItem A-1" enabled="false"/>
                <menuitem label="SubMenuItem A-2"/>
            </menuitem>
            <menuitem label="MenuItem B" type="check" toggled="true"/>
            <menuitem label="MenuItem C" type="check" toggled="false"/>
            <menuitem type="separator"/>
            <menuitem label="MenuItem D">
                <menuitem label="SubMenuItem D-1"/>
                <menuitem label="SubMenuItem D-2"/>
                <menuitem label="SubMenuItem D-3"/>
            </menuitem>
        </xmlRoot>
    </mx:XML>

    <mx:FlexNativeMenu id="xmlBasedMenu"
        showRoot="false"
        labelField="@label"
        dataProvider="{xmlMenuData}"/>

    <mx:FlexNativeMenu id="objectBasedMenu"
```

```
        dataProvider="{objMenuData}"/>

    <!-- Button controls to open the menus. -->
    <mx:Button x="10" y="5"
        label="Open XML Popup"
        click="createAndShow();"/>
    <mx:Button x="10" y="35"
        label="Open Object Popup"
        click="createAndShow2();"/>
    <!-- Text area to display the event information -->
    <mx:TextArea x="10" y="70"
        width="200" height="250"
        id="ta1"/>
</mx:WindowedApplication>
```

About window containers

Flex container components define the content, sizing, and positioning for a specific part of an application. For AIR applications, Flex includes two specific window components that serve as containers whose content area is an operating system window. Both the WindowedApplication and the Window components can be used to define the contents of an operating system window. They also provide the means to define and control characteristics of the window itself, such as the window's size, its position on the user's screen, and the presence of window chrome.

Contents

Controlling window chrome

The WindowedApplication component and the Window component allow you to control the presence and the appearance of the window chrome.

These components provide the following window controls:

- A title bar
- A minimize button
- A maximize button
- A close button

How these controls are represented depends on the setting of the `systemChrome` value, which is defined in the application.xml file for a WindowedApplication component, or in the Window component's `systemChrome` property. If `systemChrome` is set to "standard" in the application.xml file (or `flash.display.NativeWindowSystemChrome.STANDARD` in ActionScript) the operating system renders the chrome for the title bar and border of the window. However, if `systemChrome` is set to "none" (`NativeWindowSystemChrome.NONE`) the Flex Window or WindowedApplication component draws its own title bar, including the buttons listed previously.

The title bar area of the window chrome (either the operating system or Flex chrome) includes a title message and an icon that can be set and modified programmatically. In addition to the operating system chrome, Flex adds additional common window chrome elements, including a status bar with a definable status message and a resize gripper in the bottom-right corner of the window. All the Flex-drawn chrome can be hidden as a group by setting the `showFlexChrome` property to `false`. A single chrome element can be hidden by setting the `showTitleBar`, `showGripper`, or `showStatusBar` property to `false`.

The window that a WindowedApplication or Window component defines conforms to the standard behavior of the operating system. The user can move the window by dragging the title bar and resize the window by dragging on any side or corner of the window. The components also include a set of properties that allow you to control window sizing, including `minimumHeight`, `minimumWidth`, `maximumHeight`, and `maximumWidth`.

WindowedApplication container

The WindowedApplication container defines an application container that you use to create Flex applications for AIR that use the native operating system chrome. The WindowedApplication container adds window-related functionality and desktop application-specific functionality to the Flex Application container, which you can use when you build AIR applications.

The WindowedApplication container serves two roles. On one hand, it provides the entry point into the main application, which in turn executes other application logic. In that sense it serves as the core of the entire application, just as the Application component does for a browser-based Flex application. On the other hand, the WindowedApplication container represents the first native window of the application. If the application only uses one native window, the WindowedApplication is the base stage that contains all other content. If your application opens additional native windows, each window has its own stage and display list. The native window defined by the WindowedApplication is no different from any other application window in this respect. This is different from a browser-based Flex application, where all of an application's windows are drawn by Flex within the same stage (the Application container). For example, in a Flex AIR application, registering a `keyDown` event listener on the WindowedApplication component only dispatches events when a key is pressed while the initial window has focus. If the application has multiple native windows and another of the windows has focus when the key is pressed, the event is not dispatched. This behavior differs from a non-AIR Flex application, in which a `keyDown` listener registered with the Application container receives notification of all keypresses while the application has focus.

For more information on the WindowedApplication container, see the Flex 3 Language Reference.

Creating and using a WindowedApplication container

The `<mx:WindowedApplication>` container component defines an AIR application object that includes its own window controls. In an MXML AIR application, a `<mx:WindowedApplication>` tag replaces the `<mx:Application>` tag.

By default, the WindowedApplication component creates an application window for which `systemChrome` is set to `NativeWindowSystemChrome.STANDARD` and `visible` is set to `true`. These settings are made in the application.xml file for the AIR application. To eliminate the system chrome and window controls that the WindowedApplication component creates by default, in the `<mx:WindowedApplication>` tag set the `useFlexChrome` attribute to `false`, and in the application.xml file set `systemChrome` to `none`.

WindowedApplication container example

The following application shows a simple use of the WindowedApplication container:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute">
    <mx:Text text="Hello World" />
</mx:WindowedApplication>
```

Window container

The Window component is a Flex container that is used to define the content and layout of operating system windows that are opened after an application launches. In other words, it is used for windows other than the initial or main window of the application, which is a WindowedApplication component. In addition to the functionality that the Window component shares with the WindowedApplication component, a Window component allows you to define the main characteristics of the window. The characteristics you can specify include the type of window, the type of chrome, whether certain actions (such as resizing and maximizing) are permitted for the window, and more. These characteristics are accessed as properties that can be set when the component is initially created, before the actual operating system window is displayed. However, once the actual window is opened, the properties can no longer be set and can only be read.

For more information about the Window container, see the Flex 3 Language Reference.

Creating and using a Window container

The `<mx:Window>` container component defines an AIR application object that includes its own window controls. In an MXML AIR application, a `<mx:Window>` tag is used as the top-level tag of an MXML component, with the window's content defined in the body of the MXML component document. However, unlike other MXML components, a Window-based component cannot be used in another MXML document. Instead, you create an instance of the MXML component in ActionScript.

Because several of the properties of the Window component can only be set before the window is opened, they can be set as properties in the `<mx:Window>` MXML tag. They can also be set using ActionScript, either in an `<mx:Script>` block in the window's MXML document or in code that creates an instance of the window.

Once the window's initial properties are set, you call the Window component's `open()` method to cause the operating system window to appear on the user's display.

Window container example

The following example shows a basic use of the Window component. The example includes two MXML files. The first uses a WindowedApplication container and is the initial window of the application. The second uses the Window container to define a secondary window for the application. In this example, the main window simulates a "splash screen" for the application. After a set time (4 seconds) it closes the splash screen and opens the second window. In order to make a splash screen window with no window chrome, in the application.xml file the `systemChrome` tag is set to `none`.

The following code defines the main application MXML file, which contains the initial window (the splash screen) that opens automatically when the application is run:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
showFlexChrome="false" creationComplete="init();">
    <mx:Script>
        <![CDATA[
            private const LOAD_DELAY:int = 4;
            private var timeElapsed:int = 0;
            private var loadTimer:Timer;

            private var docWindow:DocumentWindow;

            private function init():void
            {
                // center the window on the screen
                var screenBounds:Rectangle = Screen.mainScreen.bounds;
                nativeWindow.x = (screenBounds.width - nativeWindow.width) / 2;
                nativeWindow.y = (screenBounds.height - nativeWindow.height) / 2;

                // start the timer, which simulates a loading delay
                loadTimer = new Timer(1000);
                loadTimer.addEventListener(TimerEvent.TIMER, incrementTime);
                loadTimer.start();

                updateStatus();
            }

            private function incrementTime(event:TimerEvent):void
            {
                timeElapsed++;

                updateStatus();

                // if the loading delay has passed, stop the timer,
                // close the splash screen, and open the document window
                if ((LOAD_DELAY - timeElapsed) == 0)
                {
                    loadTimer.stop();
                    loadTimer.removeEventListener(TimerEvent.TIMER, incrementTime);
                    loadTimer = null;

                    nativeWindow.close();

                    // open a new instance of the document window
                    docWindow = new DocumentWindow();
                    docWindow.open();
                }
            }

            private function updateStatus():void
            {
                var timeRemaining:uint = LOAD_DELAY - timeElapsed;
                var timeRemainingMsg:String = timeRemaining.toString() + " second";
                if (timeRemaining != 1) { timeRemainingMsg += "s"; }
                timeRemainingMsg += " remaining.";

                loadStatusMessage.text = "initializing... " + timeRemainingMsg;
            }
        ]]>
    </mx:Script>
    <mx:VBox horizontalCenter="0" verticalCenter="0">
        <mx:Text text="My Splash Screen" fontFamily="Courier New" fontSize="36"/>
        <mx:Text id="loadStatusMessage" text="initializing..."/>
    </mx:VBox>
</mx:Application>
```

The `incrementTime()` method is called each second and when the appropriate time is reached, a DocumentWindow instance is created and its `open()` method is called. The DocumentWindow class is defined in a separate MXML document. Its base MXML tag is the `<mx:Window>` tag, so it is a subclass of the Window class (the Window component). Here is the source code for the DocumentWindow MXML file:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<mx:Window xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    title="Document window"
    width="550" height="450">
    <mx:Text text="This is a document window." horizontalCenter="0" verticalCenter="0"/>
</mx:Window>
```

Part 4: Application development essentials

Chapter 10: AIR security

Although the Adobe® AIR™ security model is an evolution of the Adobe® Flash® Player security model, the security contract is different from the security contract applied to content in a browser. This contract offers developers a secure means of broader functionality for rich experiences with freedoms that would be inappropriate for a browser-based application.

AIR applications run with the same user privileges as native applications. In general, these privileges allow for broad access to operating system capabilities such as reading and writing files, starting applications, drawing to the screen, and communicating with the network. Operating system restrictions that apply to native applications, such as user-specific privileges, equally apply to AIR applications.

AIR applications are written using either compiled bytecode (SWF content) or interpreted script (JavaScript, HTML) so that the runtime provides memory management. This minimizes the chances of AIR applications being affected by vulnerabilities related to memory management, such as buffer overflows and memory corruption. These are some of the most common vulnerabilities affecting desktop applications written in native code.

Contents

Installation and updates

AIR applications are distributed via AIR installer files which use the `air` extension. When Adobe AIR is installed and an AIR installer file is opened, the runtime administers the installation process.

***Note:** Developers can specify a version, and application name, and a publisher source, but the initial application installation workflow itself cannot be modified. This restriction is advantageous for users because all AIR applications share a secure, streamlined, and consistent installation procedure administered by the runtime. If application customization is necessary, it can be provided when the application is first executed.*

Contents

- “Updating Adobe AIR” on page 72
- “Updating AIR applications” on page 72
- “Uninstalling an AIR application” on page 72
- “Uninstalling Adobe AIR” on page 72
- “Windows registry settings for administrators” on page 72

Runtime installation location

AIR applications first require the runtime to be installed on a user's computer, just as SWF files first require the Flash Player browser plug-in to be installed.

The runtime is installed to the following location on a user's computer:

- Mac OS: `/Library/Frameworks/`
- Windows: `C:\Program Files\Common Files\Adobe AI`

On Mac OS, to install an updated version of an application, the user must have adequate system privileges to install to the application directory. On Windows, a user must have administrative privileges.

The runtime can be installed in two ways: using the seamless install feature (installing directly from a web browser) or via a manual install. For more information, see “Distributing, Installing, and Running AIR applications” on page 331.

Seamless install (runtime and application)

The seamless install feature provides developers with a streamlined installation experience for users who do not have Adobe AIR installed yet. In the seamless install method, the developer creates a SWF file that presents the application for installation. When a user clicks in the SWF file to install the application, the SWF file attempts to detect the runtime. If the runtime cannot be detected it is installed, and the runtime is activated immediately with the installation process for the developer's application.

Manual install

Alternatively, the user can manually download and install the runtime before opening an AIR file. The developer can then distribute an AIR file by different means (for instance, via e-mail or an HTML link on a website). When the AIR file is opened, the runtime begins to process the application installation.

For more information on this process, see “Distributing, Installing, and Running AIR applications” on page 331.

Application installation flow

The AIR security model allows users to decide whether to install an AIR application. The AIR install experience provides several improvements over native application install technologies that make this trust decision easier for users:

- The runtime provides a consistent installation experience on all operating systems, even when an AIR application is installed from a link in a web browser. Most native application install experiences depend upon the browser or other application to provide security information, if it is provided at all.
- The AIR application install experience identifies the source of the application and information about what privileges are available to the application (if the user allows the installation to proceed).
- The runtime administers the installation process of an AIR application. An AIR application cannot manipulate the installation process the runtime uses.

In general, users should not install any desktop application that comes from a source that they do not trust, or that cannot be verified. The burden of proof on security for native applications is equally true for AIR applications as it is for other installable applications.

Application destination

The installation directory can be set using one of the following two options:

1 The user customizes the destination during installation. The application installs to wherever the user specifies.

2 If the user does not change the install destination, the application installs to the default path as determined by the runtime:

- Mac OS: `~/Applications/`
- Windows XP and earlier: `C:\Program Files\`
- Windows Vista: `~/Apps/`

If the developer specifies an `installFolder` setting in the application descriptor file, the application is installed to a subpath of this directory.

The AIR file system

The install process for AIR applications copies all files that the developer has included within the AIR installer file onto the user's local computer. The installed application is composed of:

- Windows: A directory containing all files included in the AIR installer file. The runtime also creates an exe file during the installation of the AIR application.
- Mac OS: An `app` file that contains all of the contents of the AIR installer file. It can be inspected using the "Show Package Contents" option in Finder. The runtime creates this app file as part of the installation of the AIR application.

An AIR application is run by:

- Windows: Running the .exe file in the install folder, or a shortcut that corresponds to this file (such as a shortcut on the Start Menu or desktop).
- Mac OS: Running the .app file or an alias that points to it.

The application file system also includes subdirectories related to the function of the application. For example, information written to encrypted local storage is saved to a subdirectory in a directory named after the application identifier of the application.

AIR application storage

AIR applications have privileges to write to any location on the user's hard drive; however, developers are encouraged to use the `app-storage:/` path for local storage related to their application. Files written to `app-storage:/` from an application are located in a standard location depending on the user's operating system:

- On Mac OS: the storage directory of an application is `<appData>/<appId>/Local Store/` where `<appData>` is the user's “preferences folder,” typically `/Users/<user>/Library/Preferences`
- On Windows: the storage directory of an application is `<appData>\<appId>\Local Store\` where `<appData>` is the user's CSIDL_APPDATA “Special Folder” typically `C:\Documents and Settings\<userName>\Application Data`

You can access the application storage directory via the air.File.applicationStorageDirectory property. You can access its contents using the resolvePath() method of the File class. For details, see “Working with the file system” on page 147.

Updating Adobe AIR

When the user installs an AIR application that requires an updated version of the runtime, the runtime automatically installs the required runtime update.

To update the runtime, a user must have administrative privileges for the computer.

Updating AIR applications

Development and deployment of software updates are one of the biggest security challenges facing native code applications. The AIR API provides a mechanism to improve this: the `Updater.update()` method can be invoked upon launch to check a remote location for an AIR file. If an update is appropriate, the AIR file is downloaded, installed, and the application restarts. Developers can use this class not only to provide new functionality but also respond to potential security vulnerabilities.

***Note:** Developers can specify the version of an application by setting the version property of the application descriptor file. AIR does not interpret the version string in any way. Thus version "3.0" is not assumed to be more current than version "2.0." It is up to the developer to maintain meaningful versioning. For details, see "Defining properties in the application descriptor file" on page 89.*

Uninstalling an AIR application

A user can uninstall an AIR application:

- On Windows: Using the Add/Remove Programs panel to remove the application.
- On Mac OS: Deleting the app file from the install location.

Removing an AIR application removes all files in the application directory. However, it does not remove files that the application may have written to outside of the application directory. Removing AIR applications does not revert changes the AIR application has made to files outside of the application directory.

Uninstalling Adobe AIR

AIR can be uninstalled:

- On Windows: by running Add/Remove Programs from the Control Panel, selecting Adobe AIR and selecting "Remove".
- On Mac OS: by running the Adobe AIR Uninstaller application in the Applications directory.

Windows registry settings for administrators

On Windows, administrators can configure a machine to prevent (or allow) AIR application installation and runtime updates. These settings are contained in the Windows registry under the following key: HKLM\Software\Policies\Adobe\AIR. They include the following:

Registry setting	**Description**
AppInstallDisabled	Specifies that AIR application installation and uninstallation are allowed. Set to 0 for "allowed," set to 1 for "disallowed."
UntrustedAppInstallDisabled	Specifies that installation of untrusted AIR applications (applications that do not includes a trusted certificate) is allowed (see "Digitally signing an AIR file" on page 339). Set to 0 for "allowed," set to 1 for "disallowed."
UpdateDisabled	Specifies that updating the runtime is allowed, either as a background task or as part of an explicit installation. Set to 0 for "allowed," set to 1 for "disallowed."

Sandboxes

AIR provides a comprehensive security architecture that defines permissions accordingly to each file in an AIR application, both internal and external. Permissions are granted to files according to their origin, and are assigned into logical security groupings called sandboxes.

Contents

About the AIR application sandboxes

The runtime security model of sandboxes is composed of the Flash Player security model with the addition of the application sandbox. Files that are not in the application sandbox have security restrictions similar to those specified by the Flash Player security model.

The runtime uses these security sandboxes to define the range of data that code may access and the operations it may execute. To maintain local security, the files in each sandbox are isolated from the files of other sandboxes. For example, a SWF file loaded into an AIR application from an external Internet URL is placed into a remote sandbox, and does not by default have permission to script into files that reside in the application directory, which are assigned to the application sandbox.

The following table describes each type of sandbox:

Sandbox	Description
application	The file resides in the application directory and operates with the full set of AIR privileges.
remote	The file is from an Internet URL, and operates under domain-based sandbox rules analogous to the rules that apply to remote files in Flash Player. (There are separate remote sandboxes for each network domain, such as http://www.example.com and https://foo.example.org.)
local-trusted	The file is a local file and has the user has designated it as trusted , using either the Settings Manager or a Flash Player trust configuration file. The file can both read from local data sources and communicate with the Internet, but does not have the full set of AIR privileges.
local-with-networking	The file is a local SWF file published with a networking designation, but has not been explicitly trusted by the user. The file can communicate with the Internet but cannot read from local data sources. This sandbox is only available to SWF content.
local-with-filesystem	The file is a local scripting file that was not published with a networking designation and has not been explicitly trusted by the user. This includes JavaScript files that have not been trusted. The file can read from local data sources but cannot communicate with the Internet.

This topic focuses primarily on the application sandbox and its relationship to other sandboxes in the AIR application. Developers that use content assigned to other sandboxes should read further documentation on the Flash Player security model. See the "Flash Player Security" chapter in the *Programming ActionScript 3.0* (http://www.adobe.com/go/flashCS3_progAS3_security) documentation and the *Adobe Flash Player 9 Security white paper* (http://www.adobe.com/go/fp9_0_security).

The application sandbox

When an application is installed, all files included within an AIR installer file are installed onto the user's computer into an application directory. Developers can reference this directory in code through the `app:/` URL scheme (see "Using AIR URL schemes in URLs" on page 326). All files within the application directory tree are assigned to the application sandbox when the application is run. Content in the application sandbox is blessed with the full privileges available to an AIR application, including interaction with the local file system.

Many AIR applications use only these locally installed files to run the application. However, AIR applications are not restricted to just the files within the application directory — they can load any type of file from any source. This includes files local to the user's computer as well as files from available external sources, such as those on a local network or on the Internet. File type has no impact on security restrictions; loaded HTML files have the same security privileges as loaded SWF files from the same source.

Content in the application security sandbox has access to AIR APIs that content in other sandboxes are prevented from using. For example, the `NativeApplication.nativeApplication.applicationDescriptor` property, which returns the contents of the application descriptor file for the application, is restricted to content in the application security sandbox. Another example of a restricted API is the FileStream class, which contains methods for reading and writing to the local file system.

ActionScript APIs that are only available to content in the application security sandbox are indicated with the AIR logo in the *ActionScript 3.0 Language Reference for Adobe AIR*. Using these APIs in other sandboxes causes the runtime to throw a SecurityError exception.

For HTML content (in an HTMLLoader object), all AIR JavaScript APIs (those that are available via the `window.runtime` property, or via the `air` object when using the AIRAliases.js file) are available to content in the application security sandbox. HTML content in another sandbox does not have access to the `window.runtime` property, so this content cannot access the AIR APIs.

JavaScript and HTML restrictions

For HTML content in the application security sandbox, there are limitations on using APIs that can dynamically transform strings into executable code after the code is loaded. This is to prevent the application from inadvertently injecting (and executing) code from non-application sources (such as potentially insecure network domains). An example is the use of the `eval()` function. For details, see "Code restrictions for content in different sandboxes" on page 77.

Restrictions on img tags in ActionScript text field content

To prevent possible phishing attacks, `img` tags in HTML content in ActionScript TextField objects are ignored in SWF content in the application security sandbox.

Restrictions on asfunction

Content in the application sandbox cannot use the asfunction protocol in HTML content in ActionScript 2.0 text fields.

No access to the cross-domain persistent cache

SWF content in the application sandbox cannot use the cross-domain cache, a feature that was added to Flash Player 9 Update 3. This feature lets Flash Player persistently cache Adobe platform component content and reuse it in loaded SWF content on demand (eliminating the need to reload the content multiple times).

Privileges of content in non-application sandboxes

Files loaded from a network or Internet location are assigned to the `remote` sandbox. Files loaded from outside the application directory are assigned to either the `local-with-filesystem`, `local-with-networking`, or the `local-trusted` sandbox; this depends on how the file was created and if the user has explicitly trusted the file through the Flash Player Global Settings Manager. For details, see http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html.

JavaScript and HTML restrictions

Unlike content in the application security sandbox, JavaScript content in a non-application security sandbox *can* call the `eval()` function to execute dynamically generated code at any time. However, there are restrictions to JavaScript in a non-application security sandbox. These include:

- JavaScript code in a non-application sandbox does not have access to the `window.runtime` object, and as such this code cannot execute AIR APIs.
- By default, content in a non-application security sandbox cannot use XMLHttpRequest calls to load data from other domains other than the domain calling the request. However, application code can grant non-application content permission to do so by setting an `allowcrosscomainxhr` attribute in the containing frame or iframe. For more information, see "Scripting between content in different domains" on page 80.
- There are restrictions on calling the JavaScript `window.open()` method. For details, see "Restrictions on calling the JavaScript window.open() method" on page 80.

For details, see "Code restrictions for content in different sandboxes" on page 77.

Restrictions on loading CSS, frame, iframe, and img elements

HTML content in remote (network) security sandboxes can only load CSS, `frame`, `iframe`, and `img` content from remote domains (from network URLs).

HTML content in local-with-filesystem, local-with-networking, or local-trusted sandboxes can only load CSS, `frame`, `iframe`, and `img` content from local sandboxes (not from application or network URLs).

HTML security

The runtime enforces rules and provides mechanisms for overcoming possible security vulnerabilities in HTML and JavaScript. Content in the application sandbox and the non-application security sandbox (see "Sandboxes" on page 73) have different privileges. When loading content into an iframe or frame, the runtime provides a secure *sandbox bridge* mechanism that allows content in the frame or iframe to communicate securely with content in the application security sandbox.

This topic describes the AIR HTML security architecture and how to use iframes, frames, and the sandbox bridge to set up your application.

For more information, see "Avoiding security-related JavaScript errors" on page 260.

Contents

Overview on configuring your HTML-based application

Frames and iframes provide a convenient structure for organizing HTML content in AIR. Frames provide a means both for maintaining data persistence and for working securely with remote content.

Because HTML in AIR retains its normal, page-based organization, the HTML environment completely refreshes if the top frame of your HTML content "navigates" to a different page. You can use frames and iframes to maintain data persistence in AIR, much the same as you would for a web application running in a browser. Define your main application objects in the top frame and they persist as long as you don't allow the frame to navigate to a new page. Use child frames or iframes to load and display the transient parts of the application. (There are a variety of ways to maintain data persistence that can be used in addition to, or instead of, frames. These include cookies, local shared objects, local file storage, the encrypted file store, and local database storage.)

HTML in AIR retains its normal, blurred line between executable code and data. Because of this, AIR puts content in the top frame of the HTML environment into the application sandbox and restricts any operations, such as `eval()`, that can convert a string of text into an executable object. This restriction is enforced even when an application does not load remote content. To work securely with remote HTML content in AIR, you must use frames or iframes. Even if you don't load remote content, it may be more convenient to run content in a sandboxed child frame so that the content can be run with no restrictions on `eval()`. (Sandboxing may be necessary when using some JavaScript application frameworks.) For a complete list of the restrictions on JavaScript in the application sandbox, see "Code restrictions for content in different sandboxes" on page 77.

Because HTML in AIR retains its ability to load remote, possibly insecure content, AIR enforces a same-origin policy that prevents content in one domain from interacting with content in another. To allow interaction between application content and content in another domain, you can set up a bridge to serve as the interface between a parent and a child frame.

Setting up a parent-child sandbox relationship

AIR adds the `sandboxRoot` and `documentRoot` attributes to the HTML frame and iframe elements. These attributes let you treat application content as if it came from another domain:

Attribute	Description
sandboxRoot	The URL to use for determining the sandbox and domain in which to place the frame content. The `file:`, `http:`, or `https:` URL schemes must be used.
documentRoot	The URL from which to load the frame content. The `file:`, `app:`, or `app-storage:` URL schemes must be used.

The following example maps content installed in the sandbox subdirectory of the application to run in the remote sandbox and the www.example.com domain:

```
<iframe
    src="ui.html"
    sandboxRoot="http://www.example.com/local/"
    documentRoot="app:/sandbox/">
</iframe>
```

Setting up a bridge between parent and child frames in different sandboxes or domains

AIR adds the `childSandboxBridge` and `parentSandboxBridge` properties to the `window` object of any child frame. These properties let you define bridges to serve as interfaces between a parent and a child frame. Each bridge goes in one direction:

`childSandboxBridge` The `childSandboxBridge` property allows the child frame to expose an interface to content in the parent frame. To expose an interface, you set the `childSandbox` property to a function or object in the child frame. You can then access the object or function from content in the parent frame. The following example shows how a script running in a child frame can expose an object containing a function and a property to its parent:

```
var interface = {};
interface.calculatePrice = function(){
    return .45 + 1.20;
}
interface.storeID = "abc"
window.childSandboxBridge = interface;
```

If this child content is in an iframe assigned an `id` of `"child"`, you can access the interface from parent content by reading the `childSandboxBridge` property of the frame:

```
var childInterface = document.getElementById("child").childSandboxBridge;
air.trace(childInterface.calculatePrice()); //traces "1.65"
air.trace(childInterface.storeID)); //traces "abc"
```

`parentSandboxBridge` The `parentSandboxBridge` property allows the parent frame to expose an interface to content in the child frame. To expose an interface, you set the `parentSandbox` property of the child frame to a function or object in the parent frame. You can then access the object or function from content in the child frame. The following example shows how a script running in the parent frame can expose an object containing a save function to a child:

```
var interface = {};
interface.save = function(text){
    var saveFile = air.File("app-storage:/save.txt");
    //write text to file
}
document.getElementById("child").parentSandboxBridge = interface;
```

Using this interface, content in the child frame could save text to a file named save.txt. However, it would not have any other access to the file system. In general, application content should expose the narrowest possible interface to other sandboxes. The child content could call the save function as follows:

```
var textToSave = "A string.";
window.parentSandboxBridge.save(textToSave);
```

If child content attempts to set a property of the `parentSandboxBridge` object, the runtime throws a SecurityError exception. If parent content attempts to set a property of the `childSandboxBridge` object, the runtime throws a SecurityError exception.

Code restrictions for content in different sandboxes

As discussed in the introduction to this topic, "HTML security" on page 75, the runtime enforces rules and provides mechanisms for overcoming possible security vulnerabilities in HTML and JavaScript. This topic lists those restrictions. If code attempts to call these restricted APIs, the runtime throws an error with the message "Adobe AIR runtime security violation for JavaScript code in the application security sandbox."

For more information, see "Avoiding security-related JavaScript errors" on page 260.

Restrictions on using the JavaScript eval() function and similar techniques

For HTML content in the application security sandbox, there are limitations on using APIs that can dynamically transform strings into executable code after the code is loaded (after the `onload` event of the `body` element has been dispatched and the `onload` handler function has finished executing). This is to prevent the application from inadvertently injecting (and executing) code from non-application sources (such as potentially insecure network domains).

For example, if your application uses string data from a remote source to write to the innerHTML property of a DOM element, the string could include executable (JavaScript) code that could perform insecure operations. However, while the content is loading, there is no risk of inserting remote strings into the DOM.

One restriction is in the use of the JavaScript `eval()` function. Once code in the application sandbox is loaded and after processing of the onload event handler, you can only use the `eval()` function in limited ways. The following rules apply to the use of the `eval()` function *after* code is loaded from the application security sandbox:

- Expressions involving literals are allowed. For example:

  ```
  eval("null");
  ```

  ```
  eval("3 + .14");
  ```

  ```
  eval("'foo'");
  ```

- Object literals are allowed, as in the following:

  ```
  { prop1: val1, prop2: val2 }
  ```

- Object literal setter/getters are *prohibited*, as in the following:

  ```
  { get prop1() { ... }, set prop1(v) { ... } }
  ```

- Array literals are allowed, as in the following:

  ```
  [ val1, val2, val3 ]
  ```

- Expressions involving property reads are *prohibited*, as in the following:

  ```
  a.b.c
  ```

- Function invocation is *prohibited.*
- Function definitions are *prohibited.*
- Setting any property is *prohibited.*
- Function literals are *prohibited.*

However, while the code is loading, before the `onload` event, and during execution the `onload` event handler function, these restrictions do not apply to content in the application security sandbox.

For example, after code is loaded, the following code results in the runtime throwing an exception:

```
eval("alert(44)");
eval("myFunction(44)");
eval("NativeApplication.applicationID");
```

Dynamically generated code, such as that which is made when calling the `eval()` function, would pose a security risk if allowed within the application sandbox. For example, an application may inadvertently execute a string loaded from a network domain, and that string may contain malicious code. For example, this could be code to delete or alter files on the user's computer. Or it could be code that reports back the contents of a local file to an untrusted network domain.

Ways to generate dynamic code are the following:

- Calling the `eval()` function.
- Using `innerHTML` properties or DOM functions to insert script tags that load a script outside of the application directory.
- Using `innerHTML` properties or DOM functions to insert script tags that have inline code (rather than loading a script via the `src` attribute).
- Setting the `src` attribute for a `script` tags to load a JavaScript file that is outside of the application directory.
- Using the `javascript` URL scheme (as in `href="javascript:alert('Test')"`).

- Using the `setInterval()` or `setTimout()`function where the first parameter (defining the function to run asynchronously) is a string (to be evaluated) rather than a function name (as in `setTimeout('x = 4', 1000)`).
- Calling `document.write()` or `document.writeln()`.

Code in the application security sandbox can only use these methods while content is loading.

These restrictions do *not* prevent using `eval()` with JSON object literals. This lets your application content work with the JSON JavaScript library. However, you are restricted from using overloaded JSON code (with event handlers).

For other Ajax frameworks and JavaScript code libraries, check to see if the code in the framework or library works within these restrictions on dynamically generated code. If they do not, include any content that uses the framework or library in a non-application security sandbox. For details, see "Privileges of content in non-application sandboxes" on page 75 and "Scripting between application and non-application content" on page 84. Adobe maintains a list of Ajax frameworks known to support the application security sandbox, at http://www.adobe.com/go/airappsandbox-frameworks.

Unlike content in the application security sandbox, JavaScript content in a non-application security sandbox *can* call the `eval()` function to execute dynamically generated code at any time.

Restrictions on access to AIR APIs (for non-application sandboxes)

JavaScript code in a non-application sandbox does not have access to the `window.runtime` object, and as such this code cannot execute AIR APIs. If content in a non-application security sandbox calls the following code, the application throws a TypeError exception:

```
try {
    window.runtime.flash.system.NativeApplication.nativeApplication.exit();
}
catch (e)
{
    alert(e);
}
```

The exception type is TypeError (undefined value), because content in the non-application sandbox does not recognize the `window.runtime` object, so it is seen as an undefined value.

You can expose runtime functionality to content in a non-application sandbox by using a script bridge. For details, see and "Scripting between application and non-application content" on page 84.

Restrictions on using XMLHttpRequest calls

HTML content n the application security sandbox cannot use synchronous XMLHttpRequest methods to load data from outside of the application sandbox while the HTML content is loading and during `onLoad` event.

By default, HTML content in non-application security sandboxes are not allowed to use the JavaScript XMLHttpRequest object to load data from domains other than the domain calling the request. A `frame` or `iframe` tag can include an `allowcrosscomainxhr` attribute. Setting this attribute to any non-null value allows the content in the frame or iframe to use the Javascript XMLHttpRequest object to load data from domains other than the domain of the code calling the request:

```
<iframe id="UI"
    src="http://example.com/ui.html"
    sandboxRoot="http://example.com/"
    allowcrossDomainxhr="true"
    documentRoot="app:/">
</iframe>
```

For more information, see "Scripting between content in different domains" on page 80.

Restrictions on loading CSS, frame, iframe, and img elements (for content in non-application sandboxes)

HTML content in remote (network) security sandboxes can only load CSS, `frame`, `iframe`, and `img` content from remote sandboxes (from network URLs).

HTML content in local-with-filesystem, local-with-networking, or local-trusted sandboxes can only load CSS, frame, iframe, and `img` content from local sandboxes (not from application or remote sandboxes).

Restrictions on calling the JavaScript window.open() method

If a window that is created via a call to the JavaScript `window.open()` method displays content from a non-application security sandbox, the window's title begins with the title of the main (launching) window, followed by a colon character. You cannot use code to move that portion of the title of the window off screen.

Content in non-application security sandboxes can only successfully call the JavaScript `window.open()` method in response to an event triggered by a user mouse or keyboard interaction. This prevents non-application content from creating windows that might be used deceptively (for example, for phishing attacks). Also, the event handler for the mouse or keyboard event cannot set the `window.open()` method to execute after a delay (for example by calling the `setTimeout()` function).

Content in remote (network) sandboxes can only use the `window.open()` method to open content in remote network sandboxes. It cannot use the `window.open()` method to open content from the application or local sandboxes.

Content in the local-with-filesystem, local-with-networking, or local-trusted sandboxes (see "Sandboxes" on page 73) can only use the `window.open()` method to open content in local sandboxes. It cannot use `window.open()` to open content from the application or remote sandboxes.

Errors when calling restricted code

If you call code that is restricted from use in a sandbox due to these security restrictions, the runtime dispatches a JavaScript error: "Adobe AIR runtime security violation for JavaScript code in the application security sandbox."

For more information, see "Avoiding security-related JavaScript errors" on page 260.

Scripting between content in different domains

AIR applications are granted special privileges when they are installed. It is crucial that the same privileges not be leaked to other content, including remote files and local files that are not part of the application.

Contents

About the AIR sandbox bridge

Normally, content from other domains cannot call scripts in other domains. To protect AIR applications from accidental leakage of privileged information or control, the following restrictions are placed on content in the `application` security sandbox (content installed with the application):

- Code in the application security sandbox cannot allow cross-scripting to other sandboxes by calling the `Security.allowDomain()` method. Calling this method from the application security sandbox has no effect.

- Importing non-application content into the application sandbox by setting the `LoaderContext.securityDomain` or the `LoaderContext.applicationDomain` property is prevented.

There are still cases where the main AIR application requires content from a remote domain to have controlled access to scripts in the main AIR application, or vice versa. To accomplish this, the runtime provides a *sandbox bridge* mechanism, which serves as a gateway between the two sandboxes. A sandbox bridge can provide explicit interaction between remote and application security sandboxes.

The sandbox bridge exposes two objects that both loaded and loading scripts can access:

- The `parentSandboxBridge` object lets loading content expose properties and functions to scripts in the loaded content.
- The `childSandboxBridge` object lets loaded content expose properties and function to scripts in the loading content.

Objects exposed via the sandbox bridge are passed by value, not by reference. All data is serialized. This means that the objects exposed by one side of the bridge cannot be set by the other side, and that objects exposed are all untyped. Also, you can only expose simple objects and functions; you cannot expose complex objects.

If child content attempts to set a property of the `parentSandboxBridge` object, the runtime throws a SecurityError exception. Similarly, if parent content attempts to set a property of the `childSandboxBridge` object, the runtime throws a SecurityError exception.

Sandbox bridge example (SWF)

Suppose an AIR music store application wants to allow remote SWF files to broadcast the price of albums, but does not want the remote SWF file to disclose whether the price is a sale price. To do this, a StoreAPI class provides a method to acquire the price, but obscures the sale price. An instance of this StoreAPI class is then assigned to the `parentSandboxBridge` property of the LoaderInfo object of the Loader object that loads the remote SWF.

The following is the code for the AIR music store:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
title="Music Store" creationComplete="initApp()">
    <mx:Script>
        import flash.display.Loader;
        import flash.net.URLRequest;

        private var child:Loader;
        private var isSale:Boolean = false;

        private function initApp():void {
            var request:URLRequest =
                    new URLRequest("http://[www.yourdomain.com]/PriceQuoter.swf")

            child = new Loader();
            child.contentLoaderInfo.parentSandboxBridge = new StoreAPI(this);
            child.load(request);
            container.addChild(child);
        }
        public function getRegularAlbumPrice():String {
            return "$11.99";
        }
        public function getSaleAlbumPrice():String {
            return "$9.99";
        }
        public function getAlbumPrice():String {
            if(isSale) {
                return getSaleAlbumPrice();
            }
            else {
                return getRegularAlbumPrice();
            }
        }
    </mx:Script>
    <mx:UIComponent id="container" />
</mx:WindowedApplication>
```

The StoreAPI object calls the main application to retrieve the regular album price, but returns “Not available” when the `getSaleAlbumPrice()` method is called. The following code defines the StoreAPI class:

```
public class StoreAPI
{
    private static var musicStore:Object;

    public function StoreAPI(musicStore:Object)
    {
        this.musicStore = musicStore;
    }

    public function getRegularAlbumPrice():String {
        return musicStore.getRegularAlbumPrice();
    }

    public function getSaleAlbumPrice():String {
        return "Not available";
    }

    public function getAlbumPrice():String {
        return musicStore.getRegularAlbumPrice();
    }
}
```

The following code represents an example of a PriceQuoter SWF file that reports the store’s price, but cannot report the sale price:

```
package
{
    import flash.display.Sprite;
    import flash.system.Security;
    import flash.text.*;

    public class PriceQuoter extends Sprite
    {
        private var storeRequester:Object;

        public function PriceQuoter() {
            trace("Initializing child SWF");
            trace("Child sandbox: " + Security.sandboxType);
            storeRequester = loaderInfo.parentSandboxBridge;

            var tf:TextField = new TextField();
            tf.autoSize = TextFieldAutoSize.LEFT;
            addChild(tf);

            tf.appendText("Store price of album is: " + storeRequester.getAlbumPrice());
            tf.appendText("\n");
            tf.appendText("Sale price of album is: " + storeRequester.getSaleAlbumPrice());
        }
    }
}
```

Sandbox bridge example (HTML)

In HTML content, the `parentSandboxBridge` and `childSandboxBridge` properties are added to the JavaScript window object of a child document. For an example of how to set up bridge functions in HTML content, see "Setting up a sandbox bridge interface" on page 272.

Limiting API exposure

When exposing sandbox bridges, it's important to expose high-level APIs that limit the degree to which they can be abused. Keep in mind that the content calling your bridge implementation may be compromised (for example, via a code injection). So, for example, exposing a "`readFile(path:String)`" method (that reads the contents of an arbitrary file) via a bridge is vulnerable to abuse. It would be better to expose a "`readApplicationSetting()`" API that doesn't take a path and reads a specific file. The more semantic approach limits the damage that an application can do once part of it is compromised.

See also

- "Cross-scripting content in different security sandboxes" on page 270
- "The application sandbox" on page 74
- "Privileges of content in non-application sandboxes" on page 75

Writing to disk

Applications running in a web browser have only limited interaction with the user's local file system. Web browsers implement security policies that ensure that a user's computer cannot be compromised as a result of loading web content. For example, SWF files running through Flash Player in a browser cannot directly interact with files already on a user's computer. Shared objects and cookies can be written to a user's computer for the purpose of maintaining user preferences and other data, but this is the limit of file system interaction. Because AIR applications are natively installed, they have a different security contract, one which includes the capability to read and write across the local file system.

This freedom comes with high responsibility for developers. Accidental application insecurities jeopardize not only the functionality of the application, but also the integrity of the user's computer. For this reason, developers should read "Best security practices for developers" on page 85.

AIR developers can access and write files to the local file system using several URL scheme conventions:

URL scheme	Description
app:/	An alias to the application directory. Files accessed from this path are assigned the application sandbox and have the full privileges granted by the runtime.
app-storage:/	An alias to the local storage directory, standardized by the runtime. Files accessed from this path are assigned a non-application sandbox.
file:///	An alias that represents the root of the user's hard disk. A file accessed from this path is assigned an application sandbox if the file exists in the application directory, and a non-application sandbox otherwise.

***Note:** AIR applications cannot modify content using the app: URL scheme. Also, the application directory may be read only because of administrator settings.*

Unless there are administrator restrictions to the user's computer, AIR applications are privileged to write to any location on the user's hard drive. Developers are advised to use the `app-storage:/` path for local storage related to their application. Files written to `app-storage:/` from an application are put in a standard location:

- On Mac OS: the storage directory of an application is `<appData>/<appId>/Local Store/` where `<appData>` is the user's preferences folder. This is typically `/Users/<user>/Library/Preferences`
- On Windows: the storage directory of an application is `<appData>\<appId>\Local Store\` where `<appData>` is the user's CSIDL_APPDATA Special Folder. This is typically `C:\Documents and Settings\<userName>\Application Data`

If an application is designed to interact with existing files in the user's file system, be sure to read "Best security practices for developers" on page 85.

Working securely with untrusted content

Content not assigned to the application sandbox can provide additional scripting functionality to your application, but only if it meets the security criteria of the runtime. This topic explains the AIR security contract with non-application content.

Contents

Security.allowDomain()

AIR applications restrict scripting access for non-application content more stringently than the Flash Player 9 browser plug-in restricts scripting access for untrusted content. For example, in Flash Player in the browser, when a SWF file that is assigned to the `local-trusted` sandbox calls the `System.allowDomain()` method, scripting access is granted to any SWF loaded from the specified domain, reassigning this remote file from the `remote` sandbox to the `local-trusted` sandbox. The analogous approach is not permitted from `application` content in AIR applications, since it would grant unreasonable access unto the non-application file into the user's file system. Remote files cannot directly access the application sandbox, regardless of calls to the `Security.allowDomain()` method.

Scripting between application and non-application content

AIR applications that script between application and non-application content have more complex security arrangements. Files that are not in the application sandbox are only allowed to access the properties and methods of files in the application sandbox through the use of a sandbox bridge. A sandbox bridge acts as a gateway between application content and non-application content, providing explicit interaction between the two files. When used correctly, sandbox bridges provide an extra layer of security, restricting non-application content from accessing object references that are part of application content.

The benefit of sandbox bridges is best illustrated through example. Suppose an AIR music store application wants to provide an API to advertisers who want to create their own SWF files, with which the store application can then communicate. The store wants to provide advertisers with methods to look up artists and CDs from the store, but also wants to isolate some methods and properties from the third-party SWF file for security reasons.

A sandbox bridge can provide this functionality. By default, content loaded externally into an AIR application at runtime does not have access to any methods or properties in the main application. With a custom sandbox bridge implementation, a developer can provide services to the remote content without exposing these methods or properties. Consider the sandbox bridge as a pathway between trusted and untrusted content, providing communication between loader and loadee content without exposing object references.

For more information on how to securely use sandbox bridges, see "Scripting between content in different domains" on page 80.

Protection against dynamically generating unsafe SWF content

The `Loader.loadBytes()` method provides a way for an application to generate SWF content from a byte array. However, injection attacks on data loaded from remote sources could do severe damage when loading content. This is especially true when loading data into the application sandbox, where the generated SWF content can access the full set of AIR APIs.

There are legitimate uses for using the `loadBytes()` method without generating executable SWF code. You can use the `loadBytes()` method to generate an image data to control the timing of image display, for example. There are also legitimate uses that *do* rely on executing code, such as dynamic creation of SWF content for audio playback. In AIR, by default the `loadBytes()` method does *not* let you load SWF content; it only allows you to load image content. In AIR, the `loaderContext` property of the `loadBytes()` method has an `allowLoadBytesCodeExecution` property, which you can set to `true` to explicitly allow the application to use `loadBytes()` to load executable SWF content. The following code shows how to use this feature:

```
var loader:Loader = new Loader();
var loaderContext:LoaderContext = new LoaderContext();
loaderContext.allowLoadBytesCodeExecution = true;
loader.loadBytes(bytes, loaderContext);
```

If you call `loadBytes()` to load SWF content and the `allowLoadBytesCodeExecution` property of the LoaderContext object is set to `false` (the default), the Loader object throws a SecurityError exception.

***Note:** In a future release of Adobe AIR, this API may change. When that occurs, you may need to recompile content that uses the `allowLoadBytesCodeExecution` property of the LoaderContext class.*

Best security practices for developers

Although AIR applications are built using web technologies, it is important for developers to note that they are not working within the browser security sandbox. This means that it is possible to build AIR applications that can do harm to the local system, either intentionally or unintentionally. AIR attempts to minimize this risk, but there are still ways where vulnerabilities can be introduced. This topic covers important potential insecurities.

Contents

Risk from importing files into the application security sandbox

Files that exist in the application directory are assigned to the application sandbox and have the full privileges of the runtime. Applications that write to the local file system are advised to write to `app-storage:/`. This directory exists separately from the application files on the user's computer, hence the files are not assigned to the application sandbox and present a reduced security risk. Developers are advised to consider the following:

- Include a file in an AIR file (in the installed application) only if it is necessary.
- Include a scripting file in an AIR file (in the installed application) only if its behavior is fully understood and trusted.
- Do not write to or modify content in the application directory. The runtime prevents applications from writing or modifying files and directories using the `app:/` URL scheme by throwing a SecurityError exception.
- Do not use data from a network source as parameters to methods of the AIR API that may lead to code execution. This includes use of the `Loader.loadBytes()` method and the JavaScript `eval()` function.

Risk from using an external source to determine paths

An AIR application can be compromised when using external data or content. For this reason, take special care when using data from the network or file system. The onus of trust is ultimately up to the developer and the network connections they make, but loading foreign data is inherently risky, and should not be used for input into sensitive operations. Developers are advised against the following:

- Using data from a network source to determine a file name
- Using data from a network source to construct a URL that the application uses to send private information

Risk from using, storing, or transmitting insecure credentials

Storing user credentials on the user's local file system inherently introduces the risk that these credentials may be compromised. Developers are advised to consider the following:

- If credentials must be stored locally, to encrypt the credentials when writing to the local file system. The runtime provides an encrypted storage unique to each installed application, via the EncryptedLocalStore class. For details, see “Storing encrypted data” on page 240.
- Do not transmit unencrypted user credentials to a network source unless that source is trusted.
- Never specify a default password in credential creation — let users create their own. Users who leave the default expose their credentials to an attacker that already knows the default password

Risk from a downgrade attack

During application install, the runtime checks to ensure that a version of the application is not currently installed. If an application is already installed, the runtime compares the version string against the version that is being installed. If this string is different, the user can choose to upgrade their installation. The runtime does not guarantee that the newly installed version is newer than the older version, only that it is different. An attacker can distribute an older version to the user to circumvent a security weakness. For this reason, the developer is advised to make version checks when the application is run. It is a good idea to have applications check the network for required updates. That way, even if an attacker gets the user to run an old version, that old version will recognize that it needs to be updated. Also, using a clear versioning scheme for your application makes it more difficult to trick users into installing a downgraded version. For details on providing application versions, see “Defining properties in the application descriptor file” on page 89.

Code signing

All AIR installer files are required to be code signed. Code signing is a cryptographic process of confirming that the specified origin of software is accurate. AIR applications can be signed either by linking a certificate from an external certificate authority (CA) or by constructing your own certificate. A commercial certificate from a well-known CA is strongly recommended and provides assurance to your users that they are installing your application, not a forgery. However, self-signed certificates can be created using `adt` from the SDK or using either Flash, Flex Builder, or another application that uses `adt` for certificate generation. Self-signed certificates do not provide any assurance that the application being installed is genuine.

For more information about digitally signing AIR applications, see "Digitally signing an AIR file" on page 339 and "Creating an AIR application using the command line tools" on page 23.

Chapter 11: Setting AIR application properties

Aside from all the files and other assets that make up an AIR application, each AIR application requires an application descriptor file—an XML file which defines the basic properties of the application.

When you use Adobe® Flex™ Builder™ 3, the application descriptor file is automatically generated when you create an Adobe® AIR™ project. If you're developing AIR applications using the Adobe® Flex™ 3 or AIR™ SDKs, you must create this file manually. A sample descriptor file, `descriptor-sample.xml`, can be found in the `samples` directory of your SDK installation.

Contents

The application descriptor file structure

The application descriptor file contains the properties that affect the entire application, such as its name, version, copyright, and so on. Any file name can be used for the application descriptor file. When you package the application with either Flex Builder or ADT, the application descriptor file is renamed `application.xml` and placed within a special directory inside the package.

Here's an example application descriptor file:

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/1.0"
    minimumPatchLevel="1047">
    <id>com.example.HelloWorld</id>
    <version>2.0</version>
    <filename>Hello World</filename>
    <name>Example Co. AIR Hello World</name>
    <description>
        The Hello World sample file from the Adobe AIR documentation.
    </description>
    <copyright>Copyright (c) 2006 Example Co.</copyright>
    <initialWindow>
        <title>Hello World</title>
        <content>
            HelloWorld-debug.swf
        </content>
        <systemChrome>none</systemChrome>
        <transparent>true</transparent>
        <visible>true</visible>
        <minimizable>true</minimizable>
        <maximizable>false</maximizable>
        <resizable>false</resizable>
        <width>640</width>
        <height>480</height>
        <minSize>320 240</minSize>
        <maxSize>1280 960</maxSize>
    </initialWindow>
    <installFolder>Example Co/Hello World</installFolder>
    <programMenuFolder>Example Co</programMenuFolder>
    <icon>
        <image16x16>icons/smallIcon.png</image16x16>
        <image32x32>icons/mediumIcon.png</image32x32>
        <image48x48>icons/bigIcon.png</image48x48>
        <image128x128>icons/biggestIcon.png</image128x128>
    </icon>
    <customUpdateUI>true</customUpdateUI>
    <allowBrowserInvocation>false</allowBrowserInvocation>
    <fileTypes>
        <fileType>
            <name>adobe.VideoFile</name>
            <extension>avf</extension>
            <description>Adobe Video File</description>
            <contentType>application/vnd.adobe.video-file</contentType>
            <icon>
                <image16x16>icons/avfIcon_16.png</image16x16>
                <image32x32>icons/avfIcon_32.png</image32x32>
                <image48x48>icons/avfIcon_48.png</image48x48>
                <image128x128>icons/avfIcon_128.png</image128x128>
            </icon>
        </fileType>
    </fileTypes>
</application>
```

Defining properties in the application descriptor file

At its root, the application descriptor file contains an `application` property that has several attributes:

```
<application version="1.0"
    xmlns="http://ns.adobe.com/air/application/1.0"
    minimumPatchLevel="5331">
```

`xmlns` The AIR namespace, which you must define as the default XML namespace. The namespace changes with each major release of AIR (but not with minor patches). The last segment of the namespace, such as “1.0” indicates the runtime version required by the application.

`minimumPatchLevel` Together with the AIR namespace, this property determines the version of the runtime required by the application. The application installer prompts the user to download and install the required version or patch, if necessary.

Defining the basic application information

The following elements define application ID, version, name, file name, description, and copyright information:

```
<id>com.example.samples.TestApp</id>
<version>2.0</version>
<filename>TestApp</filename>
<name>Example Co. Test Application</name>
<description>An MP3 player.</description>
<copyright>Copyright (c) 2006 [YourCompany, Inc.]</copyright>
```

`id` An identifier string unique to the application, known as the application ID. The attribute value is restricted to the following characters:

- 0–9
- a–z

- A–Z
- . (dot)
- - (hyphen)

The value must contain 1 to 212 characters. This element is required.

The `id` string typically uses a dot-separated hierarchy, in alignment with a reversed DNS domain address, a Java package or class name, or an OS X Universal Type Identifier. The DNS-like form is not enforced, and AIR does not create any association between the name and actual DNS domains.

`version` Specifies the version information for the application. (It has no relation to the version of the runtime). The version string is an application-defined designator. AIR does not interpret the version string in any way. Thus, version “3.0” is not assumed to be more current than version “2.0.” Examples: `"1.0"`, `".4"`, `"0.5"`, `"4.9"`, `"1.3.4a"`. This element is required.

`filename` The string to use as a filename of the application (without extension) when the application is installed. The application file launches the AIR application in the runtime. If no `name` value is provided, the `filename` is also used as the name of the installation folder. This element is required.

The `filename` property can contain any Unicode (UTF-8) character except the following, which are prohibited from use as filenames on various file systems:

Character	Hex Code
various	0x00 – x1F
*	x2A
"	x22
:	x3A
>	x3C
<	x3E
?	x3F
\	x5C
|	x7C

The `filename` value cannot end in a period.

`name` (Optional, but recommended) The title displayed by the AIR application installer. If provided, the `name` value is also used as the name of the installation folder (within the folder specified in the `installFolder` element).

`description` (Optional) Displayed in the AIR application installer.

`copyright` (Optional) The copyright information for the AIR application. On Mac OS, the copyright text appears in the About dialog box for the installed application, and it is used in the NSHumanReadableCopyright field in the Info.plist file for the application.

Defining the installation folder and program menu folder

The installation and program menu folders are defined with the following property settings:

```
<installFolder>Acme</installFolder>
<programMenuFolder>Acme/Applications</programMenuFolder>
```

`installFolder` (Optional) Identifies the subdirectory of the default installation directory.

On Windows, the default installation subdirectory is the Program Files directory. On Mac OS, it is the /Applications directory. For example, if the `installFolder` property is set to `"Acme"` and an application is named `"ExampleApp"`, then the application is installed in C:\Program Files\Acme\ExampleApp on Windows and in /Applications/Acme/Example.app on MacOS.

Use the forward-slash (/) character as the directory separator character if you want to specify a nested subdirectory, as in the following:

```
<installFolder>Acme/Power Tools</installFolder>
```

The `installFolder` property can contain any Unicode (UTF-8) character except those that are prohibited from use as folder names on various file systems (see the `filename` property above for the list of exceptions).

The `installFolder` property is optional. If you specify no `installFolder` property, the application is installed in a subdirectory of the default installation directory, based on the `name` property.

`programMenuFolder` (Optional) Identifies the location in which to place shortcuts to the application in the All Programs menu of the Windows operating system. (This setting is currently ignored on other operating systems.) The restrictions on the characters that are allowed in the value of the property are the same as those for the `installFolder` property.

Defining the properties of the initial application window

When an AIR application is loaded, the runtime uses the values in the `initialWindow` element to create the initial window for the application. The runtime then loads the SWF or HTML file specified in the `content` element into the window.

```
<initialWindow>
    <content>AIRTunes.swf</content>
    <title>AIR Tunes</title>
    <systemChrome>none</systemChrome>
    <transparent>true</transparent>
    <visible>true</visible>
    <minimizable>true</minimizable>
    <maximizable>true</maximizable>
    <resizable>true</resizable>
    <width>400</width>
    <height>600</height>
    <x>150</x>
    <y>150</y>
    <minSize>300 300</minSize>
    <maxSize>800 800</maxSize>
</initialWindow>
```

The child elements of the `initialWindow` element set the properties of the window into which the root content file is loaded.

`content` The value specified for the `content` element is the URL for the main content file of the application. This may be either a SWF file or an HTML file. The URL is specified relative to the root of the application installation folder. (When running an AIR application with ADL, the URL is relative to the folder containing the application descriptor file. You can use the `root-dir` parameter of ADL to specify a different root directory.)

Note: *Because the value of the content element is treated as a URL, characters in the name of the content file must be URL encoded according to the rules defined in RFC 1738. Space characters, for example, must be encoded as* `%20`.

`title` (Optional) The window title.

`systemChrome` (Optional) If you set this attribute to `standard`, the standard system chrome supplied by the operating system is displayed. If you set it to `none`, no system chrome is displayed. When using the Flex mx:WindowedApplication component, the component applies its custom chrome if standard system chrome is not set. The system chrome setting cannot be changed at run time.

`transparent` (Optional) Set to `"true"` if you want the application window to support alpha blending. A window with transparency may draw more slowly and require more memory. The transparent setting cannot be changed at run time.

Important: *You can only set `transparent` to `true` when `systemChrome` is `none`.*

`visible` (Optional) Set to `true` if you want the main window to be visible as soon as it is created. The default value is `false`. The Flex mx:WindowedApplication component automatically sets the window to visible immediately before the `applicationComplete` event is dispatched, unless the `showWindow` attribute is set to `false` in the MXML definition.

You may want to leave the main window hidden initially, so that changes to the window's position, the window's size, and the layout of its contents are not shown. You can then display the window by setting the `stage.nativeWindow.visible` property (for the main window) to `true`. For details, see "Working with native windows" on page 102.

`x, y, width, height` (Optional) The initial bounds of the main window of the application. If you do not set these values, the window size is determined by the settings in the root SWF file or, in the case of HTML, by the operating system.

`minSize, maxSize` (Optional) The minimum and maximum sizes of the window. If you do not set these values, they are determined by the operating system.

`minimizable, maximizable, resizable` (Optional) Specifies whether the window can be minimized, maximized, and resized. By default, these settings default to `true`.

Note: *On operating systems, such as Mac OS X, for which maximizing windows is a resizing operation, both maximizable and resizable must be set to `false` to prevent the window from being zoomed or resized.*

Specifying icon files

The `icon` property specifies one or more icon files to be used for the application. Including an icon is optional. If you do not specify an `icon` property, the operating system displays a default icon.

The path specified is relative to the application root directory. Icon files must be in the PNG format. You can specify all of the following icon sizes:

```
<icon>
    <image16x16>icons/smallIcon.png</image16x16>
    <image32x32>icons/mediumIcon.png</image32x32>
    <image48x48>icons/bigIcon.png</image48x48>
    <image128x128>icons/biggestIcon.png</image128x128>
</icon>
```

If an element for a given size is present, the image in the file must be exactly the size specified. If all sizes are not provided, the closest size is scaled to fit for a given use of the icon by the operating system.

Note: *The icons specified are not automatically added to the AIR package. The icon files must be included in their correct relative locations when the application is packaged.*

For best results, provide an image for each of the available sizes. In addition, make sure that the icons look presentable in both 16- and 32-bit color modes.

Providing a custom user interface for application updates

AIR installs and updates applications using the default installation dialogs. However, you can provide your own user interface for updating an application. To indicate that your application should handle the update process itself, set the `customUpdateUI` element to `true`:

```
<customUpdateUI>true</customUpdateUI>
```

When the installed version of your application has the `customUpdateUI` element set to `true` and the user then double-clicks the AIR file for a new version or installs an update of the application using the seamless install feature, the runtime opens the installed version of the application, rather than the default AIR application installer. Your application logic can then determine how to proceed with the update operation. (The application ID and publisher ID in the AIR file must match those in the installed application for an upgrade to proceed.)

Note: *The `customUpdateUI` mechanism only comes into play when the application is already installed and the user double-clicks the AIR installation file containing an update or installs an update of the application using the seamless install feature. You can download and start an update through your own application logic, displaying your custom UI as necessary, whether or not `customUpdateUI` is `true`.*

For more information, see "Updating AIR applications" on page 344.

Allowing browser invocation of the application

If you specify the following setting, the installed AIR application can be launched via the browser invocation feature (by the user clicking a link in a page in a web browser):

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

The default value is `false`.

If you set this value to `true`, be sure to consider security implications, described in "Browser invocation" on page 311.

For more information, see "Installing and running AIR applications from a web page" on page 332.

Declaring file type associations

The `fileTypes` element allows you to declare the file types with which an AIR application can be associated. When an AIR application is installed, any declared file type is registered with the operating system and, if these file types are not already associated with another application, they are associated with the AIR application. To override an existing association between a file type and another application, use the `NativeApplication.setAsDefaultApplication()` method at run time (preferably with the user's permission).

Note: *The runtime methods can only manage associations for the file types declared in the application descriptor.*

```
<fileTypes>
    <fileType>
        <name>adobe.VideoFile</name>
        <extension>avf</extension>
        <description>Adobe Video File</description>
        <contentType>application/vnd.adobe.video-file</contentType>
        <icon>
            <image16x16>icons/AIRApp_16.png</image16x16>
            <image32x32>icons/AIRApp_32.png</image32x32>
            <image48x48>icons/AIRApp_48.png</image48x48>
            <image128x128>icons/AIRApp_128.png</image128x128>
        </icon>
    </fileType>
</fileTypes>
```

The `fileTypes` element is optional. If present, it may contain any number of `fileType` elements.

The `name` and `extension` elements are required for each `fileType` declaration that you include. The same name can be used for multiple extensions. The extension uniquely identifies the file type. (Note that the extension is specified without the preceding period.) The `description` element is optional and is displayed to the user by the operating system user interface. The `contentType` property is also optional, but helps the operating system to locate the best application to open a file under some circumstances. The value should be the MIME type of the file content.

Icons can be specified for the file extension, using the same format as the application icon element. The icon files must also be included in the AIR installation file (they are not packaged automatically).

When a file type is associated with an AIR application, the application will be invoked whenever a user opens a file of that type. If the application is already running, AIR will dispatch the InvokeEvent object to the running instance. Otherwise, AIR will launch the application first. In both cases, the path to the file can be retrieved from the InvokeEvent object dispatched by the NativeApplication object. You can use this path to open the file.

See also

- "Managing file associations" on page 317
- "Capturing command line arguments" on page 309

Chapter 12: New functionality in Adobe AIR

This topic provides an overview of the new functionality in Adobe® AIR™ that is not available to SWF content running in Adobe® Flash® Player.

- New runtime classes
- Runtime classes with new functionality
- New Flex components
- Service monitoring framework classes

New runtime classes

The following runtime classes are new in Adobe AIR. They are not available to SWF content running in the browser:

Class	Package
BrowserInvokeEvent	flash.events
Clipboard	flash.desktop
ClipboardFormats	flash.desktop
ClipboardTransferMode	flash.desktop
CompressionAlgorithm	flash.utils
DockIcon	flash.desktop
DRMAuthenticateEvent	flash.events
DRMErrorEvent	flash.events
DRMStatusEvent	flash.events
EncryptedLocalStore	flash.data
File	flash.filesystem
FileListEvent	flash.events
FileMode	flash.filesystem
FileStream	flash.filesystem
FocusDirection	flash.display
HTMLHistoryItem	flash.html
HTMLHost	flash.html
HTMLLoader	flash.html
HTMLPDFCapability	flash.html
HTMLUncaughtScriptExceptionEvent	flash.events
HTMLWindowCreateOptions	flash.html
Icon	flash.desktop
InteractiveIcon	flash.desktop
InvokeEvent	flash.events
NativeApplication	flash.desktop
NativeDragActions	flash.desktop
NativeDragEvent	flash.events
NativeDragManager	flash.desktop
NativeDragOptions	flash.desktop
NativeMenu	flash.display
NativeMenuItem	flash.display

Class	**Package**
NativeWindow	flash.display
NativeWindowBoundsEvent	flash.events
NativeWindowDisplayState	flash.display
NativeWindowDisplayStateEvent	flash.events
NativeWindowInitOptions	flash.display
NativeWindowResize	flash.display
NativeWindowSystemChrome	flash.display
NativeWindowType	flash.display
NotificationType	flash.desktop
OutputProgressEvent	flash.events
RevocationCheckSettings	flash.security
Screen	flash.display
ScreenMouseEvent	flash.events
SignatureStatus	flash.security
SignerTrustSettings	flash.security
SQLCollationType	flash.data
SQLColumnNameStyle	flash.data
SQLColumnSchema	flash.data
SQLConnection	flash.data
SQLError	flash.errors
SQLErrorEvent	flash.events
SQLErrorOperation	flash.errors
SQLEvent	flash.events
SQLIndexSchema	flash.data
SQLResult	flash.data
SQLSchema	flash.data
SQLSchemaResult	flash.data
SQLStatement	flash.data
SQLTableSchema	flash.data
SQLTransactionLockType	flash.data
SQLTriggerSchema	flash.data
SQLUpdateEvent	flash.events
SQLViewSchema	flash.data

Class	**Package**
SystemTrayIcon	flash.desktop
Updater	flash.desktop
URLRequestDefaults	flash.net
XMLSignatureValidator	flash.utils

Also, the flash.security package includes the IURIDereferencer interface.

Runtime classes with new functionality

The following classes are available to SWF content running in the browser, but AIR provides additional properties or methods:

Class	Property or Method
Event	DISPLAYING EXITING HTML_BOUNDS_CHANGE HTML_DOM_INITIALIZE HTML_RENDER LOCATION_CHANGE NETWORK_CHANGE USER_IDLE USER_PRESENT
FileReference	uploadUnencoded()
HTTPStatusEvent	HTTP_RESPONSE_STATUS responseURL responseHeaders
KeyboardEvent	commandKey controlKey
LoaderContext	allowLoadBytesCodeExecution
LoaderInfo	parentSandboxBridge childSandboxBridge

NetStream	resetDRMVouchers()
	setDRMAuthenticationCredentials()

URLRequest	followRedirects manageCookies shouldAuthenticate shouldCacheResponse userAgent userCache setLoginCredentials()
URLStream	httpResponseStatus event
Stage	nativeWindow
Security	APPLICATION

Most of these new properties and methods are available only to content in the AIR application security sandbox. However, the new members in the URLRequest classes are also available to content running in other sandboxes.

The `ByteArray.compress()` and `ByteArray.uncompress()` methods each include a new `algorithm` parameter, allowing you to choose between deflate and zlib compression.

New Flex components

The following Adobe® Flex™ components are available when developing content for Adobe AIR:

- FileEvent
- FileSystemComboBox
- FileSystemDataGrid
- FileSystemEnumerationMode
- FileSystemHistoryButton
- FileSystemList
- FileSystemSizeDisplayMode
- FileSystemTree
- HTML
- Window
- WindowedApplication

For more information about the AIR Flex components, see "Using the Flex AIR components" on page 38.

Service monitoring framework classes

The air.net package contains classes for network detection. This package is only available to content running in Adobe AIR. It is included in the ServiceMonitor.swc file.

The package includes the following classes:

- ServiceMonitor
- SocketMonitor
- URLMonitor

Part 5: Windows, menus, and taskbars

Chapter 13: Working with native windows

You use the classes provided by the Adobe® AIR® native window API to create and manage desktop windows.

Contents

Quick Starts (Adobe AIR Developer Center)

- Interacting with a window
- Creating a transparent window application
- Customizing the look and feel of a native window
- Launching windows
- Creating toast-style windows
- Controlling the display order of windows
- Creating resizable, non-rectangular windows

Language Reference

- NativeWindow
- NativeWindowInitOptions

More Information

- Adobe AIR Developer Center for Flex (search for ‘AIR windows’)

AIR window basics

AIR provides an easy-to-use, cross-platform window API for creating native operating system windows using Flash®, Flex™, and HTML programming techniques.

With AIR, you have a wide latitude in developing the look and feel of your application. The windows you create can look like a standard desktop application, matching Apple style when run on the Mac, and conforming to Microsoft conventions when run on Windows. Or you can use the skinnable, extensible chrome provided by the Flex framework to establish your own style no matter where your application is run. You can even draw your own windows with vector and bitmap artwork with full support for transparency and alpha blending against the desktop. Tired of rectangular windows? Draw a round one.

Contents

- “Properties controlling native window style and behavior” on page 105
- “A visual window catalog” on page 107

Windows in AIR

AIR supports three distinct APIs for working with windows: the ActionScript-oriented NativeWindow class, the Flex framework mx:WindowedApplication and mx:Window classes, which “wrap” the NativeWindow class, and, in the HTML environment, the JavaScript Window class.

ActionScript windows

When you create windows with the NativeWindow class, you use the Flash Player stage and display list directly. To add a visual object to a NativeWindow, you add the object to the display list of the window stage or to another display object on the stage.

Flex Framework windows

When you create windows with the Flex framework, you typically use MXML components to populate the window. To add a Flex component to a window, you add the component element to the window MXML definition. You can also use ActionScript to add content dynamically. The mx:WindowedApplication and mx:Window components are designed as Flex containers and so can accept Flex components directly, whereas NativeWindow objects cannot. When necessary, the NativeWindow properties and methods can be accessed through the WindowedApplication and Window objects using the `nativeWindow` property. See “About window containers” on page 63 for more information about these Flex components.

HTML windows

When you create HTML windows, you use HTML, CSS, and JavaScript to display content. To add a visual object to an HTML window, you add that content to the HTML DOM. HTML windows are a special category of NativeWindow. The AIR host defines a `nativeWindow` property in HTML windows that provides access to the underlying NativeWindow instance. You can use this property to access the NativeWindow properties, methods, and events described here.

***Note:** The JavaScript Window object also has methods for scripting the containing window, such as `moveTo()` and `close()`. Where overlapping methods are available, you can use the method that is most convenient.*

The initial application window

The first window of your application is automatically created for you by AIR. AIR sets the properties and content of the window using the parameters specified in the `initialWindow` element of the application descriptor file.

If the root content is a SWF file, AIR creates a NativeWindow instance, loads the SWF file, and adds it to the window stage. If the root content is an HTML file, AIR creates an HTML window and loads the HTML.

For more information about the window properties specified in the application descriptor, see “The application descriptor file structure” on page 88.

Native window classes

The native window API contains the following classes:

Package	Classes
flash.display	• NativeWindow • NativeWindowInitOptions Window string constants are defined in the following classes: • NativeWindowDisplayState • NativeWindowResize • NativeWindowSystemChrome • NativeWindowType
flash.events	• NativeWindowBoundsEvent • NativeWindowDisplayStateEvent

Native window event flow

Native windows dispatch events to notify interested components that an important change is about to occur or has already occurred. Many window-related events are dispatched in pairs. The first event warns that a change is about to happen. The second event announces that the change has been made. You can cancel a warning event, but not a notification event. The following sequence illustrates the flow of events that occurs when a user clicks the maximize button of a window:

1. The NativeWindow object dispatches a `displayStateChanging` event.
2. If no registered listeners cancel the event, the window maximizes.
3. The NativeWindow object dispatches a `displayStateChange` event.

   In addition, the NativeWindow object also dispatches events for related changes to the window size and position. The window does not dispatch warning events for these related changes. The related events are:

   a. A `move` event is dispatched if the top, left corner of the window moved because of the maximize operation.

   b. A `resize` event is dispatched if the window size changed because of the maximize operation.

A NativeWindow object dispatches a similar sequence of events when minimizing, restoring, closing, moving, and resizing a window.

The warning events are only dispatched when a change is initiated through window chrome or other operating-system controlled mechanism. When you call a window method to change the window size, position, or display state, the window only dispatches an event to announce the change. You can dispatch a warning event, if desired, using the window `dispatchEvent()` method, then check to see if your warning event has been canceled before proceeding with the change.

For detailed information about the window API classes, methods, properties, and events, see the *Flex 3 Language Reference (http://www.adobe.com/go/learn_flex3_aslr).*

For general information about using the Flash display list, see the “Display Programming” section of the *Programming ActionScript 3.0 (http://www.adobe.com/go/programmingAS3)* reference.

Properties controlling native window style and behavior

The following properties control the basic appearance and behavior of a window:

- `type`
- `systemChrome`
- `transparent`

When you create a window, you set these properties on the NativeWindowInitOptions object passed to the window constructor. AIR reads the properties for the initial application window from the application descriptor. (Except the `type` property, which cannot be set in the application descriptor and is always set to `normal`.) The properties cannot be changed after window creation.

Some settings of these properties are mutually incompatible: `systemChrome` cannot be set to `standard` when either `transparent` is `true` or `type` is `lightweight`.

Window types

The AIR window types combine chrome and visibility attributes of the native operating system to create three functional types of window. Use the constants defined in the NativeWindowType class to reference the type names in code. AIR provides the following window types:

Type	Description
Normal	A typical window. Normal windows use the full-size style of chrome and appear on the Windows task bar and the Mac OS X window menu.
Utility	A tool palette. Utility windows use a slimmer version of the system chrome and do not appear on the Windows task bar and the Mac OS-X window menu.
Lightweight	Lightweight windows have no chrome and do not appear on the Windows task bar or the Mac OS X window menu. In addition, lightweight windows do not have the System (Alt+Space) menu on Windows. Lightweight windows are suitable for notification bubbles and controls such as combo-boxes that open a short-lived display area. When the lightweight `type` is used, `systemChrome` must be set to `none`.

Window chrome

Window chrome is the set of controls that allow users to manipulate a window in the desktop environment. Chrome elements include the title bar, title bar buttons, border, and resize grippers.

System chrome

You can set the `systemChrome` property to `standard` or `none`. Choose `standard` system chrome to give your window the set of standard controls created and styled by the user's operating system. Choose `none` to provide your own chrome for the window (or to use Flex chrome). Use the constants defined in the NativeWindowSystemChrome class to reference the system chrome settings in code.

System chrome is managed by the system. Your application has no direct access to the controls themselves, but can react to the events dispatched when the controls are used. When you use standard chrome for a window, the `transparent` property must be set to `false` and the `type` property must be `normal` or `utility`.

Flex chrome

When you use the Flex mx:WindowedApplication or mx:Window components, the window can be use either system chrome or chrome provided by the Flex framework. To use the Flex chrome, set the `systemChrome` property used to create the window to `none`.

Custom chrome

When you create a window with no system chrome and you do not use the Flex mx:WindowedApplication of mx:Window components, then you must add your own chrome controls to handle the interactions between a user and the window. You are also free to make transparent, non-rectangular windows.

Window transparency

To allow alpha blending of a window with the desktop or other windows, set the window `transparent` property to `true`. The `transparent` property must be set before the window is created and cannot be changed.

A transparent window has no default background. Any window area not occupied by a display object is invisible. If a display object has an alpha setting of less than one, then anything below the object shows through, including other display objects in the same window, other windows, and the desktop. Rendering large alpha-blended areas can be slow, so the effect should be used conservatively.

Transparent windows are useful when you want to create applications with borders that are irregular in shape or that "fade out" or appear to be invisible.

Transparency cannot be used with windows that have system chrome.

Transparency in an MXML application window

By default, the background of an MXML window is opaque, even if you create the window as *transparent*. (Notice the transparency effect at the corners of the window.) To present a transparent background for the window, set a background color and alpha value in the style sheet or <mx:Style> element contained in your application MXML file. For example, the following style declaration gives the background a slightly transparent green shade:

```
WindowedApplication
{
    background-alpha:".8";
    background-color:"0x448234";
}
```

Transparency in an HTML application window

By default the background of HTML content displayed in HTML windows and HTMLLoader objects is opaque, event if the containing window is transparent. To turn off the default background displayed for HTML content, set the `paintsDefaultBackground` property to `false`. The following example creates an HTMLLoader and turns off the default background:

```
var html:HTMLLoader = new HTMLLoader();
html.paintsDefaultBackground = false;
```

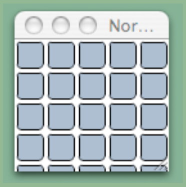
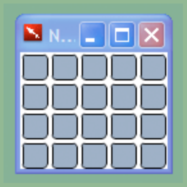
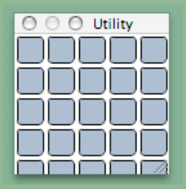
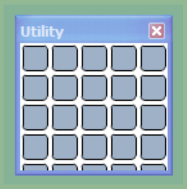
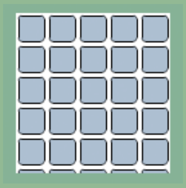
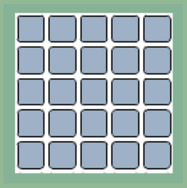
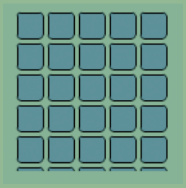
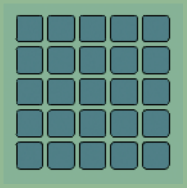
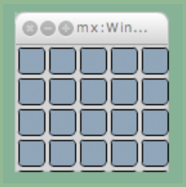
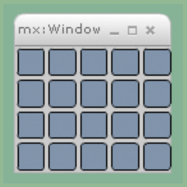
This example uses JavaScript to turn off the default background of an HTML window:

```
window.htmlLoader.paintsDefaultBackground = false;
```

If an element in the HTML document sets a background color, the background of that element is not transparent. Setting a partial transparency (or opacity) value is not supported. However, you can use a transparent PNG-format graphic as the background for a page or a page element to achieve a similar visual effect.

A visual window catalog

The following table illustrates the visual effects of different combinations of window property settings on the Mac OS X and Windows operating systems:

Window settings	Mac OS X	Microsoft Windows
Type: normal SystemChrome: standard Transparent: false		
Type: utility SystemChrome: standard Transparent: false		
Type: Any SystemChrome: none Transparent: false		
Type: Any SystemChrome: none Transparent: true		
mxWindowedApplication or mx:Window Type: Any SystemChrome: none Transparent: true		

***Note:** The following system chrome elements are not supported by AIR: the OS X Toolbar, the OS X Proxy Icon, Windows title bar icons, and alternate system chrome.*

Creating windows

AIR automatically creates the first window for an application, but you can create any additional windows you need. To create a native window, use the NativeWindow constructor method. To create a Flex window, use the mx:Window class. To create an HTML window, either use the HTMLLoader `createRootWindow()` method or, from an HTML document, call the JavaScript `window.open()` method.

Contents

Specifying window initialization properties

The initialization properties of a window cannot be changed after the desktop window is created. These immutable properties and their default values include:

Property	Default value
systemChrome	standard
type	normal
transparent	false
maximizable	true
minimizable	true
resizable	true

Set the properties for the initial window created by AIR in the application descriptor file. The main window of an AIR application is always type, *normal*. (Additional window properties can be specified in the descriptor file, such as `visible`, `width`, and `height`, but these properties can be changed at any time.)

When you create a window with the Flex mx:Window class, specify the initialization properties on the window object itself, either in the MXML declaration for the window, or in the code that creates the window. The desktop window is not created until you call the window open() method. Once a window is opened, these initialization properties cannot be changed.

Set the properties for other native and HTML windows created by your application using the NativeWindowInitOptions class. When you create a window, you must pass a NativeWindowInitOptions object specifying the window properties to either the NativeWindow constructor function or the HTMLLoader `createRootWindow()` method.

The following code creates a NativeWindowInitOptions object for a utility window:

```
var options:NativeWindowInitOptions = new NativeWindowInitOptions();
options.systemChrome = NativeWindowSystemChrome.STANDARD;
options.type = NativeWindowType.UTILITY
options.transparent = false;
options.resizable = false;
options.maximizable = false;
```

Setting `systemChrome` to `standard` *when* `transparent` is `true` *or* `type` *is* `lightweight` *is not supported.*

Note: *You cannot set the initialization properties for a window created with the JavaScript* `window.open()` *function. You can, however, override how these windows are created by implementing your own HTMLHost class.*

Creating the initial application window

AIR creates the initial application window based on the properties specified in the application descriptor and loads the file referenced in the content element. The content must be a SWF or an HTML file.

The initial window can be the main window of your application or it can merely serve to launch one or more other windows. You do not have to make it visible at all.

Creating the initial window with Flex

When creating an AIR application with the Flex framework, use the mx:WindowedApplication as the root element of your main MXML file. (You can use the mx:Application component, but it does not support all the window features available in AIR.) The WindowedApplication component serves as the initial entry point for the application.

When you launch the application, AIR creates a native window, initializes the Flex framework, and adds the WindowedApplication object to the window stage. When the launch sequence finishes, the WindowedApplication dispatches an `applicationComplete` event. Access the desktop window object with the `nativeWindow` property of the WindowedApplication instance.

The following example creates a simple WindowedApplication component that sets its x and y coordinates:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    applicationComplete="placeWindow()">
    <mx:Script>
        <![CDATA[
            private function placeWindow():void{
                this.nativeWindow.x = 300;
                this.nativeWindow.y = 300;
            }
        ]]>
    </mx:Script>
    <mx:Label text="Hello World" horizontalCenter="0" verticalCenter="0"/>
</mx:WindowedApplication>
```

Creating the initial window with ActionScript

When you create an AIR application using the Flex 3 SDK and ActionScript, the main class of your application must extend the Sprite class (or a subclass of the Sprite class). This class serves as the main entry point for the application.

When your application launches, AIR creates a window, creates an instance of the main class, and adds the instance to the window stage. To access the window, you can listen for the `addedToStage` event and then use the `nativeWindow` property of the Stage object to get a reference to the NativeWindow object.

The following example illustrates the basic skeleton for the main class of an AIR application built with ActionScript:

```
package {
    import flash.display.NativeWindow;
    import flash.display.Sprite;
    import flash.events.Event;

    public class Main extends Sprite
    {
        private var mainWindow:NativeWindow;
        public function Main(){
            this.addEventListener(Event.ADDED_TO_STAGE, initialize);
        }

        private function initialize(event:Event):void{
            mainWindow = this.stage.nativeWindow;
            //perform initialization...
            mainWindow.activate(); //show the window
        }
    }
}
```

***Note:** You can give the class any valid name.*

Creating an mx:Window

To create an mx:Window, you can create an MXML file using mx:Window as the root tag, or you can call the Window class constructor directly.

The following example creates and shows an mx:Window by calling the Window constructor:

```
var newWindow:Window = new Window();
newWindow.systemChrome = NativeWindowSystemChrome.NONE;
newWindow.transparent = true;
newWindow.title = "New Window";
newWindow.width = 200;
newWindow.height = 200;
newWindow.open(true);
```

Creating a NativeWindow

To create a NativeWindow, pass a NativeWindowInitOptions object to the NativeWindow constructor:

```
var options:NativeWindowInitOptions = new NativeWindowInitOptions();
options.systemChrome = NativeWindowSystemChrome.STANDARD;
options.transparent = false;
var newWindow:NativeWindow = new NativeWindow(options);
```

The window is not shown until you set the `visible` property to `true` or call the `activate()` method.

Once the window is created, you can initialize its properties and load content into the window using the stage property and Flash display list techniques.

In almost all cases, you should set the stage `scaleMode` property of a new native window to `noScale` (use the `StageScaleMode.NO_SCALE` constant). The Flash scale modes are designed for situations in which the application author does not know the aspect ratio of the application display space in advance. The scale modes let the author choose the least-bad compromise: clip the content, stretch or squash it, or pad it with empty space. Since you control the display space in AIR (the window frame), you can size the window to the content or the content to the window without compromise. The scale mode for Flex and HTML windows is set to `noScale` automatically.

***Note:** To determine the maximum and minimum window sizes allowed on the current operating system, use the following static NativeWindow properties:*

```
var maxOSSize:Point = NativeWindow.systemMaxSize;
var minOSSize:Point = NativeWindow.systemMinSize;
```

Creating an HTML window

To create an HTML window, you can either call the JavaScript `Window.open()` method, or you can call the AIR HTMLLoader class `createRootWindow()` method.

HTML content in any security sandbox can use the standard JavaScript `Window.open()` method. If the content is running outside the application sandbox, the `open()` method can only be called in response to user interaction, such as a mouse click or keypress. When `open()` is called, a window with system chrome is created to display the content at the specified URL. For example:

```
newWindow = window.open("xmpl.html", "logWindow", "height=600, width=400, top=10, left=10");
```

***Note:** You can extend the HTMLHost class in ActionScript to customize the window created with the JavaScript `window.open()` function. See "About extending the HTMLHost class" on page 284.*

Content in the application security sandbox has access to the more powerful method of creating windows, `HTMLLoader.createRootWindow()`. With this method, you can specify all the creation options for a new window. For example, the following code creates a lightweight type window without system chrome that is 300x400 pixels in size:

```
var options = new air.NativeWindowInitOptions();
options.systemChrome = "none";
options.type = "lightweight";

var windowBounds = new air.Rectangle(200,250,300,400);
newHTMLLoader = air.HTMLLoader.createRootWindow(true, options, true, windowBounds);
newHTMLLoader.load(new air.URLRequest("xmpl.html"));
```

***Note:** If the content loaded by a new window is outside the application security sandbox, the window object does not have the AIR properties: `runtime`, `nativeWindow`, or `htmlLoader`.*

Windows created with the `createRootWindow()` method remain independent from the opening window. The `parent` and `opener` properties of the JavaScript Window object are `null`. The opening window can access the Window object of the new window using the HTMLLoader reference returned by the `createRootWindow()` function. In the context of the previous example, the statement `newHTMLLoader.window` would reference the JavaScript Window object of the created window.

Adding content to a window

How you add content to an AIR window depends on the type of window. MXML and HTML let you declaratively define the basic content of the window. You can embed resources in the application SWF or you can load them from separate application files. Flex, Flash, and HTML content can all be created on the fly and added to a window dynamically.

When you load SWF content, or HTML content containing JavaScript, you must take the AIR security model into consideration. Any content in the application security sandbox, that is, content installed with your application and loadable with the app: URL scheme, has full privileges to access all the AIR APIs. Any content loaded from outside this sandbox cannot access the AIR APIs. JavaScript content outside the application sandbox is not able to use the `runtime`, `nativeWindow`, or `htmlLoader` properties of the JavaScript Window object.

To allow safe cross-scripting, you can use a sandbox bridge to provide a limited interface between application content and non-application content. In HTML content, you can also map pages of your application into a non-application sandbox to allow the code on that page to cross-script external content. See "AIR security" on page 69.

Loading a SWF or image

You can load Flash or images into the display list of a native window using the `flash.display.Loader` class:

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.net.URLRequest;
    import flash.display.Loader;
    public class LoadedSWF extends Sprite
    {
        public function LoadedSWF(){
            var loader:Loader = new Loader();
            loader.load(new URLRequest("visual.swf"));
            loader.contentLoaderInfo.addEventListener(Event.COMPLETE,loadFlash);
        }

        private function loadFlash(event:Event):void{
            addChild(event.target.loader);
        }
    }
}
```

You can load a SWF file that contains library code for use in an HTML-based application. The simplest way to load a SWF in an HTML window is to use the `script` tag, but you can also use the `Loader` API directly.

Note: *Older SWF files created using ActionScript 1 or 2 share global states such as class definitions, singletons, and global variables if they are loaded into the same window. If such a SWF file relies on untouched global states to work correctly, it cannot be loaded more than once into the same window, or loaded into the same window as another SWF file using overlapping class definitions and variables. This content can be loaded into separate windows.*

Loading HTML content into a NativeWindow

To load HTML content into a NativeWindow, you can either add an HTMLLoader object to the window stage and load the HTML content into the HTMLLoader, or create a window that already contains an HTMLLoader object by using the `HTMLLoader.createRootWindow()` method. The following example displays HTML content within a 300 by 500 pixel display area on the stage of a native window:

```
//newWindow is a NativeWindow instance
var htmlView:HTMLLoader = new HTMLLoader();
html.width = 300;
html.height = 500;

//set the stage so display objects are added to the top-left and not scaled
newWindow.stage.align = "TL";
newWindow.stage.scaleMode = "noScale";
newWindow.stage.addChild( htmlView );

//urlString is the URL of the HTML page to load
htmlView.load( new URLRequest(urlString) );
```

To load an HTML page into a Flex application, you can use the Flex HTML component.

Loading SWF content within an HTML page

You can load Flash or Flex SWF files in an HTML page using standard `<object>` tags. The SWF content is loaded into its own environment with an independent stage. The following tag can be used to display a SWF file on a page:

```
<object type="application/x-shockwave-flash" width="100%" height="100%">
    <movie movie="app:/SWFFile.swf"/>
</object>
```

You can also use a script to load content dynamically:

```
<script>
function showSWF(urlString){
    var display = document.getElementById("flexDisplay");
    display.appendChild(createSWFObject(urlString,650,650));
}
```

```
function createSWFObject(urlString, width, height){
    var SWFObject = document.createElement("object");
    SWFObject.setAttribute("type","application/x-shockwave-flash");
    SWFObject.setAttribute("width","100%");
    SWFObject.setAttribute("height","100%");
    var movieParam = document.createElement("param");
    movieParam.setAttribute("name","movie");
    movieParam.setAttribute("value",urlString);
    SWFObject.appendChild(movieParam);
    return SWFObject;
}
</script>
```

Adding SWF content as an overlay on an HTML window

Because HTML windows are contained within a NativeWindow instance, you can add Flash display objects both above and below the HTML layer in the display list.

To add a display object above the HTML layer, use the `addChild()` method of the `window.nativeWindow.stage` property. The `addChild()` method adds content layered above any existing content in the window.

To add a display object below the HTML layer, use the `addChildAt()` method of the `window.nativeWindow.stage` property, passing in a value of zero for the `index` parameter. Placing an object at the zero index moves existing content, including the HTML display, up one layer and insert the new content at the bottom. For content layered underneath the HTML page to be visible, you must set the `paintsDefaultBackground` property of the `HTMLlLoader` object to `false`. In addition, any elements of the page that set a background color, will not be transparent. If, for example, you set a background color for the body element of the page, none of the page will be transparent.

The following example illustrates how to add a Flash display objects as overlays and underlays to an HTML page. The example creates two simple shape objects, adds one below the HTML content and one above. The example also updates the shape position based on the `enterFrame` event.

```
<html>
<head>
<title>Bouncers</title>
<script src="AIRAliases.js" type="text/javascript"></script>
<script language="JavaScript" type="text/javascript">
air.Shape = window.runtime.flash.display.Shape;

function Bouncer(radius, color){
    this.radius = radius;
    this.color = color;

    //velocity
    this.vX = -1.3;
    this.vY = -1;

    //Create a Shape object and draw a circle with its graphics property
    this.shape = new air.Shape();
    this.shape.graphics.lineStyle(1,0);
    this.shape.graphics.beginFill(this.color,.9);
    this.shape.graphics.drawCircle(0,0,this.radius);
    this.shape.graphics.endFill();

    //Set the starting position
    this.shape.x = 100;
    this.shape.y = 100;


    //Moves the sprite by adding (vX,vY) to the current position
    this.update = function(){
        this.shape.x += this.vX;
        this.shape.y += this.vY;

        //Keep the sprite within the window
        if( this.shape.x - this.radius < 0){
            this.vX = -this.vX;
        }
        if( this.shape.y - this.radius < 0){
            this.vY = -this.vY;
        }
        if( this.shape.x  + this.radius > window.nativeWindow.stage.stageWidth){
            this.vX = -this.vX;
        }
        if( this.shape.y  + this.radius > window.nativeWindow.stage.stageHeight){
            this.vY = -this.vY;
        }

    };
}

function init(){
    //turn off the default HTML background
    window.htmlLoader.paintsDefaultBackground = false;
    var bottom = new Bouncer(60,0xff2233);
    var top = new Bouncer(30,0x2441ff);

    //listen for the enterFrame event
    window.htmlLoader.addEventListener("enterFrame",function(evt){
        bottom.update();
        top.update();
    });

    //add the bouncing shapes to the window stage
    window.nativeWindow.stage.addChildAt(bottom.shape,0);
    window.nativeWindow.stage.addChild(top.shape);
}
</script>
<body onload="init();">
<h1>de Finibus Bonorum et Malorum</h1>
<p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis
et quasi architecto beatae vitae dicta sunt explicabo.</p>
<p style="background-color:#FFFF00; color:#660000;">This paragraph has a background
color.</p>
<p>At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis
praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias
excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui
officia deserunt mollitia animi, id est laborum et dolorum fuga.</p>
</body>
</html>
```

Example: Creating a native window

The following example illustrates how to create a native window:

```
public function createNativeWindow():void {
    //create the init options
    var options:NativeWindowInitOptions = new NativeWindowInitOptions();
    options.transparent = false;
    options.systemChrome = NativeWindowSystemChrome.STANDARD;
    options.type = NativeWindowType.NORMAL;

    //create the window
    var newWindow:NativeWindow = new NativeWindow(options);
    newWindow.title = "A title";
    newWindow.width = 600;
    newWindow.height = 400;

    newWindow.stage.align = StageAlign.TOP_LEFT;
    newWindow.stage.scaleMode = StageScaleMode.NO_SCALE;

    //activate and show the new window
    newWindow.activate();
}
```

Managing windows

You use the properties and methods of the NativeWindow class to manage the appearance, behavior, and life cycle of desktop windows.

Contents

Getting a NativeWindow instance

To manipulate a window, you must first get the window instance. You can get a window instance from one of the following places:

The window constructor That is, the window constructor for a new NativeWindow.

The window stage That is, `stage.nativeWindow`.

Any display object on the stage That is, `myDisplayObject.stage.nativeWindow`.

A window event The `target` property of the event object references the window that dispatched the event.

The global nativeWindow property of an HTMLLoader or HTML window That is, `window.nativeWindow`.

The nativeApplication object `NativeApplication.nativeApplication.activeWindow` references the active window of an application (but returns `null` if the active window is not a window of this AIR application). The `NativeApplication.nativeApplication.openedWindows` array contains all of the windows in an AIR application that have not been closed.

Because the Flex Application, WindowedApplication, and Window objects are display objects, you can easily reference the application window in an MXML file using the stage property, as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
applicationComplete="init();">
```

```
    <mx:Script>
        <![CDATA[
            import flash.display.NativeWindow;

            public function init():void{
                var appWindow:NativeWindow = this.stage.nativeWindow;
                //set window properties
                appWindow.visible = true;
            }
        ]]>
    </mx:Script>
</WindowedApplication
```

***Note:** Until the WindowedApplication or Window component is added to the window stage by the Flex framework, the component's `stage` property is `null`. This behavior is consistent with that of the Flex Application component, but does mean that it is not possible to access the stage or the NativeWindow instance in listeners for events that occur earlier in the initialization cycle of the WindowedApplication and Window components, such as `creationComplete`. It is safe to access the stage and NativeWindow instance when the `applicationComplete` event is dispatched.*

Activating, showing, and hiding windows

To activate a window, call the NativeWindow `activate()` method. Activating a window brings the window to the front, gives it keyboard and mouse focus, and, if necessary, makes it visible by restoring the window or setting the `visible` property to `true`. Activating a window does not change the ordering of other windows in the application. Calling the `activate()` method causes the window to dispatch an `activate` event.

To show a hidden window without activating it, set the `visible` property to `true`. This brings the window to the front, but will not assign the focus to the window.

To hide a window from view, set its `visible` property to `false`. Hiding a window suppresses the display of both the window, any related task bar icons, and, on MacOS X, the entry in the Windows menu.

***Note:** On Mac OS X, it is not possible to completely hide a minimized window that has a dock icon. If the `visible` property is set to `false` on a minimized window, the dock icon for the window is still displayed. If the user clicks the icon, the window is restored to a visible state and displayed.*

Changing the window display order

AIR provides several methods for directly changing the display order of windows. You can move a window to the front of the display order or to the back; you can move a window above another window or behind it. At the same time, the user can reorder windows by activating them.

You can keep a window in front of other windows by setting its `alwaysInFront` property to `true`. If more than one window has this setting, then the display order of these windows is sorted among each other, but they are always sorted above windows with alwaysInFront set to false. Windows in the top-most group are also displayed above windows in other applications, even when the AIR application is not active. Because this behavior can be disruptive to a user, setting `alwaysInFront` to `true` should only be done when necessary and appropriate. Examples of justified uses include:

- Temporary pop-up windows for controls such as tooltips, pop-up lists, custom menus, or combo boxes. Because these windows should close when they lose focus, the annoyance of blocking a user from viewing another window can be avoided.
- Extremely urgent error messages and alerts. When an irrevocable change may occur if the user does not respond in a timely manner, it may be justified to push an alert window to the forefront. However, most errors and alerts can be handled in the normal window display order.
- Short-lived toast-style windows.

***Note:** AIR does not enforce proper use of the `alwaysInFront` property. However, if your application disrupts a user's workflow, it is likely to be consigned to that same user's trash can.*

The NativeWindow class provides the following properties and methods for setting the display order of a window relative to other windows:

Member	Description
alwaysInFront property	Specifies whether the window is displayed in the top-most group of windows. In almost all cases, `false` is the best setting. Changing the value from `false` to `true` brings the window to the front of all windows (but does not activate it). Changing the value from `true` to `false` orders the window behind windows remaining in the top-most group, but still in front of other windows. Setting the property to its current value for a window does not change the window display order.
orderToFront()	Brings the window to the front.
orderInFrontOf()	Brings the window directly in front of a particular window.
orderToBack()	Sends the window behind other windows.
orderBehind()	Sends the window directly behind a particular window.
activate()	Brings the window to the front (along with making the window visible and assigning focus).

***Note:** If a window is hidden (`visible` is `false`) or minimized, then calling the display order methods has no effect.*

Closing a window

To close a window, use the `NativeWindow.close()` method.

Closing a window unloads the contents of the window, but if other objects have references to this content, the content objects will not be destroyed. The `NativeWindow.close()` method executes asynchronously, the application that is contained in the window continues to run during the closing process. The close method dispatches a close event when the close operation is complete. The NativeWindow object is still technically valid, but accessing most properties and methods on a closed window generates an IllegalOperationError. You cannot reopen a closed window. Check the `closed` property of a window to test whether a window has been closed. To simply hide a window from view, set the `NativeWindow.visible` property to `false`.

If the `Nativeapplication.autoExit` property is `true`, which is the default, then the application exits when its last window closes.

Allowing cancellation of window operations

When a window uses system chrome, user interaction with the window can be canceled by listening for, and canceling the default behavior of the appropriate events. For example, when a user clicks the system chrome close button, the `closing` event is dispatched. If any registered listener calls the `preventDefault()` method of the event, then the window does not close.

When a window does not use system chrome, notification events for intended changes are not automatically dispatched before the change is made. Hence, if you call the methods for closing a window, changing the window state, or set any of the window bounds properties, the change cannot be canceled. To notify components in your application before a window change is made, your application logic can dispatch the relevant notification event using the `dispatchEvent()` method of the window.

For example, the following logic implements a cancelable event handler for a window close button:

```
public function onCloseCommand(event:MouseEvent):void{
    var closingEvent:Event = new Event(Event.CLOSING,true,true);
    dispatchEvent(closing);
    if(!closingEvent.isDefaultPrevented()){
        win.close();
    }
}
```

The `dispatchEvent()` method returns `false` if the event `preventDefault()` method is called by a listener. However, it can also return `false` for other reasons, so it is better to explicitly use the `isDefaultPrevented()` method to test whether the change should be canceled.

Maximizing, minimizing, and restoring a window

To maximize the window, use the NativeWindow `maximize()` method.

```
myWindow.maximize();
```

To minimize the window, use the NativeWindow `minimize()` method.

```
myWindow.minimize();
```

To restore the window (that is, return it to the size that it was before it was either minimized or maximized), use the NativeWindow `restore()` method.

```
myWindow.restore();
```

Note: *The behavior that results from maximizing an AIR window is different from the Mac OS X standard behavior. Rather than toggling between an application-defined "standard" size and the last size set by the user, AIR windows toggle between the size last set by the application or user and the full usable area of the screen.*

Example: Minimizing, maximizing, restoring and closing a window

The following short MXML application demonstrates the Window `maximize()`, `minimize()`, `restore()`, and `close()` methods:

```
<?xml version="1.0" encoding="utf-8"?>

<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical">


    <mx:Script>
    <![CDATA[
    public function minimizeWindow():void
    {
        this.stage.nativeWindow.minimize();
    }

    public function maximizeWindow():void
    {
        this.stage.nativeWindow.maximize();
    }

    public function restoreWindow():void
    {
        this.stage.nativeWindow.restore();
    }

    public function closeWindow():void
    {
        this.stage.nativeWindow.close();
    }
    ]]>
    </mx:Script>

    <mx:VBox>
        <mx:Button label="Minimize" click="minimizeWindow()"/>
        <mx:Button label="Restore" click="restoreWindow()"/>
        <mx:Button label="Maximize" click="maximizeWindow()"/>
        <mx:Button label="Close" click="closeWindow()"/>
    </mx:VBox>

</mx:WindowedApplication>
```

Resizing and moving a window

When a window uses system chrome, the chrome provides drag controls for resizing the window and moving around the desktop. If a window does not use system chrome you must add your own controls to allow the user to resize and move the window.

***Note:** To resize or move a window, you must first obtain a reference to the NativeWindow instance. For information about how to obtain a window reference, see "Getting a NativeWindow instance" on page 115.*

Resizing a window

To resize a window, use the NativeWindow `startResize()` method. When this method is called from a `mouseDown` event, the resizing operation is driven by the mouse and completes when the operating system receives a `mouseUp` event. When calling `startResize()`, you pass in an argument that specifies the edge or corner from which to resize the window.

The scale mode of the stage determines how the window stage and its contents behaves when a window is resized. Keep in mind that the stage scale modes are designed for situations, such as a web browser, where the application is not in control of the size or aspect ratio of its display space. In general, you get the best results by setting the stage `scaleMode` property to `StageScaleMode.NO_SCALE`. If you want the contents of the window to scale, you can still set the `scaleX` and `scaleY` parameters in response to the window bounds changes.

Moving a window

To move a window without resizing it, use the `NativeWindow startMove()` method. Like the `startResize()` method, when the `startMove()` method is called from a `mouseDown` event, the move process is mouse-driven and completes when the operating system receives a `mouseUp` event.

For more information about the `startResize()` and `startMove()` methods, see the *Flex 3 Language Reference (http://www.adobe.com/go/learn_flex3_aslr).*

Example: Resizing and moving windows

The following example shows how to initiate resizing and moving operations on a window:

```
package
{
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.display.NativeWindowResize;

    public class NativeWindowResizeExample extends Sprite
    {
        public function NativeWindowResizeExample():void
        {
            // Fills a background area.
            this.graphics.beginFill(0xFFFFFF);
            this.graphics.drawRect(0, 0, 400, 300);
            this.graphics.endFill();

            // Creates a square area where a mouse down will start the resize.
            var resizeHandle:Sprite =
                createSprite(0xCCCCCC, 20, this.width - 20, this.height - 20);
            resizeHandle.addEventListener(MouseEvent.MOUSE_DOWN, onStartResize);

            // Creates a square area where a mouse down will start the move.
            var moveHandle:Sprite = createSprite(0xCCCCCC, 20, this.width - 20, 0);
            moveHandle.addEventListener(MouseEvent.MOUSE_DOWN, onStartMove);
        }

        public function createSprite(color:int, size:int, x:int, y:int):Sprite
        {
            var s:Sprite = new Sprite();
            s.graphics.beginFill(color);
            s.graphics.drawRect(0, 0, size, size);
            s.graphics.endFill();
            s.x = x;
            s.y = y;
            this.addChild(s);
            return s;
        }

        public function onStartResize(event:MouseEvent):void
        {
            this.stage.nativeWindow.startResize(NativeWindowResize.BOTTOM_RIGHT);
        }

        public function onStartMove(event:MouseEvent):void
        {
            this.stage.nativeWindow.startMove();
        }
    }
}
```

Listening for window events

To listen for the events dispatched by a window, register a listener with the window instance. For example, to listen for the closing event, register a listener with the window as follows:

```
myWindow.addEventListener(Event.CLOSING, onClosingEvent);
```

When an event is dispatched, the `target` property references the window sending the event.

Most window events have two related messages. The first message signals that a window change is imminent (and can be canceled), while the second message signals that the change has occurred. For example, when a user clicks the close button of a window, the closing event message is dispatched. If no listeners cancel the event, the window closes and the close event is dispatched to any listeners.

Typically, the warning events, such as `closing`, are only dispatched when system chrome has been used to trigger an event. Calling the window `close()` method, for example, does not automatically dispatch the `closing` event—only the `close` event is dispatched. You can, however, construct a closing event object and dispatch it using the window `dispatchEvent()` method.

The window events that dispatch an Event object are:

Event	Description
activate	Dispatched when the window receives focus.
deactivate	Dispatched when the window loses focus
closing	Dispatched when the window is about to close. This only occurs automatically when the system chrome close button is pressed or, on Mac OS X, when the Quit command is invoked.
close	Dispatched when the window has closed.

The window events that dispatch an NativeWindowBoundsEvent object are:

Event	Description
moving	Dispatched immediately before the top-left corner of the window changes position, either as a result of moving, resizing or changing the window display state.
move	Dispatched after the top-left corner has changed position.
resizing	Dispatched immediately before the window width or height changes either as a result of resizing or a display state change.
resize	Dispatched after the window has changed size.

For NativeWindowBoundsEvent events, you can use the `beforeBounds` and `afterBounds` properties to determine the window bounds before and after the impending or completed change.

The window events that dispatch an NativeWindowDisplayStateEvent object are:

Event	Description
displayStateChanging	Dispatched immediately before the window display state changes.
displayStateChange	Dispatched after the window display state has changed.

For NativeWindowDisplayStateEvent events, you can use the `beforeDisplayState` and `afterDisplayState` properties to determine the window display state before and after the impending or completed change.

Displaying full-screen windows

Setting the `displayState` property of the Stage to `StageDisplayState.FULL_SCREEN_INTERACTIVE` puts the window in full-screen mode, and keyboard input *is* permitted in this mode. (In SWF content running in a browser, keyboard input is not permitted). To exit full-screen mode, the user presses the Escape key.

For example, the following Flex code defines a simple AIR application that sets up a simple full-screen terminal:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    applicationComplete="init()" backgroundColor="0x003030" focusRect="false">
    <mx:Script>
        <![CDATA[
            private function init():void
            {
                stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
                focusManager.setFocus(terminal);
                terminal.text = "Welcome to the dumb terminal app. Press the ESC key to 
exit..\n";
                terminal.selectionBeginIndex = terminal.text.length;
                terminal.selectionEndIndex = terminal.text.length;
            }
        ]]>
    </mx:Script>
    <mx:TextArea 
        id="terminal" 
        height="100%" width="100%" 
        scroll="false" 
        backgroundColor="0x003030" 
        color="0xCCFF00" 
        fontFamily="Lucida Console" 
        fontSize="44"/>
</mx:WindowedApplication>
```

Chapter 14: Screens

Use the Adobe® AIR® Screen class to access information about the desktop display screens attached to a computer.

Contents

Quick Starts (Adobe AIR Developer Center)

- Measuring the virtual desktop

Language Reference

- Screen

More information

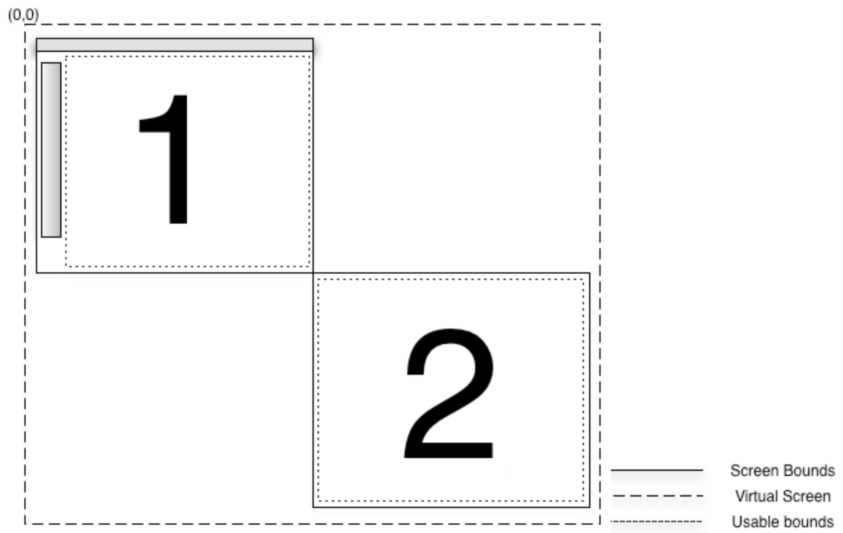
- Adobe AIR Developer Center for Flex (search for 'AIR screens')

Screen basics

The screen API contains a single class, Screen, which provides static members for getting system screen information, and instance members for describing a particular screen.

A computer system can have several monitors or displays attached, which can correspond to several desktop screens arranged in a virtual space. The AIR Screen class provides information about the screens, their relative arrangement, and their usable space. If more than one monitor maps to the same screen, only one screen exists. If the size of a screen is larger than the display area of the monitor, there is no way to determine which portion of the screen is currently visible.

A screen represents an independent desktop display area. Screens are described as rectangles within the virtual desktop. The top-left corner of screen designated as the primary display is the origin of the virtual desktop coordinate system. All values used to describe a screen are provided in pixels.



In this screen arrangement, two screens exist on the virtual desktop. The coordinates of the top-left corner of the main screen (#1) are always (0,0). If the screen arrangement is changed to designate screen #2 as the main screen, then the coordinates of screen #1 become negative. Menubars, taskbars, and docks are excluded when reporting the usable bounds for a screen.

For detailed information about the screen API class, methods, properties, and events, see the *Flex ActionScript 3.0 Language Reference (http://www.adobe.com/go/learn_flex3_aslr).*

Enumerating the screens

You can enumerate the screens of the virtual desktop with the following screen methods and properties:

Method or Property	**Description**
Screen.screens	Provides an array of Screen objects describing the available screens. Note that the order of the array is not significant.
Screen.mainScreen	Provides a Screen object for the main screen. On Mac OS X, the main screen is the screen displaying the menu bar. On Windows, the main screen is the system-designated primary screen.
Screen.getScreensForRectangle()	Provides an array of Screen objects describing the screens intersected by the given rectangle. The rectangle passed to this method is in pixel coordinates on the virtual desktop. If no screens intersect the rectangle, then the array is empty. You can use this method to find out on which screens a window is displayed.

You should not save the values returned by the Screen class methods and properties. The user or operating system can change the available screens and their arrangement at any time.

The following example uses the screen API to move a window between multiple screens in response to pressing the arrow keys. To move the window to the next screen, the example gets the `screens` array and sorts it either vertically or horizontally (depending on the arrow key pressed). The code then walks through the sorted array, comparing each screen to the coordinates of the current screen. To identify the current screen of the window, the example calls `Screen.getScreensForRectangle()`, passing in the window bounds.

```
package {
    import flash.display.Sprite;
    import flash.display.Screen;
    import flash.events.KeyboardEvent;
    import flash.ui.Keyboard;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;

    public class ScreenExample extends Sprite
    {
        public function ScreenExample()
        {
                stage.align = StageAlign.TOP_LEFT;
                stage.scaleMode = StageScaleMode.NO_SCALE;

                stage.addEventListener(KeyboardEvent.KEY_DOWN,onKey);
        }

        private function onKey(event:KeyboardEvent):void{
            if(Screen.screens.length > 1){
                switch(event.keyCode){
                    case Keyboard.LEFT :
                        moveLeft();
                        break;
                    case Keyboard.RIGHT :
                        moveRight();
                        break;
                    case Keyboard.UP :
                        moveUp();
                        break;
                    case Keyboard.DOWN :
                        moveDown();
                        break;
                }
            }
        }

        private function moveLeft():void{
            var currentScreen = getCurrentScreen();
            var left:Array = Screen.screens;
            left.sort(sortHorizontal);
            for(var i:int = 0; i < left.length - 1; i++){
                if(left[i].bounds.left < stage.nativeWindow.bounds.left){
                    stage.nativeWindow.x +=
                        left[i].bounds.left - currentScreen.bounds.left;
                    stage.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
                }
            }
        }

        private function moveRight():void{
            var currentScreen:Screen = getCurrentScreen();
            var left:Array = Screen.screens;
            left.sort(sortHorizontal);
            for(var i:int = left.length - 1; i > 0; i--){
                if(left[i].bounds.left > stage.nativeWindow.bounds.left){
                    stage.nativeWindow.x +=
                        left[i].bounds.left - currentScreen.bounds.left;
                    stage.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
                }
            }
        }

        private function moveUp():void{
            var currentScreen:Screen = getCurrentScreen();
            var top:Array = Screen.screens;
            top.sort(sortVertical);
            for(var i:int = 0; i < top.length - 1; i++){
                if(top[i].bounds.top < stage.nativeWindow.bounds.top){
                    stage.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
                    stage.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
                    break;
                }
            }
        }

        private function moveDown():void{
            var currentScreen:Screen = getCurrentScreen();

            var top:Array = Screen.screens;
            top.sort(sortVertical);
            for(var i:int = top.length - 1; i > 0; i--){
                if(top[i].bounds.top > stage.nativeWindow.bounds.top){
                    stage.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
                    stage.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
                    break;
                }
            }
        }

        private function sortHorizontal(a:Screen,b:Screen):int{
            if (a.bounds.left > b.bounds.left){
                return 1;
            } else if (a.bounds.left < b.bounds.left){
                return -1;
            } else {return 0;}
        }

        private function sortVertical(a:Screen,b:Screen):int{
            if (a.bounds.top > b.bounds.top){
                return 1;
            } else if (a.bounds.top < b.bounds.top){
                return -1;
            } else {return 0;}
        }

        private function getCurrentScreen():Screen{
            var current:Screen;
            var screens:Array = Screen.getScreensForRectangle(stage.nativeWindow.bounds);
            (screens.length > 0) ? current = screens[0] : current = Screen.mainScreen;
            return current;
        }
    }
}
```

Chapter 15: Working with native menus

Use the classes in the native menu API to define application, window, context, and pop-up menus.

Contents

Quick Starts (Adobe AIR Developer Center)

- Adding native menus to an AIR application

Language Reference

- NativeMenu
- NativeMenuItem

More information

- Adobe AIR Developer Center for Flex (search for 'AIR menus')

AIR menu basics

The native menu classes allow you to access the native menu features of the operating system on which your application is running. NativeMenu objects can be used for application menus (available on OS X), window menus (available on Windows), context menus, and pop-up menus.

Contents

AIR menu classes

The Adobe® AIR™ Menu classes include:

Package	**Classes**
flash.display	• NativeMenu • NativeMenuItem
flash.ui	• ContextMenu • ContextMenuItem
flash.events	• Event • ContextMenuEvent

Menu varieties

AIR supports the following types of menus:

Application menus An application menu is a global menu that applies to the entire application. Application menus are supported on Mac OS X, but not on Windows. On Mac OS X, the operating system automatically creates an application menu. You can use the AIR menu API to add items and submenus to the standard menus. You can add listeners for handling the existing menu commands. Or you can remove existing items.

Window menus A window menu is associated with a single window and is displayed below the title bar. Menus can be added to a window by creating a NativeMenu object and assigning it to the `menu` property of the NativeWindow object. Window menus are supported on the Windows operating system, but not on Mac OS X. Native menus can only be used with windows that have system chrome.

Context menus Context menus open in response to a right-click or command-click on an interactive object in SWF content or a document element in HTML content. You can create a context menu using the AIR NativeMenu class. (You can also use the legacy Adobe® Flash® ContextMenu class.) In HTML content, you can use the Webkit HTML and JavaScript APIs to add context menus to an HTML element.

Dock and system tray icon menus These icon menus are similar to context menus and are assigned to an application icon in the Mac OS X dock or Windows notification area. Dock and system tray icon menus use the NativeMenu class. On Mac OS X, the items in the menu are added above the standard operating system items. On Windows, there is no standard menu.

Pop-up menus An AIR pop-up menu is like a context menu, but is not necessarily associated with a particular application object or component. Pop-up menus can be displayed anywhere in a window by calling the `display()` method of any NativeMenu object.

Flex menus The Adobe® Flex™ framework provides a set of Flex menu components. The Flex menus are drawn by the Adobe AIR rather than the operating system and are not *native* menus. A Flex menu component can be used for Flex windows that do not have system chrome. Another benefit of using the Flex menu component is that you can specify menus declaratively in MXML format. If you are using the Flex Framework, use the Flex menu classes for window menus instead of the native classes. See “About the FlexNativeMenu control” on page 46.

Custom menus Native menus are drawn entirely by the operating system and, as such, exist outside the Flash and HTML rendering models. You are free to create your own non-native menus using MXML, ActionScript, or JavaScript. The AIR menu classes do not provide any facility for controlling the drawing of native menus.

Default menus

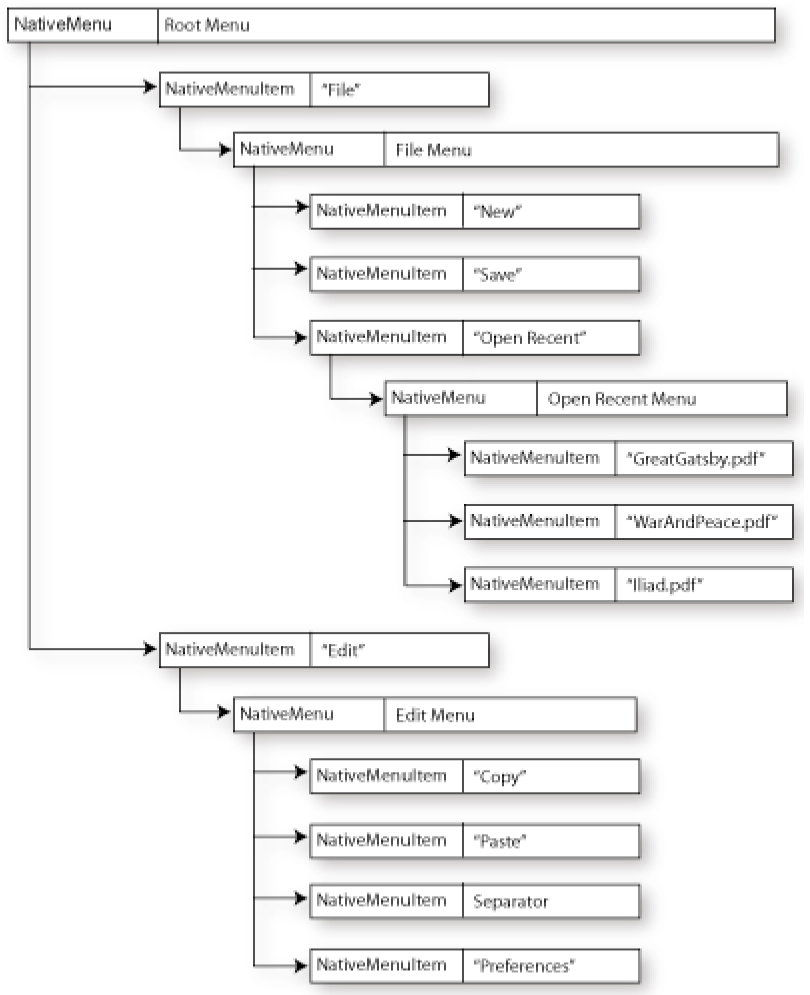
The following default menus are provided by the operating system or a built-in AIR class:

- Application menu on Mac OS X
- Dock icon menu on Mac OS X
- Context menu for selected text and images in HTML content
- Context menu for selected text in a TextField object (or an object that extends TextField)

Menu structure

Menus are hierarchical in nature. NativeMenu objects contain child NativeMenuItem objects. NativeMenuItem objects that represent submenus, in turn, can contain NativeMenu objects. The top- or root-level menu object in the structure represents the menu bar for application and window menus. (Context, icon, and pop-up menus don't have a menu bar).

The following diagram illustrates the structure of a typical menu. The root menu represents the menu bar and contains two menu items referencing a *File* submenu and an *Edit* submenu. The File submenu in this structure contains two command items and an item that references an *Open Recent Menu* submenu, which, itself, contains three items. The Edit submenu contains three commands and a separator.



Defining a submenu requires both a NativeMenu and a NativeMenuItem object. The NativeMenuItem object defines the label displayed in the parent menu and allows the user to open the submenu. The NativeMenu object serves as a container for items in the submenu. The NativeMenuItem object references the NativeMenu object through the NativeMenuItem `submenu` property.

To view a code example that creates this menu see “Example: Window and application menu” on page 138.

Menu events

NativeMenu and NativeMenuItem objects both dispatch `displaying` and `select` events:

Displaying: Immediately before a menu is displayed, the menu and its menu items dispatch a `displaying` event to any registered listeners. The `displaying` event gives you an opportunity to update the menu contents or item appearance before it is shown to the user. For example, in the listener for the `displaying` event of an “Open Recent” menu, you could change the menu items to reflect the current list of recently viewed documents.

The `target` property of the event object is always the menu that is about to be displayed. The `currentTarget` is the object on which the listener is registered: either the menu itself, or one of its items.

***Note:** The displaying event is also dispatched whenever the state of the menu or one of its items is accessed.*

Select: When a command item is chosen by the user, the item dispatches a `select` event to any registered listeners. Submenu and separator items cannot be selected and so never dispatch a `select` event.

A `select` event bubbles up from a menu item to its containing menu, on up to the root menu. You can listen for `select` events directly on an item and you can listen higher up in the menu structure. When you listen for the `select` event on a menu, you can identify the selected item using the event `target` property. As the event bubbles up through the menu hierarchy, the `currentTarget` property of the event object identifies the current menu object.

***Note:** ContextMenu and ContextMenuItem objects dispatch `menuItemSelect` and `menuSelect` events as well as `select` and `displaying` events.*

Key equivalents for menu commands

You can assign a key equivalent (sometimes called an accelerator) to a menu command. The menu item dispatches a `select` event to any registered listeners when the key, or key combination is pressed. The menu containing the item must be part of the menu of the application or the active window for the command to be invoked.

Key equivalents have two parts, a string representing the primary key and an array of modifier keys that must also be pressed. To assign the primary key, set the menu item `keyEquivalent` property to the single character string for that key. If you use an uppercase letter, the shift key is added to the modifier array automatically.

On Mac OS X, the default modifier is the command key (`Keyboard.COMMAND`). On Windows, it is the control key (`Keyboard.CONTROL`). These default keys are automatically added to the modifier array. To assign different modifier keys, assign a new array containing the desired key codes to the `keyEquivalentModifiers` property. The default array is overwritten. Whether or not you use the default modifiers or assign your own modifier array, the shift key is added if the string you assign to the `keyEquivalent` property is an uppercase letter. Constants for the key codes to use for the modifier keys are defined in the Keyboard class.

The assigned key equivalent string is automatically displayed beside the menu item name. The format depends on the user’s operating system and system preferences.

***Note:** If you assign the `Keyboard.COMMAND` value to a key modifier array on the Windows operating system, no key equivalent is displayed in the menu. However, the control key must be used to activate the menu command.*

The following example assigns `Ctrl+Shift+G` as the key equivalent for a menu item:

```
var item:NativeMenuItem = new NativeMenuItem("Ungroup");
item.keyEquivalent = "G";
```

This example assigns `Ctrl+Shift+G` as the key equivalent by setting the modifier array directly:

```
var item:NativeMenuItem = new NativeMenuItem("Ungroup");
item.keyEquivalent = "G";
item.keyEquivalentModifiers = [Keyboard.CONTROL];
```

***Note:** Key equivalents are only triggered for application and window menus. If you add a key equivalent to a context or pop-up menu, the key equivalent is displayed in the menu label, but the associated menu command is never invoked.*

Mnemonics

Mnemonics are part of the operating system keyboard interface to menus. Both Mac OS X and Windows allow users to open menus and select commands with the keyboard, but there are subtle differences. On Mac OS X, the user types the first letter or two of the menu or command and then types return.

On Windows, only a single letter is significant. By default, the significant letter is the first character in the label, but if you assign a mnemonic to the menu item, then the significant character becomes the designated letter. If two items in a menu have the same significant character (whether or not a mnemonic has been assigned), then the user's keyboard interaction with the menu changes slightly. Instead of pressing a single letter to select the menu or command, the user must press the letter as many times as necessary to highlight the desired item and then press the enter key to complete the selection. To maintain a consistent behavior, it is advisable to assign a unique mnemonic to each item in a menu for window menus.

Specify the mnemonic character as an index into the label string. The index of the first character in a label is 0. Thus, to use "r" as the mnemonic for a menu item labeled, "Format," you would set the `mnemonicIndex` property equal to 2.

```
var item:NativeMenuItem = new NativeMenuItem("Format");
item.mnemonicIndex = 2;
```

Menu item state

Menu items have the two state properties, `checked` and `enabled`:

checked Set to `true` to display a check mark next to the item label.

```
var item:NativeMenuItem = new NativeMenuItem("Format");
item.checked = true;
```

enabled Toggle the value between `true` and `false` to control whether the command is enabled. Disabled items are visually "grayed-out" and do not dispatch `select` events.

```
var item:NativeMenuItem = new NativeMenuItem("Format");
item.enabled = false;
```

Attaching an object to a menu item

The `data` property of the NativeMenuItem class allows you to reference an arbitrary object in each item. For example, in an "Open Recent" menu, you could assign the File object for each document to each menu item.

```
var file:File = File.applicationStorageDirectory.resolvePath("GreatGatsby.pdf")
var menuItem:NativeMenuItem = docMenu.addItem(new NativeMenuItem(file.name));
menuItem.data = file;
```

Creating native menus

This topic describes how to create the various types of native menu supported by AIR.

Creating a root menu object

To create a NativeMenu object to serve as the root of the menu, use the NativeMenu constructor:

```
var root:NativeMenu = new NativeMenu();
```

For application and window menus, the root menu represents the menu bar and should only contain items that open submenus. Context menu and pop-up menus do not have a menu bar, so the root menu can contain commands and separator lines as well as submenus.

After the menu is created, you can add menu items. Items appear in the menu in the order in which they are added, unless you add the items at a specific index using the `addItemAt()` method of a menu object.

Assign the menu as an application, window, icon, or context menu, or display it as a pop-up menu as shown in the following sections:

Setting the application menu

```
NativeApplication.nativeApplication.menu = root;
```

***Note:** Mac OS X defines a menu containing standard items for every application. Assigning a new NativeMenu object to the `menu` property of the NativeApplication object replaces the standard menu. You can also use the standard menu instead of replacing it.*

Setting a window menu

```
nativeWindowObject.menu = root;
```

Setting a context menu on an interactive object

```
interactiveObject.contextMenu = root;
```

Setting a dock icon menu

```
DockIcon(NativeApplication.nativeApplication.icon).menu = root;
```

***Note:** Mac OS X defines a standard menu for the application dock icon. When you assign a new NativeMenu to the menu property of the DockIcon object, the items in that menu are displayed above the standard items. You cannot remove, access, or modify the standard menu items.*

Setting a system tray icon menu

```
SystemTrayIcon(NativeApplication.nativeApplication.icon).menu = root;
```

Displaying a menu as a pop-up

```
root.display(stage, x, y);
```

Creating a submenu

To create a submenu, you add a NativeMenuItem object to the parent menu and then assign the NativeMenu object defining the submenu to the item's `submenu` property. AIR provides two ways to create submenu items and their associated menu object:

You can create a menu item and its related menu object in one step with the `addSubmenu()` method:

```
var editMenuItem:NativeMenuItem = root.addSubmenu(new NativeMenu(), "Edit");
```

You can also create the menu item and assign the menu object to its `submenu` property separately:

```
var editMenuItem:NativeMenuItem = root.addItem("Edit", false);
editMenuItem.submenu = new NativeMenu();
```

Creating a menu command

To create a menu command, add a NativeMenuItem object to a menu and add an event listener referencing the function implementing the menu command:

```
var copy:NativeMenuItem = new NativeMenuItem("Copy", false);
copy.addEventListener(Event.SELECT, onCopyCommand);
editMenu.addItem(copy);
```

You can listen for the `select` event on the command item itself (as shown in the example), or you can listen for the `select` event on a parent menu object.

Note: *Menu items that represent submenus and separator lines do not dispatch* `select` *events and so cannot be used as commands.*

Creating a menu separator line

To create a separator line, create a NativeMenuItem, setting the `isSeparator` parameter to `true` in the constructor. Then add the separator item to the menu in the correct location:

```
var separatorA:NativeMenuItem = new NativeMenuItem("A", true);
editMenu.addItem(separatorA);
```

The label specified for the separator, if any, is not displayed.

See also

- "About context menus" on page 133
- "Defining native menus declaratively" on page 135

About context menus

In SWF content, any object that inherits from InteractiveObject can be given a context menu by assigning a menu object to its `contextMenu` property. The menu object assigned to `contextMenu` can either be of type NativeMenu or of type ContextMenu.

The legacy context menu API classes allow you to use existing ActionScript code that already contains context menus. If you use the ContextMenu class, you must use the ContextMenuItem class with it; you cannot add NativeMenuItem objects to a ContextMenu object, nor can you add ContextMenuItem objects to a NativeMenu object. The primary drawback to using the context menu API is that it does not support submenus.

Although the ContextMenu class includes methods, such as `addItem()`, that are inherited from the NativeMenu class, these methods add items to the incorrect items array. In a context menu, all items must be added to the `customItems` array, not the `items` array. Either use NativeMenu objects for context menus, or use only the non-inherited ContextMenu methods and properties for adding and managing items in the menu.

The following example creates a Sprite and adds a simple edit context menu:

```
var sprite:Sprite = new Sprite();
sprite.contextMenu = createContextMenu()
private function createContextMenu():ContextMenu{
    var editContextMenu:ContextMenu = new ContextMenu();
    var cutItem:ContextMenuItem = new ContextMenuItem("Cut")
    cutItem.addEventListener(ContextMenuEvent.MENU_ITEM_SELECT, doCutCommand);
    editContextMenu.customItems.push(cutItem);

    var copyItem:ContextMenuItem = new ContextMenuItem("Copy")
    copyItem.addEventListener(ContextMenuEvent.MENU_ITEM_SELECT, doCopyCommand);
    editContextMenu.customItems.push(copyItem);

    var pasteItem:ContextMenuItem = new ContextMenuItem("Paste")
    pasteItem.addEventListener(ContextMenuEvent.MENU_ITEM_SELECT, doPasteCommand);
    editContextMenu.customItems.push(pasteItem);

    return editContextMenu
}
private function doCutCommand(event:ContextMenuEvent):void{trace("cut");}
private function doCopyCommand(event:ContextMenuEvent):void{trace("copy");}
private function doPasteCommand(event:ContextMenuEvent):void{trace("paste");}
```

***Note:** In contrast to SWF content displayed in a browser environment, context menus in AIR do not have any built-in commands.*

About context menus in HTML

In HTML content, the `contextmenu` event can be used to display a context menu. By default, a context menu is displayed automatically when the user invokes the context menu event on selected text (by right-clicking or command-clicking the text). To prevent the default menu from opening, listen for the `contextmenu` event and call the event object's `preventDefault()` method:

```
function showContextMenu(event){
    event.preventDefault();
}
```

You can then display a custom context menu using DHTML techniques or by displaying an AIR native context menu. The following example displays a native context menu by calling the menu `display()` method in response to the HTML `contextmenu` event:

```
<html>
<head>
<script src="AIRAliases.js" language="JavaScript" type="text/javascript"></script>
<script language="javascript" type="text/javascript">

function showContextMenu(event){
    event.preventDefault();
    contextMenu.display(window.nativeWindow.stage, event.clientX, event.clientY);
}

function createContextMenu(){
    var menu = new air.NativeMenu();
    var command = menu.addItem(new air.NativeMenuItem("Custom command"));
    command.addEventListener(air.Event.SELECT, onCommand);
    return menu;
}

function onCommand(){
    air.trace("Context command invoked.");
}

var contextMenu = createContextMenu();
</script>
</head>
<body>
<p oncontextmenu="showContextMenu(event)" style="-khtml-user-select:auto;">Custom context
menu.</p>
</body>
</html>
```

Defining native menus declaratively

Coding the properties of a menu and menu items can be a bit tedious. However, since menus have a natural hierarchical structure, it is straightforward to write a function that creates a menu using an XML-formatted definition.

The following class extends NativeMenu, taking an XML object in its constructor, to do just that:

```
package
{
    import flash.display.NativeMenu;
    import flash.display.NativeMenuItem;
    import flash.events.Event;

    public class DeclarativeMenu extends NativeMenu
    {
        public function DeclarativeMenu(XMLMenuDefinition:XML):void
        {
            super();
            addChildrenToMenu(this, XMLMenuDefinition.children());
        }

        private function addChildrenToMenu(menu:NativeMenu,
                                children:XMLList):NativeMenuItem
        {
            var menuItem:NativeMenuItem;
            var submenu:NativeMenu;

            for each (var child:XML in children)
            {
                if (String(child.@label).length > 0)
                {
                    menuItem = new NativeMenuItem(child.@label);
                    menuItem.name = child.name();
                }
                else
                {
                    menuItem = new NativeMenuItem(child.name());
                    menuItem.name = child.name();
                }
                menu.addItem(menuItem);
                if (child.children().length() > 0)
                {
                    menuItem.submenu = new NativeMenu();
                    addChildrenToMenu(menuItem.submenu,child.children());
                }
            }
            return menuItem;
        }
    } //End class
} //End package
```

To create a menu with this class, pass an XML menu definition as follows:

```
var menuDefinition:XML =
    <root>
        <FileMenu label='File'>
            <NewMenu label='New'>
                <NewTextFile label='Text file'/>
                <NewFolder label='Folder'/>
                <NewProject label='Project'/>
            </NewMenu>
            <OpenCommand label='Open'/>
            <SaveCommand label='Save'/>
        </FileMenu>
        <EditMenu label='Edit'>
            <CutCommand label='Cut'/>
            <CopyCommand label='Copy'/>
            <PasteCommand label='Paste'/>
        </EditMenu>
        <FoodItems label='Food Items'>
            <Jellyfish/>
            <Tripe/>
            <Gizzard/>
        </FoodItems>
    </root>;
var test:DeclarativeMenu = new DeclarativeMenu(menuDefinition);
```

To listen for menu events, you could listen at the root menu level and use the `event.target.name` property to detect which command was selected. You could also look up items in the menu by name and add individual event listeners.

Displaying pop-up menus

You can display any NativeMenu object at an arbitrary time and location above a window, by calling the menu `display()` method. The method requires a reference to the stage; thus, only content in the application sandbox can display a menu as a pop-up.

The following method displays the menu defined by a NativeMenu object named `popupMenu` in response to a mouse click:

```
private function onMouseClick(event:MouseEvent):void {
    popupMenu.display(event.target.stage, event.stageX, event.stageY);
}
```

***Note:** The menu does not need to be displayed in direct response to an event. Any method can call the `display()` function.*

Handling menu events

A menu dispatches events when the user selects the menu or when the user selects a menu item.

Contents

Events summary for menu classes

Add event listeners to menus or individual items to handle menu events.

Object	**Events dispatched**
NativeMenu	NativeMenuEvent.DISPLAYING NativeMenuEvent.SELECT (propagated from child items and submenus)
NativeMenuItem	NativeMenuEvent.SELECT NativeMenuEvent.DISPLAYING (propagated from parent menu)
ContextMenu	ContextMenuEvent.MENU_SELECT
ContextMenuItem	ContextMenuEvent.MENU_ITEM_SELECT NativeMenu.SELECT

Selecting menu events

To handle a click on a menu item, add an event listener for the `select` event to the NativeMenuItem object:

```
var menuCommandX:NativeMenuItem = new NativeMenuItem("Command X");
menuCommand.addEventListener(Event.SELECT, doCommandX)
```

Because `select` events bubble up to the containing menus, you can also listen for select events on a parent menu. When listening at the menu level, you can use the event object `target` property to determine which menu command was selected. The following example traces the label of the selected command:

```
var colorMenuItem:NativeMenuItem = new NativeMenuItem("Choose a color");
var colorMenu:NativeMenu = new NativeMenu();
colorMenuItem.submenu = colorMenu;

var red:NativeMenuItem = new NativeMenuItem("Red");
var green:NativeMenuItem = new NativeMenuItem("Green");
var blue:NativeMenuItem = new NativeMenuItem("Blue");
colorMenu.addItem(red);
colorMenu.addItem(green);
colorMenu.addItem(blue);

if(NativeApplication.supportsMenu){
    NativeApplication.nativeApplication.menu.addItem(colorMenuItem);
    NativeApplication.nativeApplication.menu.addEventListener(Event.SELECT, colorChoice);
} else if (NativeWindow.supportsMenu){
    var windowMenu:NativeMenu = new NativeMenu();
    this.stage.nativeWindow.menu = windowMenu;
    windowMenu.addItem(colorMenuItem);
    windowMenu.addEventListener(Event.SELECT, colorChoice);
}

function colorChoice(event:Event):void {
    var menuItem:NativeMenuItem = event.target as NativeMenuItem;
    trace(menuItem.label + " has been selected");
}
```

If you are using the ContextMenuItem class, you can listen for either the `select` event or the `menuItemSelect` event. The `menuItemSelect` event gives you additional information about the object owning the context menu, but does not bubble up to the containing menus.

Displaying menu events

To handle the opening of a menu, you can add a listener for the `displaying` event, which is dispatched before a menu is displayed. You can use the displaying event to update the menu, for example by adding or removing items, or by updating the enabled or checked states of individual items.

Example: Window and application menu

The following example creates the menu shown in "Menu structure" on page 129.

The menu is designed to work both on Windows, for which only window menus are supported, and on Mac OS X, for which only application menus are supported. To make the distinction, the MenuExample class constructor checks the static `supportsMenu` properties of the NativeWindow and NativeApplication classes. If `NativeWindow.supportsMenu` is `true`, then the constructor creates a NativeMenu object for the window and then creates and adds the File and Edit submenus. If `NativeApplication.supportsMenu` is `true`, then the constructor creates and adds the File and Edit menus to the existing menu provided by the OS X operating system.

The example also illustrates menu event handling. The `select` event is handled at the item level and also at the menu level. Each menu in the chain from the menu containing the selected item to the root menu responds to the `select` event. The `displaying` event is used with the "Open Recent" menu. Just before the menu is opened, the items in the menu are refreshed from the recent Documents array (which doesn't actually change in this example). Although not shown in this example, you can also listen for `displaying` events on individual items.

```
package {
    import flash.display.NativeMenu;
    import flash.display.NativeMenuItem;
    import flash.display.NativeWindow;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.filesystem.File;
    import flash.desktop.NativeApplication;

    public class MenuExample extends Sprite
    {
        private var recentDocuments:Array = 
            new Array(new File("app-storage:/GreatGatsby.pdf"), 
                      new File("app-storage:/WarAndPeace.pdf"), 
                      new File("app-storage:/Iliad.pdf"));

        public function MenuExample()
        {
            var fileMenu:NativeMenuItem;
            var editMenu:NativeMenuItem;

            if (NativeWindow.supportsMenu){
                stage.nativeWindow.menu = new NativeMenu();
                stage.nativeWindow.menu.addEventListener(Event.SELECT, selectCommandMenu);
                fileMenu = stage.nativeWindow.menu.addItem(new NativeMenuItem("File"));
                fileMenu.submenu = createFileMenu();
                editMenu = stage.nativeWindow.menu.addItem(new NativeMenuItem("Edit"));
                editMenu.submenu = createEditMenu();
            }

            if (NativeApplication.supportsMenu){
                NativeApplication.nativeApplication.menu.addEventListener(Event.SELECT,
selectCommandMenu);
                fileMenu = NativeApplication.nativeApplication.menu.addItem(new
NativeMenuItem("File"));
                fileMenu.submenu = createFileMenu();
                editMenu = NativeApplication.nativeApplication.menu.addItem(new
NativeMenuItem("Edit"));
                editMenu.submenu = createEditMenu();
            }
        }

        public function createFileMenu():NativeMenu {
            var fileMenu:NativeMenu = new NativeMenu();
            fileMenu.addEventListener(Event.SELECT, selectCommandMenu);

            var newCommand:NativeMenuItem = fileMenu.addItem(new NativeMenuItem("New"));
            newCommand.addEventListener(Event.SELECT, selectCommand);
            var saveCommand:NativeMenuItem = fileMenu.addItem(new NativeMenuItem("Save"));
            saveCommand.addEventListener(Event.SELECT, selectCommand);
            var openRecentMenu:NativeMenuItem =
                    fileMenu.addItem(new NativeMenuItem("Open Recent"));
            openRecentMenu.submenu = new NativeMenu();
            openRecentMenu.submenu.addEventListener(Event.DISPLAYING,
                                            updateRecentDocumentMenu);
            openRecentMenu.submenu.addEventListener(Event.SELECT, selectCommandMenu);

            return fileMenu;
        }

        public function createEditMenu():NativeMenu {
            var editMenu:NativeMenu = new NativeMenu();
            editMenu.addEventListener(Event.SELECT, selectCommandMenu);

            var copyCommand:NativeMenuItem = editMenu.addItem(new NativeMenuItem("Copy"));
            copyCommand.addEventListener(Event.SELECT, selectCommand);
            copyCommand.keyEquivalent = "c";
            var pasteCommand:NativeMenuItem =
                    editMenu.addItem(new NativeMenuItem("Paste"));
            pasteCommand.addEventListener(Event.SELECT, selectCommand);
            pasteCommand.keyEquivalent = "v";
            editMenu.addItem(new NativeMenuItem("", true));
            var preferencesCommand:NativeMenuItem =
                    editMenu.addItem(new NativeMenuItem("Preferences"));
            preferencesCommand.addEventListener(Event.SELECT, selectCommand);

            return editMenu;
        }

        private function updateRecentDocumentMenu(event:Event):void {
            trace("Updating recent document menu.");
            var docMenu:NativeMenu = NativeMenu(event.target);
            for each (var item:NativeMenuItem in docMenu.items) {
                docMenu.removeItem(item);
            }

            for each (var file:File in recentDocuments) {
                var menuItem:NativeMenuItem =
                        docMenu.addItem(new NativeMenuItem(file.name));
                menuItem.data = file;
                menuItem.addEventListener(Event.SELECT, selectRecentDocument);
            }
        }

        private function selectRecentDocument(event:Event):void {
            trace("Selected recent document: " + event.target.data.name);
        }

        private function selectCommand(event:Event):void {
            trace("Selected command: " + event.target.label);
        }

        private function selectCommandMenu(event:Event):void {
            if (event.currentTarget.parent != null) {
                var menuItem:NativeMenuItem =
                        findItemForMenu(NativeMenu(event.currentTarget));
                if (menuItem != null) {
                    trace("Select event for \"" +
                            event.target.label +
                            "\" command handled by menu: " +
                            menuItem.label);
                }
            } else {
                trace("Select event for \"" +
                        event.target.label +
                        "\" command handled by root menu.");
            }
        }

        private function findItemForMenu(menu:NativeMenu):NativeMenuItem {
            for each (var item:NativeMenuItem in menu.parent.items) {
                if (item != null) {
                    if (item.submenu == menu) {
                        return item;
                    }
                }
            }
            return null;
        }
    }
}
```

Chapter 16: Taskbar icons

Many operating systems provide a taskbar, such as the Mac OS X dock, that can contain an icon to represent an application. Adobe® AIR® provides an interface for interacting with the application task bar icon through the `NativeApplication.nativeApplication.icon` property.

Contents

Quick Starts (Adobe AIR Developer Center)

- Using the system tray and dock icons

Language Reference

- DockIcon
- SystemTrayIcon

More Information

- Adobe AIR Developer Center for Flex (search for 'AIR taskbar icons')

About taskbar icons

AIR creates the `NativeApplication.nativeApplication.icon` object automatically. The object type is either DockIcon or SystemTrayIcon, depending on the operating system. You can determine which of these InteractiveIcon subclasses that AIR supports on the current operating system using the `NativeApplication.supportsDockIcon` and `NativeApplication.supportsSystemTrayIcon` properties. The InteractiveIcon base class provides the properties `width`, `height`, and `bitmaps`, which you can use to change the image used for the icon. However, accessing properties specific to DockIcon or SystemTrayIcon on the wrong operating system generates a runtime error.

To set or change the image used for an icon, create an array containing one or more images and assign it to the `NativeApplication.nativeApplication.icon.bitmaps` property. The size of taskbar icons can be different on different operating systems. To avoid image degradation due to scaling, you can add multiple sizes of images to the `bitmaps` array. If you provide more than one image, AIR selects the size closest to the current display size of the taskbar icon, scaling it only if necessary. The following example sets the image for a taskbar icon using two images:

```
NativeApplication.nativeApplication.icon.bitmaps = 
            [bmp16x16.bitmapData, bmp128x128.bitmapData];
```

To change the icon image, assign an array containing the new image or images to the `bitmaps` property. You can animate the icon by changing the image in response to an `enterFrame` or `timer` event.

To remove the icon from the notification area on Windows, or restore the default icon appearance on Mac OS X, set `bitmaps` to an empty array:

```
NativeApplication.nativeApplication.icon.bitmaps = [];
```

Dock icons

AIR supports dock icons when `NativeApplication.supportsDockIcon` is `true`. The `NativeApplication.nativeApplication.icon` property represents the application icon on the dock (not a window dock icon).

Note: *AIR does not support changing window icons on the dock under Mac OS X. Also, changes to the application dock icon only apply while an application is running — the icon reverts to its normal appearance when the application terminates.*

Dock icon menus

You can add commands to the standard dock menu by creating a NativeMenu object containing the commands and assigning it to the `NativeApplication.nativeApplication.icon.menu` property. The items in the menu are displayed above the standard dock icon menu items.

Bouncing the dock

You can bounce the dock icon by calling the `NativeApplication.nativeApplication.icon.bounce()` method. If you set the `bounce() priority` parameter to informational, then the icon bounces once. If you set it to critical, then the icon bounces until the user activates the application. Constants for the `priority` parameter are defined in the NotificationType class.

Note: *The icon does not bounce if the application is already active.*

Dock icon events

When the dock icon is clicked, the NativeApplication object dispatches an `invoke` event. If the application is not running, the system launches it. Otherwise, the `invoke` event is delivered to the running application instance.

System Tray icons

AIR supports system tray icons when `NativeApplication.supportsSystemTrayIcon` is `true`, which is currently the case only on Windows. On Windows, system tray icons are displayed in the notification area of the taskbar. No icon is displayed by default. To show an icon, assign an array containing BitmapData objects to the icon `bitmaps` property. To change the icon image, assign an array containing the new images to `bitmaps`. To remove the icon, set `bitmaps` to `null`.

System tray icon menus

You can add a menu to the system tray icon by creating a NativeMenu object and assigning it to the `NativeApplication.nativeApplication.icon.menu` property (no default menu is provided by the operating system). Access the system tray icon menu by right-clicking the icon.

System tray icon tooltips

Add a tooltip to an icon by setting the tooltip property:

```
NativeApplication.nativeApplication.icon.tooltip = "Application name";
```

System tray icon events

The SystemTrayIcon object referenced by the NativeApplication.nativeApplication.icon property dispatches a ScreenMouseEvent for `click`, `mouseDown`, `mouseUp`, `rightClick`, `rightMouseDown`, and `rightMouseUp` events. You can use these events, along with an icon menu, to allow users to interact with your application when it has no visible windows.

Example: Creating an application with no windows

The following example creates an AIR application which has a system tray icon, but no visible windows. The system tray icon has a menu with a single command for exiting the application.

```
package
{
    import flash.display.Loader;
    import flash.display.NativeMenu;
    import flash.display.NativeMenuItem;
    import flash.display.NativeWindow;
    import flash.display.Sprite;
    import flash.desktop.SystemTrayIcon;
    import flash.events.Event;
    import flash.net.URLRequest;
    import flash.desktop.NativeApplication;

    public class SysTrayApp extends Sprite
    {
        public function SysTrayApp():void{
            NativeApplication.nativeApplication.autoExit = false;
            var icon:Loader = new Loader();
            var iconMenu:NativeMenu = new NativeMenu();
            var exitCommand:NativeMenuItem = iconMenu.addItem(new NativeMenuItem("Exit"));
                exitCommand.addEventListener(Event.SELECT, function(event:Event):void {
                    NativeApplication.nativeApplication.icon.bitmaps = [];
                    NativeApplication.nativeApplication.exit();
                });

            if (NativeApplication.supportsSystemTrayIcon) {
                NativeApplication.nativeApplication.autoExit = false;
                icon.contentLoaderInfo.addEventListener(Event.COMPLETE, iconLoadComplete);
                icon.load(new URLRequest("icons/AIRApp_16.png"));

                var systray:SystemTrayIcon =
                    NativeApplication.nativeApplication.icon as SystemTrayIcon;
                systray.tooltip = "AIR application";
                systray.menu = iconMenu;
            }

            if (NativeApplication.supportsDockIcon){
                icon.contentLoaderInfo.addEventListener(Event.COMPLETE,iconLoadComplete);
                icon.load(new URLRequest("icons/AIRApp_128.png"));
                var dock:DockIcon = NativeApplication.nativeApplication.icon as DockIcon;
                dock.menu = iconMenu;
            }
    }
            stage.nativeWindow.close();
        }

        private function iconLoadComplete(event:Event):void
        {
            NativeApplication.nativeApplication.icon.bitmaps =
                [event.target.content.bitmapData];
        }
    }
}
```

***Note:** The example assumes that there are image files named `AIRApp_16.png` and `AIRApp_128.png` in an `icons` subdirectory of the application. (Sample icon files, which you can copy to your project folder, are included in the AIR SDK.)*

Window taskbar icons and buttons

Iconified representations of windows are typically displayed in the window area of a taskbar or dock to allow users to easily access background or minimized windows. The Mac OS X dock displays an icon for your application as well as an icon for each minimized window. The Microsoft Windows taskbar displays a button containing the progam icon and title for each normal-type window in your application.

Highlighting the taskbar window button

When a window is in the background, you can notify the user that an event of interest related to the window has occurred. On Mac OS X, you can notify the user by bouncing the application dock icon (as described in "Bouncing the dock" on page 142). On Windows, you can highlight the window taskbar button by calling the `notifyUser()` method of the NativeWindow instance. The `type` parameter passed to the method determines the urgency of the notification:

- `NotificationType.CRITICAL`: the window icon flashes until the user brings the window to the foreground.
- `NotificationType.INFORMATIONAL`: the window icon highlights by changing color.

The following statement highlights the taskbar button of a window:

```
stage.nativeWindow.notifyUser(NotificationType.CRITICAL);
```

Calling the `NativeWindow.notifyUser()` method on an operating system that does not support window-level notification has no effect. Use the `NativeWindow.supportsNotification` property to determine if window notification is supported.

Creating windows without taskbar buttons or icons

On the Windows operating system, windows created with the types *utility* or *lightweight* do not appear on the taskbar. Invisible windows do not appear on the taskbar, either.

Because the initial window is necessarily of type, *normal*, in order to create an application without any windows appearing in the taskbar, you must either close the initial window or leave it invisible. To close all windows in your application without terminating the application, set the `autoExit` property of the NativeApplication object to `false` before closing the last window. To simply prevent the initial window from ever becoming visible, add `<visible>false</visible>` to the `<initalWindow>` element of the application descriptor file (and do not set the `visible` property to `true` or call the `activate()` method of the window).

In new windows opened by the application, set the `type` property of the NativeWindowInitOption object passed to the window constructor to `NativeWindowType.UTILITY` or `NativeWindowType.LIGHTWEIGHT`.

On Mac OS X, windows that are minimized are displayed on the dock taskbar. You can prevent the minimized icon from being displayed by hiding the window instead of minimizing it. The following example listens for a `nativeWindowDisplayState` change event and cancels it if the window is being minimized. Instead the handler sets the window `visible` property to `false`:

```
private function preventMinimize(event:NativeWindowDisplayStateEvent):void{
    if(event.afterDisplayState == NativeWindowDisplayState.MINIMIZED){
        event.preventDefault();
        event.target.visible = false;
    }
}
```

If a window is minimized on the Mac OS X dock when you set the `visible` property to `false`, the dock icon is not removed. A user can still click the icon to make the window reappear.

Part 6: Files and data

Chapter 17: Working with the file system

You use the classes provided by the Adobe® AIR™ file system API to access the file system of the host computer. Using these classes, you can access and manage directories and files, create directories and files, write data to files, and so on. Information on understanding and using the File API classes is available in the following categories:

Contents

Quick Starts (Adobe AIR Developer Center)

- Building a text-file editor
- Building a directory search application
- Reading and writing from an XML preferences file
- Compressing files and data

Language Reference

- File
- FileStream
- FileMode

More information

- Adobe AIR Developer Center for Flex (search for 'AIR filesystem')

AIR file basics

Adobe AIR provides classes that you can use to access, create, and manage both files and folders. These classes, contained in the flash.filesystem package, are used as follows:

File classes	Description
File	File object represents a path to a file or directory. You use a file object to create a pointer to a file or folder, initiating interaction with the file or folder.
FileMode	The FileMode class defines string constants used in the `fileMode` parameter of the `open()` and `openAsync()` methods of the FileStream class. The `fileMode` parameter of these methods determines the capabilities available to the FileStream object once the file is opened, which include writing, reading, appending, and updating.
FileStream	FileStream object is used to open files for reading and writing. Once you've created a File object that points to a new or existing file, you pass that pointer to the FileStream object so that you can open and then manipulate data within the file.

Some methods in the File class have both synchronous and asynchronous versions:

- `File.copyTo()` and `File.copyToAsync()`
- `File.deleteDirectory()` and `File.deleteDirectoryAsync()`
- `File.deleteFile()` and `File.deleteFileAsync()`
- `File.getDirectoryListing()` and `File.getDirectoryListingAsync()`
- `File.moveTo()` and `File.moveToAsync()`
- `File.moveToTrash()` and `File.moveToTrashAsync()`

Also, FileStream operations work synchronously or asynchronously depending on how the FileStream object opens the file: by calling the `open()` method or by calling the `openAsync()` method.

The asynchronous versions let you initiate processes that run in the background and dispatch events when complete (or when error events occur). Other code can execute while these asynchronous background processes are taking place. With asynchronous versions of the operations, you must set up event listener functions, using the `addEventListener()` method of the File or FileStream object that calls the function.

The synchronous versions let you write simpler code that does not rely on setting up event listeners. However, since other code cannot execute while a synchronous method is executing, important processes such as display object rendering and animation may be paused.

Working with File objects

A File object is a pointer to a file or directory in the file system.

The File class extends the FileReference class. The FileReference class, which is available in Adobe® Flash® Player as well as AIR, represents a pointer to a file, but the File class adds properties and methods that are not exposed in Flash Player (in a SWF file running in a browser), due to security considerations.

Contents

About the File class

You can use the File class for the following:

- Getting the path to special directories, including the user directory, the user's documents directory, the directory from which the application was launched, and the application directory
- Coping files and directories
- Moving files and directories
- Deleting files and directories (or moving them to the trash)

- Listing files and directories contained in a directory
- Creating temporary files and folders

Once a File object points to a file path, you can use it to read and write file data, using the FileStream class.

A File object can point to the path of a file or directory that does not yet exist. You can use such a File object in creating a file or directory.

Paths of File objects

Each File object has two properties that each define its path:

Property	Description
nativePath	Specifies the platform-specific path to a file. For example, on Windows a path might be "c:\Sample directory\test.txt" whereas on Mac OS it could be "/Sample directory/test.txt". A `nativePath` property uses the backslash (\) character as the directory separator character on Windows, and it uses the forward slash (/) character on Mac OS.
url	This may use the file URL scheme to point to a file. For example, on Windows a path might be "file:///c:/Sample%20directory/test.txt" whereas on Mac OS it could be "file:///Sample%20directory/test.txt". The runtime includes other special URL schemes besides `file` and are described in "Supported URL schemes" on page 153.

The File class includes properties for pointing to standard directories on both Mac and Windows.

Pointing a File object to a directory

There are different ways to set a File object to point to a directory.

Pointing to the user's home directory

You can point a File object to the user's home directory. On Windows, the home directory is the parent of the "My Documents" directory (for example, "C:\Documents and Settings*userName*\My Documents"). On Mac OS, it is the Users/*userName* directory. The following code sets a File object to point to an AIR Test subdirectory of the home directory:

```
var file:File = File.userDirectory.resolvePath("AIR Test");
```

Pointing to the user's documents directory

You can point a File object to the user's documents directory. On Windows, this is typically the "My Documents" directory (for example, "C:\Documents and Settings*userName*\My Documents"). On Mac OS, it is the Users/*userName*/Documents directory. The following code sets a File object to point to an AIR Test subdirectory of the documents directory:

```
var file:File = File.documentsDirectory.resolvePath("AIR Test");
```

Pointing to the desktop directory

You can point a File object to the desktop. The following code sets a File object to point to an AIR Test subdirectory of the desktop:

```
var file:File = File.desktopDirectory.resolvePath("AIR Test");
```

Pointing to the application storage directory

You can point a File object to the application storage directory. For every AIR application, there is a unique associated path that defines the application storage directory. This directory is unique to each application and user. You may want to use this directory to store user-specific, application-specific data (such as user data or preferences files). For example, the following code points a File object to a preferences file, prefs.xml, contained in the application storage directory:

```
var file:File = File.applicationStorageDirectory;
file = file.resolvePath("prefs.xml");
```

The application storage directory location is based on the user name, the application ID, and the publisher ID:

- On Mac OS—In:

  `/Users/`*user name*`/Library/Preferences/`*applicationID*`.`*publisherID*`/Local Store/`

  For example:

  ```
  /Users/babbage/Library/Preferences/com.example.TestApp.02D88EEED35F84C264A183921344EEA3
  53A629FD.1/Local Store
  ```

- On Windows—In the documents and Settings directory, in:

  user name`/Application Data/`*applicationID*`.`*publisherID*`/Local Store/`

  For example:

  ```
  C:\Documents and Settings\babbage\Application
  Data\com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store
  ```

The URL (and `url` property) for a File object created with `File.applicationStorageDirectory` uses the `app-storage` URL scheme (see "Supported URL schemes" on page 153), as in the following:

```
var dir:File = File.applicationStorageDirectory;
dir = dir.resolvePath("preferences");
trace(dir.url); // app-storage:/preferences
```

Pointing to the application directory

You can point a File object to the directory in which the application was installed, known as the application directory. You can reference this directory using the `File.applicationDirectory` property. You may use this directory to examine the application descriptor file or other resources installed with the application. For example, the following code points a File object to a directory named *images* in the application directory:

```
var dir:File = File.applicationDirectory;
dir = dir.resolvePath("images");
```

The URL (and `url` property) for a File object created with `File.applicationDirectory` uses the `app` URL scheme (see "Supported URL schemes" on page 153), as in the following:

```
var dir:File = File.applicationDirectory;
dir = dir.resolvePath("images");
trace(dir.url); // app:/images
```

Pointing to the filesystem root

The `File.getRootDirectories()` method lists all root volumes, such as C: and mounted volumes, on a Windows computer. On Mac, this method always returns the unique root directory for the machine (the "/" directory).

Pointing to an explicit directory

You can point the File object to an explicit directory by setting the `nativePath` property of the File object, as in the following example (on Windows):

```
var file:File = new File();
file.nativePath = "C:\\AIR Test\\";
```

Navigating to relative paths

You can use the `resolvePath()` method to obtain a path relative to another given path. For example, the following code sets a File object to point to an "AIR Test" subdirectory of the user's home directory:

```
var file:File = File.userDirectory;
file = file.resolvePath("AIR Test");
```

You can also use the `url` property of a File object to point it to a directory based on a URL string, as in the following:

```
var urlStr:String = "file:///C:/AIR Test/";
var file:File = new File()
file.url = urlStr;
```

For more information, see "Modifying File paths" on page 153.

Letting the user browse to select a directory

The File class includes the `browseForDirectory()` method, which presents a system dialog box in which the user can select a directory to assign to the object. The `browseForDirectory()` method is asynchronous. It dispatches a `select` event if the user selects a directory and clicks the Open button, or it dispatches a `cancel` event if the user clicks the Cancel button.

For example, the following code lets the user select a directory and outputs the directory path upon selection:

```
var file:File = new File();
file.addEventListener(Event.SELECT, dirSelected);
file.browseForDirectory("Select a directory");
function dirSelected(e:Event):void {
    trace(file.nativePath);
}
```

Pointing to the directory from which the application was invoked

You can get the directory location from which an application is invoked, by checking the `currentDirectory` property of the InvokeEvent object dispatched when the application is invoked. For details, see "Capturing command line arguments" on page 309.

Pointing a File object to a file

There are different ways to set the file to which a File object points.

Pointing to an explicit file path

You can use the `resolvePath()` method to obtain a path relative to another given path. For example, the following code sets a File object to point to a log.txt file within the application storage directory:

```
var file:File = File.applicationStorageDirectory;
file = file.resolvePath("log.txt");
```

You can use the `url` property of a File object to point it to a file or directory based on a URL string, as in the following:

```
var urlStr:String = "file:///C:/AIR Test/test.txt";
var file:File = new File()
file.url = urlStr;
```

You can also pass the URL to the `File()` constructor function, as in the following:

```
var urlStr:String = "file:///C:/AIR Test/test.txt";
var file:File = new File(urlStr);
```

The `url` property always returns the URI-encoded version of the URL (for example, blank spaces are replaced with `"%20`):

```
file.url = "file:///c:/AIR Test";
trace(file.url); // file:///c:/AIR%20Test
```

You can also use the `nativePath` property of a File object to set an explicit path. For example, the following code, when run on a Windows computer, sets a File object to the test.txt file in the AIR Test subdirectory of the C: drive:

```
var file:File = new File();
file.nativePath = "C:/AIR Test/test.txt";
```

You can also pass this path to the `File()` constructor function, as in the following:

```
var file:File = new File("C:/AIR Test/test.txt");
```

On Windows, you can use the forward slash (/) or backslash (\) character as the path delimiter for the `nativePath` property. On Mac OS, use the forward slash (/) character as the path delimiter for the `nativePath`:

```
var file:File = new File(/Users/dijkstra/AIR Test/test.txt");
```

For more information, see “Modifying File paths” on page 153.

Enumerating files in a directory

You can use the `getDirectoryListing()` method of a File object to get an array of File objects pointing to files and subdirectories at the root level of a directory. For more information, see “Enumerating directories” on page 157.

Letting the user browse to select a file

The File class includes the following methods that present a system dialog box in which the user can select a file to assign to the object:

- `browseForOpen()`
- `browseForSave()`
- `browseForOpenMultiple()`

These methods are each asynchronous. The `browseForOpen()` and `browseForSave()` methods dispatch the select event when the user selects a file (or a target path, in the case of browseForSave()). With the `browseForOpen()` and `browseForSave()` methods, upon selection the target File object points to the selected files. The `browseForOpenMultiple`() method dispatches a `selectMultiple` event when the user selects files. The `selectMultiple` event is of type FileListEvent, which has a `files` property that is an array of File objects (pointing to the selected files).

For example, the following code presents the user with an “Open” dialog box in which the user can select a file:

```
var fileToOpen:File = File.documentsDirectory;
selectTextFile(fileToOpen);

function selectTextFile(root:File):void
{
    var txtFilter:FileFilter = new FileFilter("Text", "*.as;*.css;*.html;*.txt;*.xml");
    root.browseForOpen("Open", [txtFilter]);
    root.addEventListener(Event.SELECT, fileSelected);
}

function fileSelected(event:Event):void
{
    trace(fileToOpen.nativePath);
}
```

If the application has another browser dialog box open when you call a browse method, the runtime throws an Error exception.

Modifying File paths

You can also modify the path of an existing File object by calling the `resolvePath()` method or by modifying the `nativePath` or `url` property of the object, as in the following examples (on Windows):

```
var file1:File = File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
trace(file1.nativePath); // C:\Documents and Settings\userName\My Documents\AIR Test
var file2:File = File.documentsDirectory;
file2 = file2.resolvePath("..");
trace(file2.nativePath); // C:\Documents and Settings\userName
var file3:File = File.documentsDirectory;
file3.nativePath += "/subdirectory";
trace(file3.nativePath); // C:\Documents and Settings\userName\My Documents\subdirectory
var file4:File = new File();
file.url = "file:///c:/AIR Test/test.txt"
trace(file3.nativePath); // C:\AIR Test\test.txt
```

When using the `nativePath` property, you use either the forward slash (/) or backslash (\) character as the directory separator character on Windows; use the forward slash (/) character on Mac OS. On Windows, remember to type the backslash character twice in a string literal.

Supported URL schemes

You can use any of the following URL schemes in defining the `url` property of a File object:

URL scheme	Description
file	Use to specify a path relative to the root of the file system. For example: `file:///c:/AIR Test/test.txt` The URL standard specifies that a file URL takes the form `file://<host>/<path>`. As a special case, `<host>` can be the empty string, which is interpreted as "the machine from which the URL is being interpreted." For this reason, file URLs often have three slashes (///).
app	Use to specify a path relative to the root directory of the installed application (the directory that contains the application.xml file for the installed application). For example, the following path points to an images subdirectory of the directory of the installed application: `app:/images`
app-storage	Use to specify a path relative to the application store directory. For each installed application, AIR defines a unique application store directory, which is a useful place to store data specific to that application. For example, the following path points to a prefs.xml file in a settings subdirectory of the application store directory: `app-storage:/settings/prefs.xml`

Finding the relative path between two files

You can use the `getRelativePath()` method to find the relative path between two files:

```
var file1:File = File.documentsDirectory.resolvePath("AIR Test");
var file2:File = File.documentsDirectory
file2 = file2.resolvePath("AIR Test/bob/test.txt");

trace(file1.getRelativePath(file2)); // bob/test.txt
```

The second parameter of the `getRelativePath()` method, the `useDotDot` parameter, allows for `..` syntax to be returned in results, to indicate parent directories:

```
var file1:File = File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
var file2:File = File.documentsDirectory;
file2 = file2.resolvePath("AIR Test/bob/test.txt");
var file3:File = File.documentsDirectory;
file3 = file3.resolvePath("AIR Test/susan/test.txt");

trace(file2.getRelativePath(file1, true)); // ../..
trace(file3.getRelativePath(file2, true)); // ../../bob/test.txt
```

Obtaining canonical versions of file names

File and path names are usually not case sensitive. In the following, two File objects point to the same file:

```
File.documentsDirectory.resolvePath("test.txt");
File.documentsDirectory.resolvePath("TeSt.TxT");
```

However, documents and directory names do include capitalization. For example, the following assumes that there is a folder named AIR Test in the documents directory, as in the following examples:

```
var file:File = File.documentsDirectory.resolvePath("AIR test");
trace(file.nativePath); // ... AIR test
file.canonicalize();
trace(file.nativePath); // ... AIR Test
```

The `canonicalize` method converts the `nativePath` object to use the correct capitalization for the file or directory name.

You can also use the `canonicalize()` method to convert short file names ("8.3" names) to long file names on Windows, as in the following examples:

```
var path:File = new File();
path.nativePath = "C:\\AIR~1";
path.canonicalize();
trace(path.nativePath); // C:\AIR Test
```

Working with packages and symbolic links

Various operating systems support package files and symbolic link files:

Packages On Mac OS, directories can be designated as packages and show up in the Mac OS Finder as a single file rather than as a directory.

Symbolic links Symbolic links allow a file to point to another file or directory on disk. Although similar, symbolic links are not the same as aliases. An alias is always reported as a file (rather than a directory), and reading or writing to an alias or shortcut never affects the original file or directory that it points to. On the other hand, a symbolic link behaves exactly like the file or directory it points to. It can be reported as a file or a directory, and reading or writing to a symbolic link affects the file or directory that it points to, not the symbolic link itself.

The File class includes the `isPackage` and `isSymbolicLink` properties for checking if a File object references a package or symbolic link.

The following code iterates through the user's desktop directory, listing subdirectories that are *not* packages:

```
var desktopNodes:File = File.desktopDirectory.getDirectoryListing();
for (var i:uint = 0; i < desktopNodes.length; i++)
{
    if (desktopNodes[i].isDirectory && !!desktopNodes[i].isPackage)
    {
        trace(desktopNodes[i].name);
    }
}
```

The following code iterates through the user's desktop directory, listing files and directories that are *not* symbolic links:

```
var desktopNodes:File = File.desktopDirectory.getDirectoryListing();
for (var i:uint = 0; i < desktopNodes.length; i++)
{
    if (!desktopNodes[i].isSymbolicLink)
    {
        trace(desktopNodes[i].name);
    }
}
```

The `canonicalize()` method changes the path of a symbolic link to point to the file or directory to which the link refers. The following code iterates through the user's desktop directory, and reports the paths referenced by files that are symbolic links:

```
var desktopNodes:File = File.desktopDirectory.getDirectoryListing();
for (var i:uint = 0; i < desktopNodes.length; i++)
{
    if (desktopNodes[i].isSymbolicLink)
    {
        var linkNode:File = desktopNodes[i] as File;
        linkNode.canonicalize();
        trace(linkNode.nativePath);
    }
}
```

Getting file system information

The File class includes the following static properties that provide some useful information about the file system:

Property	Description
File.lineEnding	The line-ending character sequence used by the host operating system. On Mac OS, this is the line-feed character. On Windows, this is the carriage return character followed by the line-feed character.
File.separator	The host operating system's path component separator character. On Mac OS, this is the forward slash (/) character. On Windows, it is the backslash (\) character.
File.systemCharset	The default encoding used for files by the host operating system. This pertains to the character set used by the operating system, corresponding to its language.

The `Capabilities` class also includes useful system information that may be useful when working with files:

Property	Description
Capabilities.hasIME	Specifies whether the player is running on a system that does (`true`) or does not (`false`) have an input method editor (IME) installed.
Capabilities.language	Specifies the language code of the system on which the player is running.
Capabilities.os	Specifies the current operating system.

Working with directories

The runtime provides you with capabilities to work with directories on the local file system.

For details on creating File objects that point to directories, see "Pointing a File object to a directory" on page 149.

Contents

Creating directories

The `File.createDirectory()` method lets you create a directory. For example, the following code creates a directory named AIR Test as a subdirectory of the user's home directory:

```
var dir:File = File.userDirectory.resolvePath("AIR Test");
dir.createDirectory();
```

If the directory exists, the `createDirectory()` method does nothing.

Also, in some modes, a FileStream object creates directories when opening files. Missing directories are created when you instantiate a FileStream instance with the `fileMode` parameter of the `FileStream()` constructor set to `FileMode.APPEND` or `FileMode.WRITE`. For more information, see "Workflow for reading and writing files" on page 161.

Creating a temporary directory

The File class includes a `createTempDirectory()` method, which creates a directory in the temporary directory folder for the System, as in the following example:

```
var temp:File = File.createTempDirectory();
```

The `createTempDirectory()` method automatically creates a unique temporary directory (saving you the work of determining a new unique location).

You may use a temporary directory to temporarily store temporary files used for a session of the application. Note that there is a `createTempFile()` method for creating new, unique temporary files in the System temporary directory.

You may want to delete the temporary directory before closing the application, as it is *not* automatically deleted.

Enumerating directories

You can use the `getDirectoryListing()` method or the `getDirectoryListingAsync()` method of a File object to get an array of File objects pointing to files and subfolders in a directory.

For example, the following code lists the contents of the user's documents directory (without examining subdirectories):

```
var directory:File = File.documentsDirectory;
var contents:Array = directory.getDirectoryListing();
for (var i:uint = 0; i < contents.length; i++)
{
    trace(contents[i].name, contents[i].size);
}
```

When using the asynchronous version of the method, the `directoryListing` event object has a `files` property that is the array of File objects pertaining to the directories:

```
var directory:File = File.documentsDirectory;
directory.getDirectoryListingAsync();
directory.addEventListener(FileListEvent.DIRECTORY_LISTING, dirListHandler);

function dirListHandler(event:FileListEvent):void
{
    var contents:Array = event.files;
    for (var i:uint = 0; i < contents.length; i++)
    {
        trace(contents[i].name, contents[i].size);
    }
}
```

Copying and moving directories

You can copy or move a directory, using the same methods as you would to copy or move a file. For example, the following code copies a directory synchronously:

```
var sourceDir:File = File.documentsDirectory.resolvePath("AIR Test");
var resultDir:File = File.documentsDirectory.resolvePath("AIR Test Copy");
sourceDir.copyTo(resultDir);
```

When you specify true for the `overwrite` parameter of the `copyTo()` method, all files and folders in an existing target directory are deleted and replaced with the files and folders in the source directory (even if the target file does not exist in the source directory).

The directory that you specify as the `newLocation` parameter of the `copyTo()` method specifies the path to the resulting directory; it does *not* specify the *parent* directory that will contain the resulting directory.

For details, see “Copying and moving files” on page 159.

Deleting directory contents

The File class includes a `deleteDirectory()` method and a `deleteDirectoryAsync()` method. These methods delete directories, the first working synchronously, the second working asynchronously (see “AIR file basics” on page 147). Both methods include a `deleteDirectoryContents` parameter (which takes a Boolean value); when this parameter is set to `true` (the default value is `false`) the call to the method deletes non-empty directories; otherwise, only empty directories are deleted.

For example, the following code synchronously deletes the AIR Test subdirectory of the user's documents directory:

```
var directory:File = File.documentsDirectory.resolvePath("AIR Test");
```

```
directory.deleteDirectory(true);
```

The following code asynchronously deletes the AIR Test subdirectory of the user's documents directory:

```
var directory:File = File.documentsDirectory.resolvePath("AIR Test");
directory.addEventListener(Event.COMPLETE, completeHandler)
directory.deleteDirectoryAsync(true);

function completeHandler(event:Event):void {
    trace("Deleted.")
}
```

Also included are the `moveToTrash()` and `moveToTrashAsync()` methods, which you can use to move a directory to the System trash. For details, see "Moving a file to the trash" on page 160.

Working with files

Using the AIR file API, you can add basic file interaction capabilities to your applications. For example, you can read and write files, copy and delete files, and so on. Since your applications can access the local file system, refer to "AIR security" on page 69, if you haven't already done so.

***Note:** You can associate a file type with an AIR application (so that double-clicking it opens the application). For details, see "Managing file associations" on page 317.*

Contents

Getting file information

The File class includes the following properties that provide information about a file or directory to which a File object points:

File property	Description
creationDate	The creation date of the file on the local disk.
creator	Obsolete—use the `extension` property. (This property reports the Macintosh creator type of the file, which is only used in Mac OS versions prior to Mac OS X.)
exists	Whether the referenced file or directory exists.
extension	The file extension, which is the part of the name following (and not including) the final dot ("."). If there is no dot in the filename, the extension is `null`.
icon	An Icon object containing the icons defined for the file.
isDirectory	Whether the File object reference is to a directory.
modificationDate	The date that the file or directory on the local disk was last modified.
name	The name of the file or directory (including the file extension, if there is one) on the local disk.

File property	Description
nativePath	The full path in the host operating system representation. See "Paths of File objects" on page 149.
parent	The folder that contains the folder or file represented by the File object. This property is `null` if the File object references a file or directory in the root of the filesystem.
size	The size of the file on the local disk in bytes.
type	Obsolete—use the `extension` property. (On the Macintosh, this property is the four-character file type, which is only used in Mac OS versions prior to Mac OS X.)
url	The URL for the file or directory. See "Paths of File objects" on page 149.

For details on these properties, see the File class entry in the *Flex 3 Language Reference (http://www.adobe.com/go/learn_flex3_aslr).*

Copying and moving files

The File class includes two methods for copying files or directories: `copyTo()` and `copyToAsync()`. The File class includes two methods for moving files or directories: `moveTo()` and `moveToAsync()`. The `copyTo()` and `moveTo()` methods work synchronously, and the `copyToAsync()` and `moveToAsync()` methods work asynchronously (see "AIR file basics" on page 147).

To copy or move a file, you set up two File objects. One points to the file to copy or move, and it is the object that calls the copy or move method; the other points to the destination (result) path.

The following copies a test.txt file from the AIR Test subdirectory of the user's documents directory to a file named copy.txt in the same directory:

```
var original:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var newFile:File = File.resolvePath("AIR Test/copy.txt");
original.copyTo(newFile, true);
```

In this example, the value of `overwrite` parameter of the `copyTo()` method (the second parameter) is set to `true`. By setting this to `true`, an existing target file is overwritten. This parameter is optional. If you set it to `false` (the default value), the operation dispatches an IOErrorEvent event if the target file exists (and the file is not copied).

The "Async" versions of the copy and move methods work asynchronously. Use the `addEventListener()` method to monitor completion of the task or error conditions, as in the following code:

```
var original = File.documentsDirectory;
original = original.resolvePath("AIR Test/test.txt");

var destination:File = File.documentsDirectory;
destination =  destination.resolvePath("AIR Test 2/copy.txt");

original.addEventListener(Event.COMPLETE, fileMoveCompleteHandler);
original.addEventListener(IOErrorEvent.IO_ERROR, fileMoveIOErrorEventHandler);
original.moveToAsync(destination);

function fileMoveCompleteHandler(event:Event):void {
    trace(event.target); // [object File]
}
function fileMoveIOErrorEventHandler(event:IOErrorEvent):void {
    trace("I/O Error.");
}
```

The File class also includes the `File.moveToTrash()` and `File.moveToTrashAsync()` methods, which move a file or directory to the system trash.

Deleting a file

The File class includes a `deleteFile()` method and a `deleteFileAsync()` method. These methods delete files, the first working synchronously, the second working asynchronously (see "AIR file basics" on page 147).

For example, the following code synchronously deletes the test.txt file in the user's documents directory:

```
var file:File = File.documentsDirectory.resolvePath("test.txt");
file.deleteFile();
```

The following code asynchronously deletes the test.txt file of the user's documents directory:

```
var file:File = File.documentsDirectory.resolvePath("test.txt");
file.addEventListener(Event.COMPLETE, completeHandler)
file.deleteFileAsync();

function completeHandler(event:Event):void {
    trace("Deleted.")
}
```

Also included are the `moveToTrash()` and `moveToTrashAsync` methods, which you can use to move a file or directory to the System trash. For details, see "Moving a file to the trash" on page 160.

Moving a file to the trash

The File class includes a `moveToTrash()` method and a `moveToTrashAsync()` method. These methods send a file or directory to the System trash, the first working synchronously, the second working asynchronously (see "AIR file basics" on page 147).

For example, the following code synchronously moves the test.txt file in the user's documents directory to the System trash:

```
var file:File = File.documentsDirectory.resolvePath("test.txt");
file.moveToTrash();
```

Creating a temporary file

The File class includes a `createTempFile()` method, which creates a file in the temporary directory folder for the System, as in the following example:

```
var temp:File = File.createTempFile();
```

The `createTempFile()` method automatically creates a unique temporary file (saving you the work of determining a new unique location).

You may use a temporary file to temporarily store information used in a session of the application. Note that there is also a `createTempDirectory()` method, for creating a unique temporary directory in the System temporary directory.

You may want to delete the temporary file before closing the application, as it is *not* automatically deleted.

Reading and writing files

The FileStream class lets AIR applications read and write to the file system.

Contents

Workflow for reading and writing files

The workflow for reading and writing files is as follows.

1. Initialize a File object that points to the path.

This is the path of the file that you want to work with (or a file that you will later create).

```
var file:File = File.documentsDirectory;
file = file.resolvePath("AIR Test/testFile.txt");
```

This example uses the `File.documentsDirectory` property and the `resolvePath()` method of a File object to initialize the File object. However, there are many other ways to point a File object to a file. For more information, see “Pointing a File object to a file” on page 151.

2. Initialize a FileStream object.

3. Call the `open()` method or the `openAsync()` method of the FileStream object.

The method you call depends on whether you want to open the file for synchronous or asynchronous operations. Use the File object as the `file` parameter of the open method. For the `fileMode` parameter, specify a constant from the FileMode class that specifies the way in which you will use the file.

For example, the following code initializes a FileStream object that is used to create a file and overwrite any existing data:

```
var fileStream:FileStream = new FileStream();
fileStream.open(file, FileMode.WRITE);
```

For more information, see “Initializing a FileStream object, and opening and closing files” on page 163 and “FileStream open modes” on page 162.

4. If you opened the file asynchronously (using the `openAsync()` method), add and set up event listeners for the FileStream object.

These event listener methods respond to events dispatched by the FileStream object in a variety of situations, such as when data is read in from the file, when I/O errors are encountered, or when the complete amount of data to be written has been written.

For details, see “Asynchronous programming and the events generated by a FileStream object opened asynchronously” on page 166.

5. Include code for reading and writing data, as needed.

There are many methods of the FileStream class related to reading and writing. (They each begin with "read" or "write".) The method you choose to use to read or write data depends on the format of the data in the target file.

For example, if the data in the target file is UTF-encoded text, you may use the `readUTFBytes()` and `writeUTFBytes()` methods. If you want to deal with the data as byte arrays, you may use the `readByte()`, `readBytes()`, `writeByte()`, and `writeBytes()` methods. For details, see "Data formats, and choosing the read and write methods to use" on page 167.

If you opened the file asynchronously, then be sure that enough data is available before calling a read method. For details, see "The read buffer and the bytesAvailable property of a FileStream object" on page 165.

6. Call the `close()` method of the FileStream object when you are done working with the file.

This makes the file available to other applications.

For details, see "Initializing a FileStream object, and opening and closing files" on page 163.

To see a sample application that uses the FileStream class to read and write files, see the following articles at the Adobe AIR Developer Center:

- Building a text-file editor
- Reading and writing from an XML preferences file

Working with FileStream objects

The FileStream class defines methods for opening, reading, and writing files.

Contents

FileStream open modes

The `open()` and `openAsync()` methods of a FileStream object each include a `fileMode` parameter, which defines some properties for a file stream, including the following:

- The ability to read from the file
- The ability to write to the file
- Whether data will always be appended past the end of the file (when writing)
- What to do when the file does not exist (and when its parent directories do not exist)

The following are the various file modes (which you can specify as the `fileMode` parameter of the `open()` and `openAsync()` methods):

File mode	Description
FileMode.READ	Specifies that the file is open for reading only.
FileMode.WRITE	Specifies that the file is open for writing. If the file does not exist, it is created when the FileStream object is opened. If the file does exist, any existing data is deleted.

File mode	Description
FileMode.APPEND	Specifies that the file is open for appending. The file is created if it does not exist. If the file exists, existing data is not overwritten, and all writing begins at the end of the file.
FileMode.UPDATE	Specifies that the file is open for reading and writing. If the file does not exist, it is created. Specify this mode for random read/write access to the file. You can read from any position in the file, and when writing to the file, only the bytes written overwrite existing bytes (all other bytes remain unchanged).

Initializing a FileStream object, and opening and closing files

When you open a FileStream object, you make it available to read and write data to a file. You open a FileStream object by passing a File object to the `open()` or `openAsync()` method of the FileStream object:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, FileMode.READ);
```

The `fileMode` parameter (the second parameter of the `open()` and `openAsync()` methods), specifies the mode in which to open the file: for read, write, append, or update. For details, see the previous section, "FileStream open modes" on page 162.

If you use the `openAsync()` method to open the file for asynchronous file operations, set up event listeners to handle the asynchronous events:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completeHandler);
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(IOErrorEvent.IOError, errorHandler);
myFileStream.open(myFile, FileMode.READ);

function completeHandler(event:Event):void {
    // ...
}

function progressHandler(event:ProgressEvent):void {
    // ...
}

function errorHandler(event:IOErrorEvent):void {
    // ...
}
```

The file is opened for synchronous or asynchronous operations, depending upon whether you use the `open()` or `openAsync()` method. For details, see "AIR file basics" on page 147.

If you set the `fileMode` parameter to `FileMode.READ` or `FileMode.UPDATE` in the open method of the FileStream object, data is read into the read buffer as soon as you open the FileStream object. For details, see "The read buffer and the bytesAvailable property of a FileStream object" on page 165.

You can call the `close()` method of a FileStream object to close the associated file, making it available for use by other applications.

The position property of a FileStream object

The `position` property of a FileStream object determines where data is read or written on the next read or write method.

Before a read or write operation, set the `position` property to any valid position in the file.

For example, the following code writes the string `"hello"` (in UTF encoding) at position 8 in the file:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, FileMode.UPDATE);
myFileStream.position = 8;
myFileStream.writeUTFBytes("hello");
```

When you first open a FileStream object, the `position` property is set to 0.

Before a read operation, the value of `position` must be at least 0 and less than the number of bytes in the file (which are existing positions in the file).

The value of the `position` property is modified only in the following conditions:

- When you explicitly set the `position` property.
- When you call a read method.
- When you call a write method.

When you call a read or write method of a FileStream object, the `position` property is immediately incremented by the number of bytes that you read or write. Depending on the read method you use, the `position` property is either incremented by the number of bytes you specify to read or by the number of bytes available. When you call a read or write method subsequently, it reads or writes starting at the new position.

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, FileMode.UPDATE);
myFileStream.position = 4000;
trace(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
trace(myFileStream.position); // 4200
```

There is, however, one exception: for a FileStream opened in append mode, the `position` property is not changed after a call to a write method. (In append mode, data is always written to the end of the file, independent of the value of the `position` property.)

For a file opened for asynchronous operations, the write operation does not complete before the next line of code is executed. However, you can call multiple asynchronous methods sequentially, and the runtime executes them in order:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.openAsync(myFile, FileMode.WRITE);
myFileStream.writeUTFBytes("hello");
myFileStream.writeUTFBytes("world");
myFileStream.addEventListener(Event.CLOSE, closeHandler);
myFileStream.close();
trace("started.");

closeHandler(event:Event):void
{
    trace("finished.");
}
```

The trace output for this code is the following:

```
started.
finished.
```

You *can* specify the `position` value immediately after you call a read or write method (or at any time), and the next read or write operation will take place starting at that position. For example, note that the following code sets the `position` property right after a call to the `writeBytes()` operation, and the `position` is set to that value (300) even after the write operation completes:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.openAsync(myFile, FileMode.UPDATE);
myFileStream.position = 4000;
trace(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
myFileStream.position = 300;
trace(myFileStream.position); // 300
```

The read buffer and the bytesAvailable property of a FileStream object

When a FileStream object with read capabilities (one in which the `fileMode` parameter of the `open()` or `openAsync()` method was set to `READ` or `UPDATE`) is opened, the runtime stores the data in an internal buffer. The FileStream object begins reading data into the buffer as soon as you open the file (by calling the `open()` or `openAsync()` method of the FileStream object).

For a file opened for synchronous operations (using the `open()` method), you can always set the `position` pointer to any valid position (within the bounds of the file) and begin reading any amount of data (within the bounds of the file), as shown in the following code (which assumes that the file contains at least 100 bytes):

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, FileMode.READ);
myFileStream.position = 10;
myFileStream.readBytes(myByteArray, 0, 20);
myFileStream.position = 89;
myFileStream.readBytes(myByteArray, 0, 10);
```

Whether a file is opened for synchronous or asynchronous operations, the read methods always read from the "available" bytes, represented by the `bytesAvalable` property. When reading synchronously, all of the bytes of the file are available all of the time. When reading asynchronously, the bytes become available starting at the position specified by the `position` property, in a series of asynchronous buffer fills signaled by `progress` events.

For files opened for *synchronous* operations, the `bytesAvailable` property is always set to represent the number of bytes from the `position` property to the end of the file (all bytes in the file are always available for reading).

For files opened for *asynchronous* operations, you need to ensure that the read buffer has consumed enough data before calling a read method. For a file opened asynchronously, as the read operation progresses, the data from the file, starting at the `position` specified when the read operation started, is added to the buffer, and the `bytesAvailable` property increments with each byte read. The `bytesAvailable` property indicates the number of bytes available starting with the byte at the position specified by the `position` property to the end of the buffer. Periodically, the FileStream object sends a `progress` event.

For a file opened asynchronously, as data becomes available in the read buffer, the FileStream object periodically dispatches the `progress` event. For example, the following code reads data into a ByteArray object, `bytes`, as it is read into the buffer:

```
var bytes:ByteArray = new ByteArray();
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, FileMode.READ);

function progressHandler(event:ProgressEvent):void
```

```
{
    myFileStream.readBytes(bytes, myFileStream.position, myFileStream.bytesAvailable);
}
```

For a file opened asynchronously, only the data in the read buffer can be read. Furthermore, as you read the data, it is removed from the read buffer. For read operations, you need to ensure that the data exists in the read buffer before calling the read operation. For example, the following code reads 8000 bytes of data starting from position 4000 in the file:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);
myFileStream.position = 4000;

var str:String = "";

function progressHandler(event:Event):void
{
    if (myFileStream.bytesAvailable > 8000 )
    {
        str += myFileStream.readMultiByte(8000, "iso-8859-1");
    }
}
```

During a write operation, the FileStream object does not read data into the read buffer. When a write operation completes (all data in the write buffer is written to the file), the FileStream object starts a new read buffer (assuming that the associated FileStream object was opened with read capabilities), and starts reading data into the read buffer, starting from the position specified by the `position` property. The `position` property may be the position of the last byte written, or it may be a different position, if the user specifies a different value for the `position` object after the write operation.

Asynchronous programming and the events generated by a FileStream object opened asynchronously

When a file is opened asynchronously (using the `openAsync()` method), reading and writing files are done asynchronously. As data is read into the read buffer and as output data is being written, other ActionScript code can execute.

This means that you need to register for events generated by the FileStream object opened asynchronously.

By registering for the `progress` event, you can be notified as new data becomes available for reading, as in the following code:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, FileMode.READ);
var str:String = "";

function progressHandler(event:ProgressEvent):void
{
    str += myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

You can read the entire data by registering for the `complete` event, as in the following code:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);
var str:String = "";
function completeHandler(event:Event):void
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

In much the same way that input data is buffered to enable asynchronous reading, data that you write on an asynchronous stream is buffered and written to the file asynchronously. As data is written to a file, the FileStream object periodically dispatches an `OutputProgressEvent` object. An `OutputProgressEvent` object includes a `bytesPending` property that is set to the number of bytes remaining to be written. You can register for the `outputProgress` event to be notified as this buffer is actually written to the file, perhaps in order to display a progress dialog. However, in general, it is not necessary to do so. In particular, you may call the `close()` method without concern for the unwritten bytes. The FileStream object will continue writing data and the `close` event will be delivered after the final byte is written to the file and the underlying file is closed.

Data formats, and choosing the read and write methods to use

Every file is a set of bytes on a disk. In ActionScript, the data from a file can always be represented as a ByteArray. For example, the following code reads the data from a file into a ByteArray object named `bytes`:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);
var bytes:ByteArray = new ByteArray();

function completeHandler(event:Event):void
{
    myFileStream.readBytes(bytes, 0, myFileStream.bytesAvailable);
}
```

Similarly, the following code writes data from a ByteArray named `bytes` to a file:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.open(myFile, FileMode.WRITE);
myFileStream.writeBytes(bytes, 0, bytes.length);
```

However, often you do not want to store the data in an ActionScript ByteArray object. And often the data file is in a specified file format.

For example, the data in the file may be in a text file format, and you may want to represent such data in a String object.

For this reason, the FileStream class includes read and write methods for reading and writing data to and from types other than ByteArray objects. For example, the `readMultiByte()` method lets you read data from a file and store it to a string, as in the following code:

```
var myFile:File = File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream:FileStream = new FileStream();
myFileStream.addEventListener(Event.COMPLETE, completed);
myFileStream.openAsync(myFile, FileMode.READ);
var str:String = "";
```

```
function completeHandler(event:Event):void
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

The second parameter of the `readMultiByte()` method specifies the text format that ActionScript uses to interpret the data ("iso-8859-1" in the example). ActionScript supports common character set encodings, and these are listed in the ActionScript 3.0 Language Reference (see Supported character sets at http://livedocs.macromedia.com/flex/2/langref/charset-codes.html).

The FileStream class also includes the `readUTFBytes()` method, which reads data from the read buffer into a string using the UTF-8 character set. Since characters in the UTF-8 character set are of variable length, do not use `readUTFBytes()` in a method that responds to the `progress` event, since the data at the end of the read buffer may represent an incomplete character. (This is also true when using the `readMultiByte()` method with a variable-length character encoding.) For this reason, read the entire set of data when the FileStream object dispatches the `complete` event.

There are also similar write methods, `writeMultiByte()` and `writeUTFBytes()`, for working with String objects and text files.

The `readUTF()` and the `writeUTF()` methods (not to be confused with `readUTFBytes()` and `writeUTFBytes()`) also read and write the text data to a file, but they assume that the text data is preceded by data specifying the length of the text data, which is not a common practice in standard text files.

Some UTF-encoded text files begin with a "UTF-BOM" (byte order mark) character that defines the endianness as well as the encoding format (such as UTF-16 or UTF-32).

For an example of reading and writing to a text file, see “Example: Reading an XML file into an XML object” on page 168.

The `readObject()` and `writeObject()` are convenient ways to store and retrieve data for complex ActionScript objects. The data is encoded in AMF (ActionScript Message Format). This format is proprietary to ActionScript. Applications other than AIR, Flash Player, Flash Media Server, and Flex Data Services do not have built-in APIs for working with data in this format.

There are some other read and write methods (such as `readDouble()` and `writeDouble()`). However, if you use these, make sure that the file format matches the formats of the data defined by these methods.

File formats are often more complex than simple text formats. For example, an MP3 file includes compressed data that can only be interpreted with the decompression and decoding algorithms specific to MP3 files. MP3 files also may include ID3 tags that contain metatag information about the file (such as the title and artist for a song). There are multiple versions of the ID3 format, but the simplest (ID3 version 1) is discussed in the “Example: Reading and writing data with random access” on page 169 section.

Other files formats (for images, databases, application documents, and so on) have different structures, and to work with their data in ActionScript, you must understand how the data is structured.

Example: Reading an XML file into an XML object

The following examples demonstrate how to read and write to a text file that contains XML data.

To read from the file, initialize the File and FileStream objects, call the `readUTFBytes()` method of the FileStream and convert the string to an XML object:

```
var file:File = File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
var fileStream:FileStream = new FileStream();
fileStream.open(file, FileMode.READ);
var prefsXML:XML = XML(fileStream.readUTFBytes(fileStream.bytesAvailable));
fileStream.close();
```

Similarly, writing the data to the file is as easy as setting up appropriate File and FileStream objects, and then calling a write method of the FileStream object. Pass the string version of the XML data to the write method as in the following code:

```
var prefsXML:XML = <prefs><autoSave>true</autoSave></prefs>;
var file:File = File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
fileStream = new FileStream();
fileStream.open(file, FileMode.WRITE);

var outputString:String = '<?xml version="1.0" encoding="utf-8"?>\n';
outputString += prefsXML.toXMLString();

fileStream.writeUTFBytes(outputString);
fileStream.close();
```

These examples use the `readUTFBytes()` and `writeUTFBytes()` methods, because they assume that the files are in UTF-8 format. If not, you may need to use a different method (see “Data formats, and choosing the read and write methods to use” on page 167).

The previous examples use FileStream objects opened for synchronous operation. You can also open files for asynchronous operations (which rely on event listener functions to respond to events). For example, the following code shows how to read an XML file asynchronously:

```
var file:File = File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
var fileStream:FileStream = new FileStream();
fileStream.addEventListener(Event.COMPLETE, processXMLData);
fileStream.openAsync(file, FileMode.READ);
var prefsXML:XML;

function processXMLData(event:Event):void
{
    prefsXML = XML(fileStream.readUTFBytes(fileStream.bytesAvailable));
    fileStream.close();
}
```

The `processXMLData()` method is invoked when the entire file is read into the read buffer (when the FileStream object dispatches the `complete` event). It calls the `readUTFBytes()` method to get a string version of the read data, and it creates an XML object, `prefsXML`, based on that string.

To see a sample application that shows these capabilities, see Reading and writing from an XML preferences file.

Example: Reading and writing data with random access

MP3 files can include ID3 tags, which are sections at the beginning or end of the file that contain metadata identifying the recording. The ID3 tag format itself has different revisions. This example describes how to read and write from an MP3 file that contains the simplest ID3 format (ID3 version 1.0) using "random access to file data", which means that it reads from and writes to arbitrary locations in the file.

An MP3 file that contains an ID3 version 1 tag includes the ID3 data at the end of the file, in the final 128 bytes.

When accessing a file for random read/write access, it is important to specify `FileMode.UPDATE` as the `fileMode` parameter for the `open()` or `openAsync()` method:

```
var file:File = File.documentsDirectory.resolvePath("My Music/Sample ID3 v1.mp3");
var fileStr:FileStream = new FileStream();
fileStr.open(file, FileMode.UPDATE);
```

This lets you both read and write to the file.

Upon opening the file, you can set the `position` pointer to the position 128 bytes before the end of the file:

```
fileStr.position = file.size - 128;
```

This code sets the `position` property to this location in the file because the ID3 v1.0 format specifies that the ID3 tag data is stored in the last 128 bytes of the file. The specification also says the following:

- The first 3 bytes of the tag contain the string `"TAG"`.
- The next 30 characters contain the title for the MP3 track, as a string.
- The next 30 characters contain the name of the artist, as a string.
- The next 30 characters contain the name of the album, as a string.
- The next 4 characters contain the year, as a string.
- The next 30 characters contain the comment, as a string.
- The next byte contains a code indicating the track's genre.
- All text data is in ISO 8859-1 format.

The `id3TagRead()` method checks the data after it is read in (upon the `complete` event):

```
function id3TagRead():void
{
    if (fileStr.readMultiByte(3, "iso-8859-1").match(/tag/i))
    {
        var id3Title:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Artist:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Album:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Year:String = fileStr.readMultiByte(4, "iso-8859-1");
        var id3Comment:String = fileStr.readMultiByte(30, "iso-8859-1");
        var id3GenreCode:String =  fileStr.readByte().toString(10);
    }
}

function id3TagRead()
{
    if (fileStr.readMultiByte(3, "iso-8859-1").match(/tag/i))
    {
        var id3Title = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Artist = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Album = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Year = fileStr.readMultiByte(4, "iso-8859-1");
        var id3Comment = fileStr.readMultiByte(30, "iso-8859-1");
        var id3GenreCode =  fileStr.readByte().toString(10);
    }
}
```

You can also perform a random-access write to the file. For example, you could parse the `id3Title` variable to ensure that it is correctly capitalized (using methods of the String class), and then write a modified string, called `newTitle`, to the file, as in the following:

```
fileStr.position = file.length - 125;    // 128 - 3
fileStr.writeMultiByte(newTitle, "iso-8859-1");
```

To conform with the ID3 version 1 standard, the length of the `newTitle` string should be 30 characters, padded at the end with the character code 0 (`String.fromCharCode(0)`).

Chapter 18: Drag and drop

Use the classes in the drag-and-drop API to support user-interface drag-and-drop gestures. A *gesture* in this sense is an action by the user, mediated by both the operating system and your application, expressing an intent, in this case, to copy, move, or link information. A *drag-out* gesture occurs when the user drags an object out of a component or application. A *drag-in* gesture occurs when the user drags in an object from outside a component or application.

With the drag-and-drop API, you can allow a user to drag data between applications and between components within an application. Supported transfer formats include:

- Bitmaps
- Files
- HTML-formatted text
- Text
- URLs
- Serialized objects
- Object references (only valid within the originating application)

Contents

Quick Starts (Adobe AIR Developer Center)

- Supporting drag-and-drop and copy-and-paste

Language Reference

- NativeDragManager
- NativeDragOptions
- Clipboard
- NativeDragEvent

More Information

- Adobe AIR Developer Center for Flex (search for 'AIR drag and drop')

Drag and drop basics

The drag-and-drop API contains the following classes.

Package	Classes
flash.desktop	• NativeDragManager • NativeDragOptions • Clipboard Constants used with the drag-and-drop API are defined in the following classes: • NativeDragActions • ClipboardFormat • ClipboardTransferModes
flash.events	NativeDragEvent

Drag-and-drop gesture stages

The drag-and-drop gesture has three stages:

Initiation *A user initiates a drag-and-drop operation by dragging from a component, or an item in a component, while holding down the mouse button.* The component that is the source of the dragged item is typically designated as the drag initiator and dispatches `nativeDragStart` and `nativeDragComplete` events. An Adobe® AIR™ application starts a drag operation by calling the `NativeDragManager.doDrag()` method in response to a `mouseDown` or `mouseMove` event.

Dragging *While holding down the mouse button, the user moves the mouse cursor to another component, application, or to the desktop.* AIR optionally displays a proxy image during the drag. As long as the drag is underway, the initiator object dispatches `nativeDragUpdate` events. When the user moves the mouse over a possible drop target in an AIR application, the drop target dispatches a `nativeDragEnter` event. The event handler can inspect the event object to determine whether the dragged data is available in a format that the target accepts and, if so, let the user drop the data onto it by calling the `NativeDragManager.acceptDragDrop()` method.

As long as the drag gesture remains over an interactive object, that object dispatches `nativeDragOver` events. When the drag gesture leaves the interactive object, it dispatches a `nativeDragExit` event.

Drop *The user releases the mouse over an eligible drop target.* If the target is an AIR application or component, then the component dispatches a `nativeDragDrop` event. The event handler can access the transferred data from the event object. If the target is outside AIR, the operating system or another application handles the drop. In both cases, the initiating object dispatches a `nativeDragComplete` event (if the drag started from within AIR).

The NativeDragManager class controls both drag-in and drag-out gestures. All the members of the NativeDrag-Manager class are static, do not create an instance of this class.

The Clipboard object

Data that is dragged into or out of an application or component is contained in a Clipboard object. A single Clipboard object can make available different representations of the same information to increase the likelihood that another application can understand and use the data. For example, an image could be included as image data, a serialized Bitmap object, and as a file. Rendering of the data in a format can be deferred to a rendering function that is not called until the data is read.

Once a drag gesture has started, the Clipboard object can only be accessed from within an event handler for the `nativeDragEnter`, `nativeDragOver`, and `nativeDragDrop` events. After the drag gesture has ended, the Clipboard object cannot be read or reused.

An application object can be transferred as a reference and as a serialized object. References are only valid within the originating application. Serialized object transfers are valid between AIR applications, but can only be used with objects that remain valid when serialized and deserialized. Objects that are serialized are converted into the Action Message Format for ActionScript 3 (AMF3), a string-based data-transfer format.

Working with the Flex framework

In most cases, it is better to use the Adobe® Flex™ drag-and-drop API when building Flex applications. The Flex framework provides an equivalent feature set when a Flex application is run in AIR (it uses the AIR NativeDrag-Manager internally). Flex also maintains a more limited feature set when an application or component is running within the more restrictive browser environment. AIR classes cannot be used in components or applications that run outside the AIR run-time environment.

Supporting the drag-out gesture

To support the drag-out gesture, you must create a Clipboard object in response to a `mouseDown` event and send it to the `NativeDragManager.doDrag()` method. Your application can then listen for the `nativeDragComplete` event on the initiating object to determine what action to take when the user completes or abandons the gesture.

Contents

Preparing data for transfer

To prepare data or an object for dragging, create a Clipboard object and add the information to be transferred in one or more formats. You can use the standard data formats to pass data that can be translated automatically to native clipboard formats, and application-defined formats to pass objects. If it is computationally expensive to convert the information to be transferred into a particular format, you can supply the name of a handler function to perform the conversion. The function is called if and only if the receiving component or application reads the associated format. For more information, see "Clipboard data formats" on page 192.

The following example illustrates how to create a Clipboard object containing a bitmap in several formats: a Bitmap object, a native bitmap format, and a file list format containing the file from which the bitmap was originally loaded:

```
import flash.desktop.Clipboard;
import flash.display.Bitmap;
import flash.filesystem.File;
public function createClipboard(image:Bitmap, sourceFile:File):Clipboard{
    var transfer:Clipboard = new Clipboard();
    transfer.setData("CUSTOM_BITMAP", image, true); //Flash object by value and by reference
    transfer.setData(ClipboardFormats.BITMAP_FORMAT, image.bitmapData, false);
    transfer.setData(ClipboardFormats.FILE_LIST_FORMAT, new Array(sourceFile), false);
    return transfer;
}
```

Starting a drag-out operation

To start a drag operation, call the `NativeDragManager.doDrag()` method in response to a mouse down event. The `doDrag()` method is a static method that takes the following parameters:

Parameter	Description
initiator	The object from which the drag originates, and which dispatches the `dragStart` and `dragComplete` events. The initiator must be an interactive object.
clipboard	The Clipboard object containing the data to be transferred. The Clipboard object is referenced in the Native-DragEvent objects dispatched during the drag-and-drop sequence.
dragImage	(Optional) A BitmapData object to display during the drag. The image can specify an `alpha` value. (Note: Microsoft Windows always applies a fixed alpha fade to drag images).
offset	(Optional) A Point object specifying the offset of the drag image from the mouse hotspot. Use negative coor-dinates to move the drag image up and left relative to the mouse cursor. If no offset is provided, the top, left corner of the drag image is positioned at the mouse hotspot.
actionsAllowed	(Optional) A NativeDragOptions object specifying which actions (copy, move, or link) are valid for the drag operation. If no argument is provided, all actions are permitted. The DragOptions object is referenced in NativeDragEvent objects to enable a potential drag target to check that the allowed actions are compatible with the purpose of the target component. For example, a “trash” component might only accept drag gestures that allow the move action.

The following example illustrates how to start a drag operation for a bitmap object loaded from a file. The example loads an image and, on a `mouseDown` event, starts the drag operation.

```
package
{
import flash.desktop.NativeDragManager;
import mx.core.UIComponent;
import flash.display.Loader;
import flash.system.LoaderContext;
import flash.net.URLRequest;
import flash.geom.Point;
import flash.desktop.Clipboard;
import flash.display.Bitmap;
import flash.filesystem.File;
import flash.events.Event;
import flash.events.MouseEvent;

public class DragOutExample extends UIComponent {
    protected var fileURL:String = "app:/image.jpg";
    protected var display:Bitmap;

    private function init():void {
        loadImage();
    }
    private function onMouseDown(event:MouseEvent):void {
        var bitmapFile:File = new File(fileURL);
        var transferObject:Clipboard = createClipboard(display, bitmapFile);
        NativeDragManager.doDrag(this,
                            transferObject,
                            display.bitmapData,
                            new Point(-mouseX,-mouseY));
    }
    public function createClipboard(image:Bitmap, sourceFile:File):Clipboard {
        var transfer:Clipboard = new Clipboard();
        transfer.setData("bitmap",
                            image,
                            true);
                            // ActionScript 3 Bitmap object by value and by reference
        transfer.setData(ClipboardFormats.BITMAP_FORMAT,
                            image.bitmapData,
                            false);
                            // Standard BitmapData format
        transfer.setData(ClipboardFormats.FILE_LIST_FORMAT,
                            new Array(sourceFile),
                            false);
                            // Standard file list format
        return transfer;
    }
    private function loadImage():void {
        var url:URLRequest = new URLRequest(fileURL);
        var loader:Loader = new Loader();
        loader.load(url,new LoaderContext());
        loader.contentLoaderInfo.addEventListener(Event.COMPLETE, onLoadComplete);
    }
    private function onLoadComplete(event:Event):void {
        display = event.target.loader.content;
        var flexWrapper:UIComponent = new UIComponent();
        flexWrapper.addChild(event.target.loader.content);
        addChild(flexWrapper);
        flexWrapper.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
    }
}
}
```

Completing a drag-out transfer

When a user drops the dragged item by releasing the mouse, the initiator object dispatches a `nativeDragComplete` event. You can check the `dropAction` property of the event object and then take the appropriate action. For example, if the action is `NativeDragAction.MOVE`, you could remove the source item from its original location. The user can abandon a drag gesture by releasing the mouse button while the cursor is outside an eligible drop target. The drag manager sets the `dropAction` property for an abandoned gesture to `NativeDragAction.NONE`.

Supporting the drag-in gesture

To support the drag-in gesture, your application (or, more typically, a visual component of your application) must respond to `nativeDragEnter` or `nativeDragOver` events.

Contents

Steps in a typical drop operation

The following sequence of events is typical for a drop operation:

1 The user drags a clipboard object over a component.

2 The component dispatches a `nativeDragEnter` event.

3 The `nativeDragEnter` event handler examines the event object to check the available data formats and allowed actions. If the component can handle the drop, it calls `NativeDragManager.acceptDragDrop()`.

4 The NativeDragManager changes the mouse cursor to indicate that the object can be dropped.

5 The user drops the object over the component.

6 The receiving component dispatches a `nativeDragDrop` event.

7 The receiving component reads the data in the desired format from the Clipboard object within the event object.

8 If the drag gesture originated within an AIR application, then the initiating interactive object dispatches a `nativeDragComplete` event. If the gesture originated outside AIR, no feedback is sent.

Acknowledging a drag-in gesture

When a user drags a clipboard item into the bounds of a visual component, the component dispatches `nativeDragEnter` and `nativeDragOver` events. To determine whether the component can accept the clipboard item, the handlers for these events can check the `clipboard` and `allowedActions` properties of the event object. To signal that the component can accept the drop, the event handler must call the `NativeDragManager.acceptDragDrop()` method, passing a reference to the receiving component. If more than one registered event listener calls the `acceptDragDrop()` method, the last handler in the list takes precedence. The `acceptDragDrop()` call remains valid until the mouse leaves the bounds of the accepting object, triggering the `nativeDragExit` event.

If more than one action is permitted in the `allowedActions` parameter passed to `doDrag()`, the user can indicate which of the allowed actions they intend to perform by holding down a modifier key. The drag manager changes the cursor image to tell the user which action would occur if they completed the drop. The intended action is reported by the `dropAction` property of the NativeDragEvent object. The action set for a drag gesture is advisory only. The components involved in the transfer must implement the appropriate behavior. To complete a move action, for example, the drag initiator might remove the dragged item and the drop target might add it.

Your drag target can limit the drop action to one of the three possible actions by setting the `dropAction` property of NativeDragManager class. If a user tries to choose a different action using the keyboard, then the NativeDrag-Manager displays the *unavailable* cursor. Set the `dropAction` property in the handlers for both the `nativeDragEnter` and the `nativeDragOver` events.

The following example illustrates an event handler for a `nativeDragEnter` or `nativeDragOver` event. This handler only accepts a drag-in gesture if the clipboard being dragged contains text-format data.

```
import flash.desktop.NativeDragManager;
import flash.events.NativeDragEvent;

public function onDragIn(event:NativeDragEvent):void{
    NativeDragManager.dropAction = NativeDragActions.MOVE;
    if(event.clipboard.hasFormat(ClipboardFormats.TEXT_FORMAT)){
        NativeDragManager.acceptDragDrop(this); //'this' is the receiving component
    }
}
```

Completing the drop

When the user drops a dragged item on an interactive object that has accepted the gesture, the interactive object dispatches a `nativeDragDrop` event. The handler for this event can extract the data from the `clipboard` property of the event object.

When the clipboard contains an application-defined format, the `transferMode` parameter passed to the `getData()` method of the Clipboard object determines whether the drag manager returns a reference or a serialized version of the object.

The following example illustrates an event handler for the `nativeDragDrop` event:

```
import flash.desktop.Clipboard;
import flash.events.NativeDragEvent;

public function onDrop(event:NativeDragEvent):void {
    if (event.clipboard.hasFormat(ClipboardFormats.TEXT_FORMAT)) {
    var text:String =
        String(event.clipboard.getData(ClipboardFormats.TEXT_FORMAT,
                                ClipboardTransferMode.ORIGINAL_PREFERRED));
}
```

Once the event handler exits, the Clipboard object is no longer valid. Any attempt to access the object or its data generates an error.

Updating the visual appearance of a component

A component can update its visual appearance based on the NativeDragEvent events. The following table describes the types of changes that a typical component would make in response to the different events:

Event	Description
nativeDragStart	The initiating interactive object can use the `nativeDragStart` event to provide visual feedback that the drag gesture originated from that interactive object.
nativeDragUpdate	The initiating interactive object can use the nativeDragUpdate event to update its state during the gesture.
nativeDragEnter	A potential receiving interactive object can use this event to take the focus, or indicate visually that it can or cannot accept the drop.
nativeDragOver	A potential receiving interactive object can use this event to respond to the movement of the mouse within the interactive object, such as when the mouse enters a “hot” region of a complex component such as a street map display.
nativeDragExit	A potential receiving interactive object can use this event to restore its state when a drag gesture moves outside its bounds.
nativeDragComplete	The initiating interactive object can use this event to update its associated data model, such as by removing an item from a list, and to restore its visual state.

Tracking mouse position during a drag-in gesture

While a drag gesture remains over a component, that component dispatches `nativeDragOver` events. These events are dispatched every few milliseconds and also whenever the mouse moves. The `nativeDragOver` event object can be used to determine the position of the mouse over the component. Having access to the mouse position can be helpful in situations where the receiving component is complex, but is not made up of sub-components. For example, if your application displayed a bitmap containing a street map and you wanted to highlight zones on the map when the user dragged information into them, you could use the mouse coordinates reported in the `nativeDragOver` event to track the mouse position within the map.

HTML Drag and drop

To drag data into and out of an HTML-based application (or into and out of the HTML displayed in an HTMLLoader), you can use HTML drag and drop events. The HTML drag-and-drop API allows you to drag to and from DOM elements in the HTML content.

***Note:** You can also use the AIR NativeDragEvent and NativeDragManager APIs by listening for events on the HTMLLoader object containing the HTML content. However, the HTML API is better integrated with the HTML DOM and gives you control of the default behavior.*

Contents

Default drag-and-drop behavior

THe HTML environment provides default behavior for drag-and-drop gestures involving text, images, and URLs. Using the default behavior, you can always drag these types of data out of an element. However, you can only drag text into an element and only to elements in an editable region of a page. When you drag text between or within editable regions of a page, the default behavior performs a move action. When you drag text to an editable region from a non-editable region or from outside the application, then the default behavior performs a copy action.

You can override the default behavior by handling the drag-and-drop events yourself. To cancel the default behavior, you must call the `preventDefault()` methods of the objects dispatched for the drag-and-drop events. You can then insert data into the drop target and remove data from the drag source as necessary to perform the chosen action.

By default, the user can select and drag any text, and drag images and links. You can use the WebKit CSS property, `-webkit-user-select` to control how any HTML element can be selected. For example, if you set `-webkit-user-select` to `none`, then the element contents are not selectable and so cannot be dragged. You can also use the `-webkit-user-drag` CSS property to control whether an element as a whole can be dragged. However, the contents of the element are treated separately. The user could still drag a selected portion of the text. For more information, see "Extensions to CSS" on page 256.

Drag-and-drop events in HTML

The events dispatched by the initiator element from which a drag originates, are:

Event	Description
dragstart	Dispatched when the user starts the drag gesture. The handler for this event can prevent the drag, if necessary, by calling the preventDefault() method of the event object. To control whether the dragged data can be copied, linked, or moved, set the effectAllowed property. Selected text, images, and links are put onto the clipboard by the default behavior, but you can set different data for the drag gesture using the dataTransfer property of the event object.
drag	Dispatched continuously during the drag gesture.
dragend	Dispatched when the user releases the mouse button to end the drag gesture.

The events dispatched by a drag target are:

Event	Description
dragover	Dispatched continuously while the drag gesture remains within the element boundaries. The handler for this event should set the dataTransfer.dropEffect property to indicate whether the drop will result in a copy, move, or link action if the user releases the mouse.
dragenter	Dispatched when the drag gesture enters the boundaries of the element. If you change any properties of a dataTransfer object in a dragenter event handler, those changes are quickly overridden by the next dragover event. On the other hand, there is a short delay between a dragenter and the first dragover event that can cause the cursor to flash if different properties are set. In many cases, you can use the same event handler for both events.
dragleave	Dispatched when the drag gesture leaves the element boundaries.
drop	Dispatched when the user drops the data onto the element. The data being dragged can only be accessed within the handler for this event.

The event object dispatched in response to these events is similar to a mouse event. You can use mouse event properties such as (`clientX`, `clientY`) and (`screenX`, `screenY`), to determine the mouse position.

The most important property of a drag event object is `dataTransfer`, which contains the data being dragged. The `dataTransfer` object itself has the following properties and methods:

Property or Method	Description
effectAllowed	The effect allowed by the source of the drag. Typically, the handler for the dragstart event sets this value. See "Drag effects in HTML" on page 180.
dropEffect	The effect chosen by the target or the user. If you set the `dropEffect` in a `dragover` or `dragenter` event handler, then AIR updates the mouse cursor to indicate the effect that occurs if the user releases the mouse. If the `dropEffect` set does not match one of the allowed effects, no drop is allowed and the *unavailable* cursor is displayed. If you have not set a `dropEffect` in response to the latest `dragover` or `dragenter` event, then the user can choose from the allowed effects with the standard operating system modifier keys. The final effect is reported by the `dropEffect` property of the object dispatched for `dragend`. If the user abandons the drop by releasing the mouse outside an eligible target, then `dropEffect` is set to `none`.
types	An array containing the MIME type strings for each data format present in the `dataTransfer` object.
getData(mimeType)	Gets the data in the format specified by the `mimeType` parameter. The `getData()` method can only be called in response to the `drop` event.

Property or Method	Description
setData(mimeType)	Adds data to the `dataTransfer` in the format specified by the mimeType parameter. You can add data in multiple formats by calling `setData()` for each MIME type. Any data placed in the `dataTransfer` object by the default drag behavior is cleared. The `setData()` method can only be called in response to the `dragstart` event.
clearData(mimeType)	Clears any data in the format specified by the `mimeType` parameter.
setDragImage(image, offsetX, offsetY)	Sets a custom drag image. The `setDragImage()` method can only be called in response to the dragstart event.

MIME types for the HTML drag-and-drop

The MIME types to use with the `dataTransfer` object of an HTML drag-and-drop event include:

Data format	MIME type
Text	"text/plain"
HTML	"text/html"
URL	"text/uri-list"
Bitmap	"image/x-vnd.adobe.air.bitmap"
File list	"application/x-vnd.adobe.air.file-list"

You can also use other MIME strings, including application-defined strings. However, other applications may not be able to recognize or use the transferred data. It is your responsibility to add data to the `dataTransfer` object in the expected format.

Important: *Only code running in the application sandbox can access dropped files. Attempting to read or set any property of a File object within a non-application sandbox generates a security error. See "Handling file drops in non-application HTML sandboxes" on page 184 for more information.*

Drag effects in HTML

The initiator of the drag gesture can limit the allowed drag effects by setting the `dataTransfer.effectAllowed` property in the handler for the `dragstart` event. The following string values can be used:

String value	Description
"none"	No drag operations are allowed.
"copy"	The data will be copied to the destination, leaving the original in place.
"link"	The data will be shared with the drop destination using a link back to the original.
"move"	The data will be copied to the destination and removed from the original location.
"copyLink"	The data can be copied or linked.
"copyMove"	The data can be copied or moved.
"linkMove"	The data can be linked or moved.
"all"	The data can be copied, moved, or linked. *All* is the default effect when you prevent the default behavior.

The target of the drag gesture can set the `dataTransfer.dropEffect` property to indicate the action that is taken if the user completes the drop. If the drop effect is one of the allowed actions, then the system displays the appropriate copy, move, or link cursor. If not, then the system displays the *unavailable* cursor. If no drop effect is set by the target, the user can choose from the allowed actions with the modifier keys.

Set the `dropEffect` value in the handlers for both the `dragover` and `dragenter` events:

```
function doDragStart(event) {
    event.dataTransfer.setData("text/plain","Text to drag");
    event.dataTransfer.effectAllowed = "copyMove";
}

function doDragOver(event) {
    event.dataTransfer.dropEffect = "copy";
}


function doDragEnter(event) {
    event.dataTransfer.dropEffect = "copy";
}
```

***Note:** Although you should always set the `dropEffect` property in the handler for `dragenter`, be aware that the next `dragover` event resets the property to its default value. Set `dropEffect` in response to both events.*

Dragging data out of an HTML element

The default behavior allows most content in an HTML page to be copied by dragging. You can control the content allowed to be dragged using CSS properties `-webkit-user-select` and `-webkit-user-drag`.

Override the default drag-out behavior in the handler for the `dragstart` event. Call the `setData()` method of the `dataTransfer` property of the event object to put your own data into the drag gesture.

To indicate which drag effects a source object supports when you are not relying on the default behavior, set the `dataTransfer.effectAllowed` property of the event object dispatched for the `dragstart` event. You can choose any combination of effects. For example, if a source element supports both *copy* and *link* effects, set the property to `"copyLink"`.

Setting the dragged data

Add the data for the drag gesture in the handler for the `dragstart` event with the `dataTransfer` property. Use the `dataTransfer.setData()` method to put data onto the clipboard, passing in the MIME type and the data to transfer.

For example, if you had an image element in your application, with the id *imageOfGeorge*, you could use the following dragstart event handler. This example adds representations of a picture of George in several data formats, which increases the likelihood that other applications can use the dragged data.

```
function dragStartHandler(event){
    event.dataTransfer.effectAllowed = "copy";

    var dragImage = document.getElementById("imageOfGeorge");
    var dragFile = new air.File(dragImage.src);
    event.dataTransfer.setData("text/plain","A picture of George");
    event.dataTransfer.setData("image/x-vnd.adobe.air.bitmap", dragImage);
    event.dataTransfer.setData("application/x-vnd.adobe.air.file-list",
                                new Array(dragFile));
}
```

***Note:** When you call the `setData()` method of `dataTransfer` object, no data is added by the default drag-and-drop behavior.*

Dragging data into an HTML element

The default behavior only allows text to be dragged into editable regions of the page. You can specify that an element and its children can be made editable by including the `contenteditable` attribute in the opening tag of the element. You can also make an entire document editable by setting the document object `designMode` property to `"on"`.

You can support alternate drag-in behavior on a page by handling the `dragenter`, `dragover`, and `drop` events for any elements that can accept dragged data.

Enabling drag-in

To handle the drag-in gesture, you must first cancel the default behavior. Listen for the `dragenter` and `dragover` events on any HTML elements you want to use as drop targets. In the handlers for these events, call the `preventDefault()` method of the dispatched event object. Canceling the default behavior allows non-editable regions to receive a drop.

Getting the dropped data

You can access the dropped data in the handler for the `ondrop` event:

```
function doDrop(event){
    droppedText = event.dataTransfer.getData("text/plain");
}
```

Use the `dataTransfer.getData()` method to read the data onto the clipboard, passing in the MIME type of the data format to read. You can find out which data formats are available using the `types` property of the `dataTransfer` object. The `types` array contains the MIME type string of each available format.

When you cancel the default behavior in the dragenter or dragover events, you are responsible for inserting any dropped data into its proper place in the document. No API exists to convert a mouse position into an insertion point within an element. This limitation can make it difficult to implement insertion-type drag gestures.

Example: Overriding the default HTML drag-in behavior

This example implements a drop target that displays a table showing each data format available in the dropped item.

The default behavior is used to allow text, links, and images to be dragged within the application. The example overrides the default drag-in behavior for the div element that serves as the drop target. The key step to enabling non-editable content to accept a drag-in gesture is to call the `preventDefault()` method of the event object dispatched for both the `dragenter` and `dragover` events. In response to a `drop` event, the handler converts the transferred data into an HTML row element and inserts the row into a table for display.

```
<html>
<head>
<title>Drag-and-drop</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript">
    function init(){
        var target = document.getElementById('target');
        target.addEventListener("dragenter", dragEnterOverHandler);
        target.addEventListener("dragover", dragEnterOverHandler);
        target.addEventListener("drop", dropHandler);

        var source = document.getElementById('source');
        source.addEventListener("dragstart", dragStartHandler);
        source.addEventListener("dragend", dragEndHandler);

        emptyRow = document.getElementById("emptyTargetRow");
    }
    function dragStartHandler(event){
        event.dataTransfer.effectAllowed = "copy";
    }

    function dragEndHandler(event){
        air.trace(event.type + ": " + event.dataTransfer.dropEffect);
    }

    function dragEnterOverHandler(event){
        event.preventDefault();
    }

    var emptyRow;
    function dropHandler(event){
        for(var prop in event){
            air.trace(prop + " = " + event[prop]);
        }
        var row = document.createElement('tr');
        row.innerHTML = "<td>" + event.dataTransfer.getData("text/plain") + "</td>" +
            "<td>" + event.dataTransfer.getData("text/html") + "</td>" +
            "<td>" + event.dataTransfer.getData("text/uri-list") + "</td>" +
            "<td>" + event.dataTransfer.getData("application/x-vnd.adobe.air.file-list") +
            "</td>";

        var imageCell = document.createElement('td');
        if((event.dataTransfer.types.toString()).search("image/x-vnd.adobe.air.bitmap") > -
1){
            imageCell.appendChild(event.dataTransfer.getData("image/x-
vnd.adobe.air.bitmap"));
        }
        row.appendChild(imageCell);
        var parent = emptyRow.parentNode;
        parent.insertBefore(row, emptyRow);
    }
</script>
</head>
<body onLoad="init()" style="padding:5px">
<div>
    <h1>Source</h1>
    <p>Items to drag:</p>
    <ul id="source">
    <li>Plain text.</li>
        <li>HTML <b>formatted</b> text.</li>
        <li>A <a href="http://www.adobe.com">URL.</a></li>
        <li><img src="icons/AIRApp_16.png" alt="An image"/></li>
        <li style="-webkit-user-drag:none;">
        Uses "-webkit-user-drag:none" style.
        </li>
        <li style="-webkit-user-select:none;">
        Uses "-webkit-user-select:none" style.
        </li>

    </ul>
</div>
<div id="target" style="border-style:dashed;">
    <h1 >Target</h1>
    <p>Drag items from the source list (or elsewhere).</p>
    <table id="displayTable" border="1">
    <tr><th>Plain text</th><th>Html text</th><th>URL</th><th>File list</th><th>Bitmap
Data</th></tr>
```

```
        <tr 
id="emptyTargetRow"><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;</td><td>&nbsp;
</td></tr>
    </table>
    </div>
</div>
</body>
</html>
```

Handling file drops in non-application HTML sandboxes

Non-application content cannot access the File objects that result when files are dragged into an AIR application. Nor is it possible to pass one of these File objects to application content through a sandbox bridge. (The object properties must be accessed during serialization.) However, you can still drop files in your application by listening for the AIR nativeDragDrop events on the HTMLLoader object.

Normally, if a user drops a file into a frame that hosts non-application content, the drop event does not propagate from the child to the parent. However, since the events dispatched by the HTMLLoader (which is the container for all HTML content in an AIR application) are not part of the HTML event flow, you can still receive the drop event in application content.

To receive the event for a file drop, the parent document adds an event listener to the HTMLLoader object using the reference provided by `window.htmlLoader`:

```
window.htmlLoader.addEventListener("nativeDragDrop",function(event){
    var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
    air.trace(filelist[0].url);
});
```

The following example uses a parent document that loads a child page into a remote sandbox (http://localhost/). The parent listens for the `nativeDragDrop` event on the HTMLLoader object and traces out the file url.

```
<html>
<head>
<title>Drag-and-drop in a remote sandbox</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript">
    window.htmlLoader.addEventListener("nativeDragDrop",function(event){
        var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
        air.trace(filelist[0].url);
    });
</script>
</head>
<body>
    <iframe src="child.html"
            sandboxRoot="http://localhost/"
            documentRoot="app:/"
            frameBorder="0"  width="100%" height="100%">
    </iframe>
</body>
</html>
```

The child document must present a valid drop target by preventing the Event object `preventDefault()` method in the HTML `dragenter` and `dragover` event handlers or the drop event can never occur.

```
<html>
<head>
    <title>Drag and drop target</title>
    <script language="javascript" type="text/javascript">
        function preventDefault(event){
            event.preventDefault();
        }
    </script>
</head>
<body ondragenter="preventDefault(event)" ondragover="preventDefault(event)">
<div>
<h1>Drop Files Here</h1>
</div>
</body>
</html>
```

See also

- “Programming in HTML and JavaScript” on page 258

Chapter 19: Copy and paste

Use the classes in the clipboard API to copy information to and from the system clipboard. The data formats that can be transferred into or out of an Adobe® AIR™ application include:

- Bitmaps
- Files
- Text
- URL strings
- Serialized objects
- Object references (only valid within the originating application)

Contents

Quick Starts (Adobe AIR Developer Center)

- Supporting drag-and-drop and copy and paste

Language Reference

- Clipboard
- ClipboardFormats
- ClipboardTransferMode

More Information

- Adobe AIR Developer Center for Flex (search for 'AIR copy and paste')

Copy-and-paste basics

The copy-and-paste API contains the following classes.

Package	Classes
flash.desktop	• Clipboard Constants used with the copy-and-paste API are defined in the following classes: • ClipboardFormats • ClipboardTransferMode

The static `Clipboard.generalClipboard` property represents the operating system clipboard. Access the system clipboard through the static `Clipboard.generalClipboard` property. The Clipboard class provides methods for reading and writing data to clipboard objects. Clipboard objects are also used to transfer data through the drag-and-drop API.

The HTML environment provides an alternate API for copy and paste. Either API can be used by code running within the application sandbox, but only the HTML API can be used in non-application content.

The HTMLLoader and TextField classes implement default behavior for the normal copy and paste keyboard shortcuts. To implement copy and paste shortcut behavior for custom components, you can listen for these keystrokes directly. You can also use native menu commands along with key equivalents to respond to the keystrokes indirectly.

Different representations of the same information can be made available in a single Clipboard object to increase the ability of other applications to understand and use the data. For example, an image might be included as image data, a serialized Bitmap object, and as a file. Rendering of the data in a format can be deferred so that the format is not actually created until the data in that format is read.

Reading from and writing to the system clipboard

To read the operating system clipboard, call the `getData()` method of the `Clipboard.generalClipbooard` object, passing in the name of the format to read:

```
import flash.desktop.Clipboard;
import flash.desktop.ClipboardFormats;

if(Clipboard.generalClipboard.hasFormat(ClipboardFormats.TEXT_FORMAT)){
    var text:String = Clipboard.generalClipboard.getData(ClipboardFormats.TEXT_FORMAT);
}
```

To write to the clipboard, add the data to the `Clipboard.generalClipboard` object in one or more formats. Any existing data in the same format is overwritten automatically. However, it is a good practice to also clear the system clipboard before writing new data to it to make sure that unrelated data in any other formats is also deleted.

```
import flash.desktop.Clipboard;
import flash.desktop.ClipboardFormats;

var textToCopy:String = "Copy to clipboard.";
Clipboard.generalClipboard.clear();
Clipboard.generalClipboard.setData(ClipboardFormats.TEXT_FORMAT, textToCopy, false);
```

***Note:** Only code running in the application sandbox can access the system clipboard directly. In non-application HTML content, you can only access the clipboard through the `clipboardData` property of an event object dispatched by one of the HTML copy or paste events.*

HTML copy and paste

The HTML environment provides its own set of events and default behavior for copy and paste. Only code running in the application sandbox can access the system clipboard directly through the AIR `Clipboard.generalClipboard` object. JavaScript code in a non-application sandbox can access the clipboard through the event object dispatched in response to one of the copy or paste events dispatched by an element in an HTML document.

Copy and paste events include: `copy`, `cut`, and `paste`. The object dispatched for these events provides access to the clipboard through the `clipboardData` property.

Contents

Default behavior

By default, AIR copies selected items in response to the copy command, which can be generated either by a keyboard shortcut or a context menu. Within editable regions, AIR cuts text in response to the cut command or pastes text to the cursor or selection in response to the paste command.

To prevent the default behavior, your event handler can call the `preventDefault()` method of the dispatched event object.

Using the clipboardData property of the event object

The `clipboardData` property of the event object dispatched as a result of one of the copy or paste events allows you to read and write clipboard data.

To write to the clipboard when handling a copy or cut event, use the `setData()` method of the `clipboardData` object, passing in the data to copy and the MIME type:

```
function customCopy(event){
    event.clipboardData.setData("text/plain", "A copied string.");
}
```

To access the data that is being pasted, you can use the `getData()` method of the `clipboardData` object, passing in the MIME type of the data format. The available formats are reported by the `types` property.

```
function customPaste(event){
    var pastedData = event.clipboardData("text/plain");
}
```

The `getData()` method and the `types` property can only be accessed in the event object dispatched by the `paste` event.

The following example illustrates how to override the default copy and paste behavior in an HTML page. The `copy` event handler italicizes the copied text and copies it to the clipboard as HTML text. The `cut` event handler copies the selected data to the clipboard and removes it from the document. The `paste` handler inserts the clipboard contents as HTML and bolds the insertion as well.

```
<html>
<head>
    <title>Copy and Paste</title>
    <script language="javascript" type="text/javascript">
        function onCopy(event){
            var selection = window.getSelection();
            event.clipboardData.setData("text/html","<i>" + selection + "</i>");
            event.preventDefault();
        }

        function onCut(event){
             var selection = window.getSelection();
             event.clipboardData.setData("text/html","<i>" + selection + "</i>");
             var range = selection.getRangeAt(0);
             range.extractContents();
            event.preventDefault();
        }

        function onPaste(event){
            var insertion = document.createElement("b");
            insertion.innerHTML = event.clipboardData.getData("text/html");
             var selection = window.getSelection();
             var range = selection.getRangeAt(0);
             range.insertNode(insertion);
            event.preventDefault();
        }
    </script>
</head>
<body onCopy="onCopy(event)"
     onPaste="onPaste(event)"
     onCut="onCut(event)">
<p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore
veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam
voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur
magni dolores eos qui ratione voluptatem sequi nesciunt.</p>
</body>
</html>
```

Menu commands and keystrokes for copy and paste

Copy and paste functionality is commonly triggered through menu commands and keyboard shortcuts. On OS X, an edit menu with the copy and paste commands is automatically created by the operating system, but you must add listeners to these menu commands to hook up your own copy and paste functions. On Windows, you can add a native edit menu to any window that uses system chrome. (You can also create non-native menus with Flex and ActionScript, or, in HTML content, you can use DHTML, but that is beyond the scope of this discussion.

To trigger copy and paste commands in response to keyboard shortcuts, you can either assign key equivalents to the appropriate command items in a native application or window menu, or you can listen for the keystrokes directly.

Contents

Starting a copy or paste operation with a menu command

To trigger a copy or paste operation with a menu command, you must add listeners for the `select` event on the menu items that call your handler functions.

When your handler function is called, you can find the object to be copied from or pasted into using the `focus` property of the stage. You can then call the appropriate method of the focused object (or a general fallback method, if no object has focus) to carry out the copy, cut, or paste logic. For example, the following `copy` event handler checks whether the focused object is of the correct type, in this case, a class named *Scrap*, and then calls the object’s `doCopy()` method.

```
function copyCommand(event:Event):void{
    if(NativeApplication.nativeApplication.activeWindow.stage.focus is Scrap){
        Scrap(NativeApplication.nativeApplication.activeWindow.stage.focus).doCopy();
    } else {
        NativeApplication.nativeApplication.copy();
    }
}
```

If `copyCommand()` in the example does not recognize the class of the focused object, it calls the NativeApplication `copy()` method. The NativeApplication `copy()` method sends an internal copy command to the focused object. The internal command is only recognized by the TextArea and HTMLLoader objects. Similar commands are available for cut, paste, select all, and for the TextArea only, clear, undo, and redo.

Note: *There is no API provided to respond to these internal commands in a custom component. You must either extend the TextArea or HTMLLoader classes, or include one of these objects in your custom component. If you include a TextArea or HTMLLoader, your component must manage the focus such that the TextArea or HTMLLoader object always keeps the focus when the component itself has focus.*

In HTML content, the default copy and paste behavior can be triggered using the NativeApplication edit commands. The following example creates an edit menu for an editable HTML document:

```
<html>
<head>
    <title>Edit Menu</title>
    <script src="AIRAliases.js" type="text/javascript"></script>
    <script language="javascript" type="text/javascript">
        function init(){
            document.designMode = "On";
            addEditMenu();
        }

        function addEditMenu(){
            var menu = new air.NativeMenu
            var edit = menu.addSubmenu(new air.NativeMenu(), "Edit");

            var copy = edit.submenu.addItem(new air.NativeMenuItem("Copy"));
            var cut = edit.submenu.addItem(new air.NativeMenuItem("Cut"));
            var paste = edit.submenu.addItem(new air.NativeMenuItem("Paste"));
            var selectAll = edit.submenu.addItem(new air.NativeMenuItem("Select All"));


            copy.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.copy();
            });
            cut.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.cut();
            });
            paste.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.paste();
            });

            selectAll.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.selectAll();
            });

            copy.keyEquivalent = "c";
            cut.keyEquivalent = "x";
            paste.keyEquivalent = "v";
            selectAll.keyEquivalent = "a";

            if(air.NativeWindow.supportsMenu){
                window.nativeWindow.menu = menu;
            } else if (air.NativeApplication.supportsMenu){
                air.NativeApplication.nativeApplication.menu = menu;
            }
        }
    </script>
</head>
<body onLoad="init()">
    <p>Neque porro quisquam est qui dolorem ipsum
    quia dolor sit amet, consectetur, adipisci velit.</p>
</body>
</html>
```

The previous example replaces the application menu on Mac OS X, but you can also make use of the default Edit menu by finding the existing items and adding event listeners to them.

If you use a context menu to invoke a copy or paste command, you can use the `contextMenuOwner` property of the ContextMenuEvent object dispatched when the menu is opened or an item is selected to determine which object is the proper target of the copy or paste command.

Finding default menu items on Mac OS X

To find the default edit menu and the specific copy, cut, and paste command items in the application menu on Mac OS X, you can search through the menu hierarchy using the `label` property of the NativeMenuItem objects. For example, the following function takes a name and finds the item with the matching label in the menu:

```
private function findItemByName(menu:NativeMenu,
                    name:String,
                    recurse:Boolean = false):NativeMenuItem{
    var searchItem:NativeMenuItem = null;
    for each (var item:NativeMenuItem in menu.items){
        if(item.label == name){
            searchItem = item;
            break;
        }
        if((item.submenu != null) && recurse){
             searchItem = findItemByName(item.submenu, name);
        }
    }
    return searchItem;
}
```

You can set the `recurse` parameter to `true` to include submenus in the search, or `false` to include only the passed-in menu.

Starting a copy or paste command with a keystroke

If your application uses native window or application menus for copy and paste, you can add key equivalents to the menu items to implement keyboard shortcuts. Otherwise, you can listen for the relevant keystrokes yourself, as demonstrated in the following example:

```
private function init():void{
    stage.addEventListener(KeyboardEvent.KEY_DOWN, keyListener);
}
private function keyListener(event:KeyboardEvent):void{
    if(event.ctrlKey){
        event.preventDefault();
        switch(String.fromCharCode(event.charCode)){
            case "c":
                NativeApplication.nativeApplication.copy();
                break;
            case "x":
                NativeApplication.nativeApplication.cut();
                break;
            case "v":
                NativeApplication.nativeApplication.paste();
                break;
            case "a":
                NativeApplication.nativeApplication.selectAll();
                break;
            case "z":
                NativeApplication.nativeApplication.undo();
                break;
            case "y":
                NativeApplication.nativeApplication.redo();
                break;
        }
    }
}
```

In HTML content, the keyboard shortcuts for copy and paste commands are implemented by default. It is not possible to trap all of the keystrokes commonly used for copy and paste using a key event listener. If you need to override the default behavior, a better strategy is to listen for the `copy` and `paste` events themselves.

Clipboard data formats

Clipboard formats describe the data placed in a Clipboard object. AIR automatically translates the standard data formats between ActionScript data types and system clipboard formats. In addition, application objects can be transferred within and between AIR applications using application-defined formats.

A Clipboard object can contain representations of the same information in different formats. For example, a Clipboard object representing a Sprite could include a reference format for use within the same application, a serialized format for use by another AIR application, a bitmap format for use by an image editor, and a file list format, perhaps with deferred rendering to encode a PNG file, for copying or dragging a representation of the Sprite to the file system.

Contents

Standard data formats

The constants defining the standard format names are provided in the ClipboardFormats class:

Constant	Description
TEXT_FORMAT	Text-format data is translated to and from the ActionScript String class.
BITMAP_FORMAT	Bitmap-format data is translated to and from the ActionScript BitmapData class.
FILE_LIST_FORMAT	File-list-format data is translated to and from an array of ActionScript File objects.
URL_FORMAT	URL-format data is translated to and from the ActionScript String class.

When copying and pasting data in response to a `copy`, `cut`, or `paste` event in HTML content, MIME types must be used instead of the ClipboardFormat strings. The valid data MIME types are:

MIME type	**Description**
Text	"text/plain"
URL	"text/uri-list"
Bitmap	"image/x-vnd.adobe.air.bitmap"
File list	"application/x-vnd.adobe.air.file-list"

Custom data formats

You can use application-defined custom formats to transfer objects as references or as serialized copies. References are only valid within the same AIR application. Serialized objects can be transferred between Adobe AIR applications, but can only be used with objects that remain valid when serialized and deserialized. Objects can usually be serialized if their properties are either simple types or serializable objects.

To add a serialized object to a Clipboard object, set the serializable parameter to `true` when calling the `Clipboard.setData()` method. The format name can be one of the standard formats or an arbitrary string defined by your application.

Transfer modes

When an object is written to the clipboard using a custom data format, the object data can be read from the clipboard either as reference or as a serialized copy of the original object. AIR defines four transfer modes that determine whether objects are transferred as references or as serialized copies:

Transfer mode	**Description**
ClipboardTransferModes.ORIGINAL_ONLY	Only a reference is returned. If no reference is available, a `null` value is returned.
ClipboardTransferModes.ORIGINAL_PREFFERED	A reference is returned, if available. Otherwise a serialized copy is returned.
ClipboardTransferModes.CLONE_ONLY	Only a serialized copy is returned. If no serialized copy is available, then a `null` value is returned.
ClipboardTransferModes.CLONE_PREFFERED	A serialized copy is returned, if available. Otherwise a reference is returned.

Reading and writing custom data formats

You can use any string that does not begin with the reserved prefix `air:` for the format parameter when writing an object to the clipboard. Use the same string as the format to read the object. The following examples illustrate how to read and write objects to the clipboard:

```
public function createClipboardObject(object:Object):Clipboard{
    var transfer:Clipboard = new Clipboard();
    transfer.setData("object", object, true);
}
```

To extract a serialized object from the clipboard object (after a drop or paste operation), use the same format name and the `cloneOnly` or `clonePreferred` transfer modes.

```
var transfer:Object = clipboard.getData("object", ClipboardTransferMode.CLONE_ONLY);
```

A reference is always added to the Clipboard object. To extract the reference from the clipboard object (after a drop or paste operation), instead of the serialized copy, use the `originalOnly` or `originalPreferred` transfer modes:

```
var transferredObject:Object =
    clipboard.getData("object", ClipboardTransferMode.ORIGINAL_ONLY);
```

References are only valid if the Clipboard object originates from the current AIR application. Use the `originalPreferred` transfer mode to access the reference when it is available, and the serialized clone when the reference is not available.

Deferred rendering

If creating a data format is computationally expensive, you can use deferred rendering by supplying a function that supplies the data on demand. The function is only called if a receiver of the drop or paste operation requests data in the deferred format.

The rendering function is added to a Clipboard object using the `setDataHandler()` method. The function must return the data in the appropriate format. For example, if you called `setDataHandler(ClipboardFormat.TEXT_FORMAT, writeText)`, then the `writeText()` function must return a string.

If a data format of the same type is added to a Clipboard object with the `setData()` method, that data will take precedence over the deferred version (the rendering function is never called). The rendering function may or may not be called again if the same clipboard data is accessed a second time.

Pasting text using a deferred rendering function

The following example illustrates how to implement a deferred rendering function.

When the Copy button in the example is pressed, the application clears the system clipboard to ensure that no data is left over from previous clipboard operations, then puts the `renderData()` function onto the clipboard with the clipboard `setDataHandler()` method.

When the Paste button is pressed, the application accesses the clipboard and sets the destination text. Since the text data format on the clipboard has been set with a function rather than a string, the clipboard will call the `renderData()` function. The `renderData()` function returns the text in the source text, which is then assigned to the destination text.

Notice that if you edit the source text before pressing the Paste button, the edit will be reflected in the pasted text, even when the edit occurs after the copy button was pressed. This is because the rendering function doesn't copy the source text until the paste button is pressed. (When using deferred rendering in a real application, you might want to store or protect the source data in some way to prevent this problem.)

```
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
width="326" height="330">
<mx:Script>
    <![CDATA[
        import flash.desktop.Clipboard;
        import flash.desktop.ClipboardFormats;

        public function doCopy():void{
            Clipboard.generalClipboard.clear();
            Clipboard.generalClipboard.setDataHandler(ClipboardFormats.TEXT_FORMAT,
                                                    renderData);
        }

        public function doPaste():void{
            destination.text =
                Clipboard.generalClipboard.getData(ClipboardFormats.TEXT_FORMAT) as String;
        }

        public function renderData():String{
            trace("Rendering data");
            return source.text;
        }
    ]]>
</mx:Script>
    <mx:Label x="10" y="10" text="Source"/>
    <mx:TextArea id="source" x="10" y="36" width="300" height="100">
    <mx:text>Neque porro quisquam est qui dolorem ipsum
    quia dolor sit amet, consectetur, adipisci velit.</mx:text>
    </mx:TextArea>
    <mx:Label x="10" y="181" text="Destination"/>
    <mx:TextArea id="destination"  x="12" y="207" width="300" height="100"/>
    <mx:Button click="doPaste();" x="166" y="156" label="Paste"/>
    <mx:Button click="doCopy();" x="91" y="156" label="Copy"/>
</mx:WindowedApplication>
```

Chapter 20: Working with byte arrays

The ByteArray class allows you to read from and write to a binary stream of data, which is essentially an array of bytes. This class provides a way to access data at the most elemental level. Because computer data consists of bytes, or groups of 8 bits, the ability to read data in bytes means that you can access data for which classes and access methods do not exist. The ByteArray class allows you to parse any stream of data, from a bitmap to a stream of data traveling over the network, at the byte level.

The `writeObject()` method allows you to write an object in serialized Action Message Format (AMF) to a ByteArray, while the `readObject()` method allows you to read a serialized object from a ByteArray to a variable of the original data type. You can serialize any object except for display objects, which are those objects that can be placed on the display list. You can also assign serialized objects back to custom class instances if the custom class is available to the runtime. After converting an object to AMF, you can efficiently transfer it over a network connection or save it to a file.

The sample Adobe® AIR™ application described here reads a .zip file as an example of processing a byte stream; extracting a list of the files that the .zip file contains and writing them to the desktop.

Contents

Reading and writing a ByteArray

The ByteArray class is part of the flash.utils package. To create a ByteArray object in ActionScript 3.0, import the ByteArray class and invoke the constructor, as shown in the following example:

```
import flash.utils.ByteArray;
var stream:ByteArray = new ByteArray();
```

Contents

ByteArray methods

Any meaningful data stream is organized into a format that you can analyze to find the information that you want. A record in a simple employee file, for example, would probably include an ID number, a name, an address, a phone number, and so on. An MP3 audio file contains an ID3 tag that identifies the title, author, album, publishing date, and genre of the file that's being downloaded. The format allows you to know the order in which to expect the data on the data stream. It allows you to read the byte stream intelligently.

The ByteArray class includes several methods that make it easier to read from and write to a data stream. Some of these methods include `readBytes()` and `writeBytes()`, `readInt()` and `writeInt()`, `readFloat()` and `writeFloat()`, `readObject()` and `writeObject()`, and `readUTFBytes()` and `writeUTFBytes()`. These methods enable you to read data from the data stream into variables of specific data types and write from specific data types directly to the binary data stream.

For example, the following code reads a simple array of strings and floating-point numbers and writes each element to a ByteArray. The organization of the array allows the code to call the appropriate ByteArray methods (`writeUTFBytes()` and `writeFloat()`) to write the data. The repeating data pattern makes it possible to read the array with a loop.

```
// The following example reads a simple Array (groceries), made up of strings
// and floating-point numbers, and writes it to a ByteArray.

import flash.utils.ByteArray;

// define the grocery list Array
var groceries:Array = ["milk", 4.50, "soup", 1.79, "eggs", 3.19, "bread" , 2.35]
// define the ByteArray
var bytes:ByteArray = new ByteArray();
// for each item in the array
for (var i:int = 0; i < groceries.length; i++) {
        bytes.writeUTFBytes(groceries[i++]); //write the string and position to the next 
item
        bytes.writeFloat(groceries[i]);// write the float
        trace("bytes.position is: " + bytes.position);//display the position in ByteArray
}
trace("bytes length is: " +  bytes.length);// display the length
```

The position property

The position property stores the current position of the pointer that indexes the ByteArray during reading or writing. The initial value of the position property is 0 (zero) as shown in the following code:

```
var bytes:ByteArray = new ByteArray();
trace("bytes.position is initially: " + bytes.position); // 0
```

When you read from or write to a ByteArray, the method that you use updates the position property to point to the location immediately following the last byte that was read or written. For example, the following code writes a string to a ByteArray and afterward the position property points to the byte immediately following the string in the ByteArray:

```
var bytes:ByteArray = new ByteArray();
trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
trace("bytes.position is now: " + bytes.position);// 12
```

Likewise, a read operation increments the position property by the number of bytes read.

```
var bytes:ByteArray = new ByteArray();

trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
trace("bytes.position is now: " + bytes.position);// 12
bytes.position = 0;
trace("The first 6 bytes are: " + (bytes.readUTFBytes(6)));//Hello 
trace("And the next 6 bytes are: " + (bytes.readUTFBytes(6)));// World!
```

Notice that you can set the position property to a specific location in the ByteArray to read or write at that offset.

The bytesAvailable and length properties

The `length` and `bytesAvailable` properties tell you how long a ByteArray is and how many bytes remain in it from the current position to the end. The following example illustrates how you can use these properties. The example writes a String of text to the ByteArray and then reads the ByteArray one byte at a time until it encounters either the character “a” or the end (`bytesAvailable <= 0`).

```
var bytes:ByteArray = new ByteArray();
var text:String = "Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vivamus etc.";

bytes.writeUTFBytes(text); // write the text to the ByteArray
trace("The length of the ByteArray is: " + bytes.length);// 70
bytes.position = 0; // reset position
while (bytes.bytesAvailable > 0 && (bytes.readUTFBytes(1) != 'a')) {
    //read to letter a or end of bytes
}
if (bytes.position < bytes.bytesAvailable) {
    trace("Found the letter a; position is: " + bytes.position); // 23
    trace("and the number of bytes available is: " + bytes.bytesAvailable);// 47
}
```

The endian property

Computers can differ in how they store multibyte numbers, that is, numbers that require more than 1 byte of memory to store them. An integer, for example, can take 4 bytes, or 32 bits, of memory. Some computers store the most significant byte of the number first, in the lowest memory address, and others store the least significant byte first. This attribute of a computer, or of byte ordering, is referred to as being either *big endian* (most significant byte first) or *little endian* (least significant byte first). For example, the number 0x31323334 would be stored as follows for big endian and little endian byte ordering, where a0 represents the lowest memory address of the 4 bytes and a3 represents the highest:

Big Endian			
a0	a1	a2	a3
31	32	33	34

Little Endian			
a0	a1	a2	a3
34	33	32	31

The `endian` property of the ByteArray class allows you to denote this byte order for multibyte numbers that you are processing. The acceptable values for this property are either `"bigEndian"` or `"littleEndian"` and the Endian class defines the constants `BIG_ENDIAN` and `LITTLE_ENDIAN` for setting the `endian` property with these strings.

The compress() and uncompress() methods

The `compress()` method allows you to compress a ByteArray in accordance with a compression algorithm that you specify as a parameter. The `uncompress()` method allows you to uncompress a compressed ByteArray in accordance with a compression algorithm. After calling `compress()` and `uncompress()`, the length of the byte array is set to the new length and the position property is set to the end.

The CompressionAlgorithm class defines constants that you can use to specify the compression algorithm. AIR supports both the deflate and zlib algorithms. The deflate compression algorithm is used in several compression formats, such as zlib, gzip, and some zip implementations. The zlib compressed data format is described at http://www.ietf.org/rfc/rfc1950.txt and the deflate compression algorithm is described at http://www.ietf.org/rfc/rfc1951.txt.

The following example compresses a ByteArray called `bytes` using the deflate algorithm:

```
bytes.compress(CompressionAlgorithm.DEFLATE);
```

The following example uncompresses a compressed ByteArray using the deflate algorithm:

```
bytes.uncompress(CompressionAlgorithm.DEFLATE);
```

Reading and writing objects

The `readObject()` and `writeObject()` methods read an object from and write an object to a ByteArray, encoded in serialized Action Message Format (AMF). AMF is a proprietary message protocol created by Adobe and used by various ActionScript 3.0 classes, including Netstream, NetConnection, NetStream, LocalConnection, and Shared Objects.

A one-byte type marker describes the type of the encoded data that follows. AMF uses the following 13 data types:

```
value-type = undefined-marker | null-marker | false-marker | true-marker | integer-type |
    double-type | string-type | xml-doc-type | date-type | array-type | object-type |
    xml-type | byte-array-type
```

The encoded data follows the type marker unless the marker represents a single possible value, such as null or true or false, in which case nothing else is encoded.

There are two versions of AMF: AMF0 and AMF3. AMF 0 supports sending complex objects by reference and allows endpoints to restore object relationships. AMF 3 improves AMF 0 by sending object traits and strings by reference, in addition to object references, and by supporting new data types that were introduced in ActionScript 3.0. The `ByteArray.objectEcoding` property specifies the version of AMF that is used to encode the object data. The flash.net.ObjectEncoding class defines constants for specifying the AMF version: `ObjectEncoding.AMF0` and `ObjectEncoding.AMF3`.

The following example calls `writeObject()` to write an XML object to a ByteArray, which it then compresses using the Deflate algorithm and writes to the `order` file on the desktop. The example uses a label to display the message “Wrote order file to desktop!” in the AIR window when it is finished.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
    creationComplete="init();">
<mx:Script>
    <![CDATA[

import flash.filesystem.*;
import flash.utils.ByteArray;

import mx.controls.Label;
private function init():void {

    var bytes:ByteArray = new ByteArray();
    var myLabel:Label = new Label();
    myLabel.move(150, 150);
    myLabel.width = 200;
    addChild(myLabel);

    var myXML:XML =
     <order>
         <item id='1'>
             <menuName>burger</menuName>
             <price>3.95</price>
         </item>
         <item id='2'>
             <menuName>fries</menuName>
             <price>1.45</price>
         </item>
    </order>

    // Write XML object to ByteArray
    bytes.writeObject(myXML);
    bytes.position = 0;//reset position to beginning
    bytes.compress(CompressionAlgorithm.DEFLATE);// compress ByteArray
    outFile("order", bytes);
    myLabel.text = "Wrote order file to desktop!";

} // end of init()

function outFile(fileName:String, data:ByteArray):void {
    var outFile:File = File.desktopDirectory; // dest folder is desktop
    outFile = outFile.resolvePath(fileName);  // name of file to write
    var outStream:FileStream = new FileStream();
    // open output file stream in WRITE mode
    outStream.open(outFile, FileMode.WRITE);
    // write out the file
    outStream.writeBytes(data, 0, data.length);
    // close it
    outStream.close();
}

    ]]>
</mx:Script>
</mx:WindowedApplication>
```

The `readObject()` method reads an object in serialized AMF from a ByteArray and stores it in an object of the specified type. The following example reads the `order` file from the desktop into a ByteArray (`inBytes`), uncompresses it, and calls `readObject()` to store it in the XML object `orderXML`. The example uses a `for each()` loop construct to add each node to a text area for display. The example also displays the value of the `objectEncoding` property along with a header for the contents of the `order` file.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
    creationComplete="init();">
<mx:Script>
        <![CDATA[

import flash.filesystem.*;
import flash.utils.ByteArray;

import mx.controls.TextArea;

private function init():void {

    var inBytes:ByteArray = new ByteArray();
    // define text area for displaying XML content
    var myTxt:TextArea = new TextArea();
    myTxt.width = 550;
    myTxt.height = 400;
    addChild(myTxt);
    //display objectEncoding and file heading
    myTxt.text = "Object encoding is: " + inBytes.objectEncoding + "\n\n" + "order file: 
\n\n";
    readFile("order", inBytes);

    inBytes.position = 0; // reset position to beginning
    inBytes.uncompress(CompressionAlgorithm.DEFLATE);
    inBytes.position = 0;//reset position to beginning
    // read XML Object
    var orderXML:XML = inBytes.readObject();

    //for each node in orderXML
    for each(var child:XML in orderXML) {
        // append child node to text area
        myTxt.text += child + "\n";
    }

} // end of init()

// read specified file into byte array
function readFile(fileName:String, data:ByteArray) {
    var inFile:File = File.desktopDirectory; // source folder is desktop
    inFile = inFile.resolvePath(fileName);  // name of file to read
    var inStream:FileStream = new FileStream();
    inStream.open(inFile, FileMode.READ);
    inStream.readBytes(data, 0, data.length);
    inStream.close();
}

        ]]>
    </mx:Script>

</mx:WindowedApplication>
```

ByteArray example: Reading a .zip file

This example demonstrates how to read a simple .zip file containing several files of different types. It does so by extracting relevant data from the metadata for each file, uncompressing each file into a ByteArray and writing the file to the desktop.

The general structure of a .zip file is based on the specification by PKWARE Inc., which is maintained at http://www.pkware.com/documents/casestudies/APPNOTE.TXT. First is a file header and file data for the first file in the .zip archive, followed by a file header and file data pair for each additional file. (The structure of the file header is described later.) Next, the .zip file optionally includes a data descriptor record (usually when the output zip file was created in memory rather than saved to a disk). Next are several additional optional elements: archive decryption header, archive extra data record, central directory structure, Zip64 end of central directory record, Zip64 end of central directory locator, and end of central directory record.

The code in this example is written to only parse zip files that do not contain folders and it does not expect data descriptor records. It ignores all information following the last file data.

The format of the file header for each file is as follows:

file header signature	4 bytes
required version	2 bytes
general-purpose bit flag	2 bytes
compression method	2 bytes (8=DEFLATE; 0=UNCOMPRESSED)
last modified file time	2 bytes
last modified file date	2 bytes
crc-32	4 bytes
compressed size	4 bytes
uncompressed size	4 bytes
file name length	2 bytes
extra field length	2 bytes
file name	variable
extra field	variable

Following the file header is the actual file data, which can be either compressed or uncompressed, depending on the compression method flag. The flag is 0 (zero) if the file data is uncompressed, 8 if the data is compressed using the DEFLATE algorithm, or another value for other compression algorithms.

The user interface for this example consists of a label and a text area (`taFiles`). The application writes the following information to the text area for each file it encounters in the .zip file: the file name, the compressed size, and the uncompressed size. The following MXML document defines the user interface for the application:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
creationComplete="init();">
    <mx:Script>
        <![CDATA[
            // The application code goes here
        ]]>
    </mx:Script>
    <mx:Form>
        <mx:FormItem label="Output">
            <mx:TextArea id="taFiles" width="320" height="150"/>
        </mx:FormItem>
    </mx:Form>
</mx:WindowedApplication>
```

The beginning of the program performs the following tasks:

- Imports the required classes

```
import flash.filesystem.*;
import flash.utils.ByteArray;
import flash.events.Event;
```

- Defines the `bytes` ByteArray

```
var bytes:ByteArray = new ByteArray();
```

- Defines variables to store metadata from the file header

```
// variables for reading fixed portion of file header
var fileName:String = new String();
var flNameLength:uint;
var xfldLength:uint;
var offset:uint;
var compSize:uint;
var uncompSize:uint;
var compMethod:int;
var signature:int;
```

- Defines File (`zfile`) and FileStream (`zStream`) objects to represent the .zip file, and specifies the location of the .zip file from which the files are extracted—a file named "HelloAIR.zip" in the desktop directory.

```
// File variables for accessing .zip file
var zfile:File = File.desktopDirectory.resolvePath("HelloAIR.zip");
var zStream:FileStream = new FileStream();
```

The program code starts in the `init()` method, which is called as the `creationComplete` handler for the root `mx:WindowedApplication` tag.

```
private function init():void
{
```

The program begins by opening the .zip file in READ mode.

```
    zStream.open(zfile, FileMode.READ);
```

It then sets the `endian` property of `bytes` to `LITTLE_ENDIAN` to indicate that the byte order of numeric fields has the least significant byte first.

```
    bytes.endian = Endian.LITTLE_ENDIAN;
```

Next, a `while()` statement begins a loop that continues until the current position in the file stream is greater than or equal to the size of the file.

```
    while (zStream.position < zfile.size)
    {
```

The first statement inside the loop reads the first 30 bytes of the file stream into the ByteArray `bytes`. The first 30 bytes make up the fixed-size part of the first file header.

```
        // read fixed metadata portion of local file header
        zStream.readBytes(bytes, 0, 30);
```

Next, the code reads an integer (`signature`) from the first bytes of the 30-byte header. The ZIP format definition specifies that the signature for every file header is the hexadecimal value `0x04034b50`; if the signature is different it means that the code has moved beyond the file portion of the .zip file and there are no more files to extract. In that case the code exits the `while` loop immediately rather than waiting for the end of the byte array.

```
        bytes.position = 0;
        signature = bytes.readInt();
        // if no longer reading data files, quit
        if (signature != 0x04034b50)
        {
            break;
        }
```

The next part of the code reads the header byte at offset position 8 and stores the value in the variable `compMethod`. This byte contains a value indicating the compression method that was used to compress this file. Several compression methods are allowed, but in practice nearly all .zip files use the DEFLATE compression algorithm. If the current file is compressed with DEFLATE compression, `compMethod` is 8; if the file is uncompressed, `compMethod` is 0.

```
        bytes.position = 8;
        compMethod = bytes.readByte();  // store compression method (8 == Deflate)
```

Following the first 30 bytes is a variable-length portion of the header that contains the file name and, possibly, an extra field. The variable `offset` stores the size of this portion. The size is calculated by adding the file name length and extra field length, read from the header at offsets 26 and 28.

```
        offset = 0;// stores length of variable portion of metadata
        bytes.position = 26;  // offset to file name length
        flNameLength = bytes.readShort();// store file name
        offset += flNameLength; // add length of file name
        bytes.position = 28;// offset to extra field length
        xfldLength = bytes.readShort();
        offset += xfldLength;// add length of extra field
```

Next the program reads the variable-length portion of the file header for the number of bytes stored in the `offset` variable.

```
        // read variable length bytes between fixed-length header and compressed file data
        zStream.readBytes(bytes, 30, offset);
```

The program reads the file name from the variable length portion of the header and displays it in the text area along with the compressed (zipped) and uncompressed (original) sizes of the file.

```
        bytes.position = 30;
        fileName = bytes.readUTFBytes(flNameLength); // read file name
        taFiles.text += fileName + "\n"; // write file name to text area
        bytes.position = 18;
        compSize = bytes.readUnsignedInt();  // store size of compressed portion
        taFiles.text += "\tCompressed size is: " + compSize + '\n';
        bytes.position = 22;  // offset to uncompressed size
        uncompSize = bytes.readUnsignedInt();  // store uncompressed size
        taFiles.text += "\tUncompressed size is: " + uncompSize + '\n';
```

The example reads the rest of the file from the file stream into `bytes` for the length specified by the compressed size, overwriting the file header in the first 30 bytes. The compressed size is accurate even if the file is not compressed because in that case the compressed size is equal to the uncompressed size of the file.

```
    // read compressed file to offset 0 of bytes; for uncompressed files
    // the compressed and uncompressed size is the same
    zStream.readBytes(bytes, 0, compSize);
```

Next, the example uncompresses the compressed file and calls the `outfile()` function to write it to the output file stream. It passes `outfile()` the file name and the byte array containing the file data.

```
        if (compMethod == 8) // if file is compressed, uncompress
        {
            bytes.uncompress(CompressionAlgorithm.DEFLATE);
        }
        outFile(fileName, bytes);   // call outFile() to write out the file
```

The closing braces indicate the end of the `while` loop and of the `init()` method and the application code, except for the `outFile()` method. Execution loops back to the beginning of the `while` loop and continues processing the next bytes in the .zip file—either extracting another file or ending processing of the .zip file if the last file has been processed.

```
    } // end of while loop
} // end of init() method
```

The `outfile()` function opens an output file in WRITE mode on the desktop, giving it the name supplied by the `filename` parameter. It then writes the file data from the `data` parameter to the output file stream (`outStream`) and closes the file.

```
private function outFile(fileName:String, data:ByteArray):void
{
    var outFile:File = File.desktopDirectory; // dest folder is desktop
    outFile = outFile.resolvePath(fileName);  // name of file to write
    var outStream:FileStream = new FileStream();
    // open output file stream in WRITE mode
    outStream.open(outFile, FileMode.WRITE);
    // write out the file
    outStream.writeBytes(data, 0, data.length);
    // close it
    outStream.close();
}
```

Chapter 21: Working with local SQL databases

Adobe AIR includes the capability of creating and working with local SQL databases. The runtime includes a SQL database engine with support for many standard SQL features, using the open source SQLite database system. A local SQL database can be used for storing local, persistent data. For instance, it can be used for application data, application user settings, documents, or any other type of data that you might want your application to save locally.

Contents

Quick Starts (Adobe AIR Developer Center)

- Working asynchronously with a local SQL database
- Working synchronously with a local SQL database

Language Reference

- SQLCollationType
- SQLColumnNameStyle
- SQLColumnSchema
- SQLConnection
- SQLError
- SQLErrorEvent
- SQLErrorOperation
- SQLEvent
- SQLIndexSchema
- SQLMode
- SQLResult
- SQLSchema
- SQLSchemaResult
- SQLStatement
- SQLTableSchema
- SQLTransactionLockType
- SQLTriggerSchema
- SQLUpdateEvent
- SQLViewSchema

More information

- Adobe AIR Developer Center for Flex (search for 'AIR SQL')

About local SQL databases

Adobe AIR includes a SQL-based relational database engine that runs within the runtime, with data stored locally in database files on the computer on which the AIR application runs (for example, on the computer's hard drive). Because the database runs and data files are stored locally, a database can be used by an AIR application regardless of whether a network connection is available. Thus, the runtime's local SQL database engine provides a convenient mechanism for storing persistent, local application data, particularly if you have experience with SQL and relational databases.

Contents

Uses for local SQL databases

The AIR local SQL database functionality can be used for any purpose for which you might want to store application data on a user's local computer. Adobe AIR includes several mechanisms for storing data locally, each of which has different advantages. The following are some possible uses for a local SQL database in your AIR application:

- For a data-oriented application (for example an address book), a database can be used to store the main application data.
- For a document-oriented application, where users create documents to save and possibly share, each document could be saved as a database file, in a user-designated location. (Note, however, that any AIR application would be able to open the database file, so a separate encryption mechanism would be recommended for potentially sensitive documents.)
- For a network-aware application, a database can be used to store a local cache of application data, or to store data temporarily when a network connection isn't available. You could create a mechanism for synchronizing the local database with the network data store.
- For any application, a database can be used to store individual users' application settings, such as user options or application information like window size and position.

About AIR databases and database files

An individual Adobe AIR local SQL database is stored as a single file in the computer's file system. The runtime includes the SQL database engine that manages creation and structuring of database files and manipulation and retrieval of data from a database file. The runtime does not specify how or where database data is stored on the file system; rather, each database is stored completely within a single file. You specify the location in the file system where the database file is stored. A single AIR application can access one or many separate databases (that is, separate database files). Because the runtime stores each database as a single file on the file system, you can locate your database as needed by the design of your application and file access constraints of the operating system. Each user can have a separate database file for their specific data, or a database file can be accessed by all application users on a single computer for shared data. Because the data is local to a single computer, data is not automatically shared among users on different computers. The local SQL database engine doesn't provide any capability to execute SQL statements against a remote or server-based database.

About relational databases

A relational database is a mechanism for storing (and retrieving) data on a computer. Data is organized into tables: rows represent records or items, and columns (sometimes called "fields") divide each record into individual values. For example, an address book application could contain a "friends" table. Each row in the table would represent a single friend stored in the database. The table's columns would represent data such as first name, last name, birth date, and so forth. For each friend row in the table, the database stores a separate value for each column.

Relational databases are designed to store complex data, where one item is associated with or related to items of another type. In a relational database, any data that has a one-to-many relationship—where a single record can be related to multiple records of a different type—should be divided among different tables. For example, suppose you want your address book application to store multiple phone numbers for each friend; this is a one-to-many relationship. The "friends" table would contain all the personal information for each friend. A separate "phone numbers" table would contain all the phone numbers for all the friends.

In addition to storing the data about friends and phone numbers, each table would need a piece of data to keep track of the relationship between the two tables—to match individual friend records with their phone numbers. This data is known as a primary key—a unique identifier that distinguishes each row in a table from other rows in that table. The primary key can be a "natural key," meaning it's one of the items of data that naturally distinguishes each record in a table. In the "friends" table, if you knew that none of your friends share a birth date, you could use the birth date column as the primary key (a natural key) of the "friends" table. If there is no natural key, you would create a separate primary key column such as a "friend id"—an artificial value that the application uses to distinguish between rows.

Using a primary key, you can set up relationships between multiple tables. For instance, suppose the "friends" table has a column "friend id" that contains a unique number for each row (each friend). The related "phone numbers" table can be structured with two columns: one with the "friend id" of the friend to whom the phone number belongs, and one with the actual phone number. That way, no matter how many phone numbers a single friend has, they can all be stored in the "phone numbers" table and can be linked to the related friend using the "friend id" primary key. When a primary key from one table is used in a related table to specify the connection between the records, the value in the related table is known as a foreign key. Unlike many databases, the AIR local database engine does not allow you to create foreign key constraints, which are constraints that automatically check that an inserted or updated foreign key value has a corresponding row in the primary key table. Nevertheless, foreign key relationships are an important part of the structure of a relational database, and foreign keys should be used when creating relationships between tables in your database.

About SQL

Structured Query Language (SQL) is used with relational databases to manipulate and retrieve data. SQL is a descriptive language rather than a procedural language. Instead of giving the computer instructions on how it should retrieve data, a SQL statement describes the set of data you want. The database engine determines how to retrieve that data.

The SQL language has been standardized by the American National Standards Institute (ANSI). The Adobe AIR local SQL database supports most of the SQL-92 standard. For specific descriptions of the SQL language supported in Adobe AIR, see the appendix "SQL support in local databases" in the Flex 3 Language Reference.

About SQL database classes

To work with local SQL databases in ActionScript 3.0, you use instances of these classes in the flash.data package:

Class	Description
flash.data.SQLConnection	Provides the means to create and open databases (database files), as well as methods for performing database-level operations and for controlling database transactions.
flash.data.SQLStatement	Represents a single SQL statement (a single query or command) that is executed on a database, including defining the statement text and setting parameter values.
flash.data.SQLResult	Provides a way to get information about or results from executing a statement, such as the result rows from a `SELECT` statement, the number of rows affected by an `UPDATE` or `DELETE` statement, and so forth.

To obtain schema information describing the structure of a database, you use these classes in the flash.data package:

Class	Description
flash.data.SQLSchemaResult	Serves as a container for database schema results generated by calling the `SQLConnection.loadSchema()` method.
flash.data.SQLTableSchema	Provides information describing a single table in a database.
flash.data.SQLViewSchema	Provides information describing a single view in a database.
flash.data.SQLIndexSchema	Provides information describing a single column of a table or view in a database.
flash.data.SQLTriggerSchema	Provides information describing a single trigger in a database.

Other classes in the flash.data package provide constants that are used with the SQLConnection class and the SQLColumnSchema class:

Class	Description
flash.data.SQLMode	Defines a set of constants representing the possible values for the `openMode` parameter of the `SQLConnection.open()` and `SQLConnection.openAsync()` methods.
flash.data.SQLColumnNameStyle	Defines a set of constants representing the possible values for the `SQLConnection.columnNameStyle` property.
flash.data.SQLTransactionLockType	Defines a set of constants representing the possible values for the option parameter of the `SQLConnection.begin()` method.
flash.data.SQLCollationType	Defines a set of constants representing the possible values for the `SQLColumnSchema.defaultCollationType` property and the `defaultCollationType` parameter of the `SQLColumnSchema()` constructor.

In addition, the following classes in the flash.events package represent the events (and supporting constants) that you use:

Class	Description
flash.data.SQLEvent	Defines the events that a SQLConnection or SQLStatement instance dispatches when any of its operations execute successfully. Each operation has an associated event type constant defined in the SQLEvent class.
flash.data.SQLErrorEvent	Defines the event that a SQLConnection or SQLStatement instance dispatches when any of its operations results in an error.
flash.data.SQLUpdateEvent	Defines the event that a SQLConnection instances dispatches when table data in one of its connected databases changes as a result of an `INSERT`, `UPDATE`, or `DELETE` SQL statement being executed.

Finally, the following classes in the flash.errors package provide information about database operation errors:

Class	Description
flash.data.SQLError	Provides information about a database operation error, including the operation that was being attempted and the cause of the failure.
flash.data.SQLErrorEvent	Defines a set of constants representing the possible values for the SQLError class's `operation` property, which indicates the database operation that resulted in an error.

About synchronous and asynchronous execution modes

When you're writing code to work with a local SQL database, you specify that database operations execution in one of two execution modes: asynchronous or synchronous execution mode. In general, the code examples show how to perform each operation in both ways, so that you can use the example that's most appropriate for your needs.

In asynchronous execution mode, you give the runtime an instruction and the runtime dispatches an event when your requested operation completes or fails. First you tell the database engine to perform an operation. The database engine does its work in the background while the application continues running. Finally, when the operation is completed (or when it fails) the database engine dispatches an event. Your code, triggered by the event, carries out subsequent operations. This approach has a significant benefit: the runtime performs the database operations in the background while the main application code continues executing. If the database operation takes a notable amount of time, the application continues to run. Most importantly, the user can continue to interact with it without the screen freezing. Nevertheless, asynchronous operation code can be more complex to write than other code. This complexity is usually in cases where multiple dependent operations must be divided up among various event listener methods.

Conceptually, it is simpler to code operations as a single sequence of steps—a set of synchronous operations—rather than a set of operations split into several event listener methods. In addition to asynchronous database operations, Adobe AIR also allows you to execute database operations synchronously. In synchronous execution mode, operations don't run in the background. Instead they run in the same execution sequence as all other application code. You tell the database engine to perform an operation. The code then pauses at that point while the database engine does its work. When the operation completes, execution continues with the next line of your code.

Whether operations execute asynchronously or synchronously is set at the SQLConnection level. Using a single database connection, you can't execute some operations or statements synchronously and others asynchronously. You specify whether a SQLConnection operates in synchronous or asynchronous execution mode by calling a SQLConnection method to open the database. If you call `SQLConnection.open()` the connection operates in synchronous execution mode, and if you call `SQLConnection.openAsync()` the connection operates in asynchronous execution mode. Once a SQLConnection instance is connected to a database using `open()` or `openAsync()`, it is fixed to synchronous or asynchronous execution mode unless you close and reopen the connection to the database.

Each execution mode has benefits. While most aspects of each mode are similar, there are some differences you'll want to keep in mind when working in each mode. For more information on these topics, and suggestions for working in each mode, see "Using synchronous and asynchronous database operations" on page 231.

Creating and modifying a database

Before your application can add or retrieve data, there must be a database with tables defined in it that your application can access. Described here are the tasks of creating a database and creating the data structure within a database. While these tasks are less frequently used than data insertion and retrieval, they are necessary for most applications.

Contents

Creating a database

To create a database file, you first create a SQLConnection instance. You call its `open()` method to open it in synchronous execution mode, or its `openAsync()` method to open it in asynchronous execution mode. The `open()` and `openAsync()` methods are used to open a connection to a database. If you pass a File instance that refers to a non-existent file location for the `reference` parameter (the first parameter), the `open()` or `openAsync()` method creates a database file at that file location and open a connection to the newly created database.

Whether you call the `open()` method or the `openAsync()` method to create a database, the database file's name can be any valid file name, with any file extension. If you call the `open()` or `openAsync()` method with `null` for the `reference` parameter, a new in-memory database is created rather than a database file on disk.

The following code listing shows the process of creating a database file (a new database) using asynchronous execution mode. In this case, the database file is saved in the application's storage directory, with the file name "DBSample.db":

```
import flash.data.SQLConnection;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;
import flash.filesystem.File;

var conn:SQLConnection = new SQLConnection();
conn.addEventListener(SQLEvent.OPEN, openHandler);
conn.addEventListener(SQLErrorEvent.ERROR, errorHandler);

var dbFile:File = File.applicationStorageDirectory.resolvePath("DBSample.db");

conn.openAsync(dbFile);

function openHandler(event:SQLEvent):void
{
    trace("the database was created successfully");
}

function errorHandler(event:SQLErrorEvent):void
{
    trace("Error message:", event.error.message);
    trace("Details:", event.error.details);
}
```

To execute operations synchronously, when you open a database connection with the SQLConnection instance, call the `open()` method. The following example shows how to create and open a SQLConnection instance that executes its operations synchronously:

```
import flash.data.SQLConnection;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;
import flash.filesystem.File;

var conn:SQLConnection = new SQLConnection();

var dbFile:File = File.applicationStorageDirectory.resolvePath("DBSample.db");

try
{
    conn.open(dbFile);
    trace("the database was created successfully");
}
catch (error:SQLError)
{
    trace("Error message:", error.message);
    trace("Details:", error.details);
}
```

Creating database tables

Creating a table in a database involves executing a SQL statement on that database, using the same process that you use to execute a SQL statement such as `SELECT`, `INSERT`, and so forth. To create a table, you use a `CREATE TABLE` statement, which includes definitions of columns and constraints for the new table. For more information about executing SQL statements, see "Working with SQL statements" on page 215.

The following example demonstrates creating a table named "employees" in an existing database file, using asynchronous execution mode. Note that this code assumes there is a SQLConnection instance named `conn` that is already instantiated and is already connected to a database.

```
import flash.data.SQLConnection;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn ...

var createStmt:SQLStatement = new SQLStatement();
createStmt.sqlConnection = conn;

var sql:String =
    "CREATE TABLE IF NOT EXISTS employees (" +
    "    empId INTEGER PRIMARY KEY AUTOINCREMENT, " +
    "    firstName TEXT, " +
    "    lastName TEXT, " +
    "    salary NUMERIC CHECK (salary > 0)" +
    ")";
createStmt.text = sql;

createStmt.addEventListener(SQLEvent.RESULT, createResult);
createStmt.addEventListener(SQLErrorEvent.ERROR, createError);

createStmt.execute();

function createResult(event:SQLEvent):void
{
    trace("Table created");
}

function createError(event:SQLErrorEvent):void
{
    trace("Error message:", event.error.message);
    trace("Details:", event.error.details);
}
```

The following example demonstrates how to create a table named “employees” in an existing database file, using synchronous execution mode. Note that this code assumes there is a SQLConnection instance named `conn` that is already instantiated and is already connected to a database.

```
import flash.data.SQLConnection;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn ...

var createStmt:SQLStatement = new SQLStatement();
createStmt.sqlConnection = conn;

var sql:String =
    "CREATE TABLE IF NOT EXISTS employees (" +
    "    empId INTEGER PRIMARY KEY AUTOINCREMENT, " +
    "    firstName TEXT, " +
    "    lastName TEXT, " +
    "    salary NUMERIC CHECK (salary > 0)" +
    ")";
createStmt.text = sql;

try
{
    createStmt.execute();
    trace("Table created");
}
catch (error:SQLError)
{
    trace("Error message:", error.message);
    trace("Details:", error.details);
}
```

Manipulating SQL database data

There are some common tasks that you perform when you’re working with local SQL databases. These tasks include connecting to a database, adding data to and retrieving data from tables in a database. There are also several issues you’ll want to keep in mind while performing these tasks, such as working with data types and handling errors.

Note that there are also several database tasks that are things you’ll deal with less frequently, but will often need to do before you can perform these more common tasks. For example, before you can connect to a database and retrieve data from a table, you’ll need to create the database and create the table structure in the database. Those less-frequent initial setup tasks are discussed in “Creating and modifying a database” on page 211.

You can choose to perform database operations asynchronously, meaning the database engine runs in the background and notifies you when the operation succeeds or fails by dispatching an event. You can also perform these operations synchronously. In that case the database operations are performed one after another and the entire application (including updates to the screen) waits for the operations to complete before executing other code. The examples in this section demonstrate how to perform the operations both asynchronously and synchronously. For more information on working in asynchronous or synchronous execution mode, see “Using synchronous and asynchronous database operations” on page 231.

Contents

Connecting to a database

Before you can perform any database operations, first open a connection to the database file. A SQLConnection instance is used to represent a connection to one or more databases. The first database that is connected using a SQLConnection instance is known as the “main” database. This database is connected using the `open()` method (for synchronous execution mode) or the `openAsync()` method (for asynchronous execution mode).

If you open a database using the asynchronous `openAsync()` operation, register for the SQLConnection instance’s `open` event in order to know when the `openAsync()` operation completes. Register for the SQLConnection instance’s `error` event to determine if the operation fails.

The following example shows how to open an existing database file for asynchronous execution. The database file is named “DBSample.db” and is located in the user’s application storage directory.

```
import flash.data.SQLConnection;
import flash.data.SQLMode;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;
import flash.filesystem.File;

var conn:SQLConnection = new SQLConnection();
conn.addEventListener(SQLEvent.OPEN, openHandler);
conn.addEventListener(SQLErrorEvent.ERROR, errorHandler);

var dbFile:File = File.applicationStorageDirectory.resolvePath("DBSample.db");

conn.openAsync(dbFile, SQLMode.UPDATE);

function openHandler(event:SQLEvent):void
{
    trace("the database opened successfully");
}

function errorHandler(event:SQLErrorEvent):void
{
    trace("Error message:", event.error.message);
    trace("Details:", event.error.details);
}
```

The following example shows how to open an existing database file for synchronous execution. The database file is named “DBSample.db” and is located in the user’s application storage directory.

```
import flash.data.SQLConnection;
import flash.data.SQLMode;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;
import flash.filesystem.File;

var conn:SQLConnection = new SQLConnection();

var dbFile:File = File.applicationStorageDirectory.resolvePath("DBSample.db");

try
{
    conn.open(dbFile, SQLMode.UPDATE);
    trace("the database opened successfully");
}
catch (error:SQLError)
{
    trace("Error message:", error.message);
    trace("Details:", error.details);
}
```

Notice that in the `openAsync()` method call in the asynchronous example, and the `open()` method call in the synchronous example, the second argument is the constant `SQLMode.UPDATE`. Specifying `SQLMode.UPDATE` for the second parameter (`openMode`) causes the runtime to dispatch an error if the specified file doesn't exist. If you pass `SQLMode.CREATE` for the `openMode` parameter (or if you leave the `openMode` parameter off), the runtime attempts to create a database file if the specified file doesn't exist. You can also specify `SQLMode.READ` for the `openMode` parameter to open an existing database in a read-only mode. In that case data can be retrieved from the database but no data can be added, deleted, or changed.

Working with SQL statements

An individual SQL statement (a query or command) is represented in the runtime as a SQLStatement object. Follow these steps to create and execute a SQL statement:

1. Create a SQLStatement instance.

The SQLStatement object represents the SQL statement in your application.

```
var selectData:SQLStatement = new SQLStatement();
```

2. Specify which database the query runs against.

To do this, set the SQLStatement object's `sqlConnection` property to the SQLConnection instance that's connected with the desired database.

```
// A SQLConnection named "conn" has been created previously
selectData.sqlConnection = conn;
```

3. Specify the actual SQL statement.

Create the statement text as a String and assign it to the SQLStatement instance's `text` property.

```
selectData.text = "SELECT col1, col2 FROM my_table WHERE col1 = :param1";
```

4. Define functions to handle the result of the execute operation (asynchronous execution mode only).

Use the `addEventListener()` method to register functions as listeners for the SQLStatement instance's `result` and `error` events.

```
// using listener methods and addEventListener();
selectData.addEventListener(SQLEvent.RESULT, resultHandler);
selectData.addEventListener(SQLErrorEvent.ERROR, errorHandler);

function resultHandler(event:SQLEvent):void
{
    // do something after the statement execution succeeds
}

function errorHandler(event:SQLErrorEvent):void
{
    // do something after the statement execution fails
}
```

Alternatively, you can specify listener methods using a Responder object. In that case you create the Responder instance and link the listener methods to it.

```
// using a Responder (flash.net.Responder)
var selectResponder = new Responder(onResult, onError);

function onResult(result:SQLResult):void
{
    // do something after the statement execution succeeds
}

function onError(error:SQLError):void
{
    // do something after the statement execution fails
}
```

5. If the statement text includes parameter definitions, assign values for those parameters.

To assign parameter values, use the SQLStatement instance's `parameters` associative array property.

```
selectData.parameters[":param1"] = 25;
```

6. Execute the SQL statement.

Call the SQLStatement instance's `execute()` method.

```
// using synchronous execution mode
// or listener methods in asynchronous execution mode
selectData.execute();
```

Additionally, if you're using a Responder instead of event listeners in asynchronous execution mode, pass the Responder instance to the `execute()` method.

```
// using a Responder in asynchronous execution mode
selectData.execute(-1, selectResponder);
```

For specific examples that demonstrate these steps, see the following topics:

- "Retrieving data from a database" on page 219
- "Inserting data" on page 224
- "Changing or deleting data" on page 227

Using parameters in statements

A SQL statement parameter allows you to create a reusable SQL statement. When you use statement parameters, values within the statement can change (such as values being added in an INSERT statement) but the basic statement text remains unchanged. This provides performance benefits as well as making it easier to code an application.

Contents

Understanding statement parameters

Frequently an application uses a single SQL statement multiple times in an application, with slight variation. For example, consider an inventory-tracking application where a user can add new inventory items to the database. The application code that adds an inventory item to the database executes a SQL `INSERT` statement that actually adds the data to the database. However, each time the statement is executed there is a slight variation. Specifically, the actual values that are inserted in the table are different because they are specific to the inventory item being added.

In cases where you have a SQL statement that’s used multiple times with different values in the statement, the best approach is to use a SQL statement that includes parameters rather than literal values in the SQL text. A parameter is a placeholder in the statement text that is replaced with an actual value each time the statement is executed. To use parameters in a SQL statement, you create the SQLStatement instance as usual. For the actual SQL statement assigned to the `text` property, use parameter placeholders rather than literal values. You then define the value for each parameter by setting the value of an element in the SQLStatement instance’s `parameters` property. The parameters property is an associative array, so you set a particular value using the following syntax:

```
statement.parameters[parameter_identifier] = value;
```

The *`parameter_identifier`* is a string if you’re using a named parameter, or an integer index if you’re using an unnamed parameter.

Using named parameters

A parameter can be a named parameter. A named parameter has a specific name that the database uses to match the parameter value to its placeholder location in the statement text. A parameter name consists of the “:” or “@” character followed by a name, as in the following examples:

```
:itemName
@firstName
```

The following code listing demonstrates the use of named parameters:

```
var sql:String =
    "INSERT INTO inventoryItems (name, productCode)" +
    "VALUES (:name, :productCode)";
var addItemStmt:SQLStatement = new SQLStatement();
addItemStmt.sqlConnection = conn;
addItemStmt.text = sql;

// set parameter values
addItemStmt.parameters[":name"] = "Item name";
addItemStmt.parameters[":productCode"] = "12345";

addItemStmt.execute();
```

Using unnamed parameters

As an alternative to using named parameters, you can also use unnamed parameters. To use an unnamed parameter you denote a parameter in a SQL statement using a “?” character. Each parameter is assigned a numeric index, according to the order of the parameters in the statement, starting with index 0 for the first parameter. The following example demonstrates a version of the previous example, using unnamed parameters:

```
var sql:String =
    "INSERT INTO inventoryItems (name, productCode)" +
    "VALUES (?, ?)";
var addItemStmt:SQLStatement = new SQLStatement();
```

```
addItemStmt.sqlConnection = conn;
addItemStmt.text = sql;

// set parameter values
addItemStmt.parameters[0] = "Item name";
addItemStmt.parameters[1] = "12345";

addItemStmt.execute();
```

Benefits of using parameters

Using parameters in a SQL statement provides several benefits:

Better performance A SQLStatement instance that uses parameters can execute more efficiently compared to one that dynamically creates the SQL text each time it executes. The performance improvement is because the statement is prepared a single time and can then be executed multiple times using different parameter values, without needing to recompile the SQL statement.

Explicit data typing Parameters are used to allow for typed substitution of values that are unknown at the time the SQL statement is constructed. The use of parameters is the only way to guarantee the storage class for a value passed in to the database. When parameters are not used, the runtime attempts to convert all values from their text representation to a storage class based on the associated column's type affinity. For more information on storage classes and column affinity, see the section “Data type support” in the appendix “SQL support in local databases” in the Flex 3 Language Reference.

Greater security The use of parameters helps prevent a malicious technique known as a SQL injection attack. In a SQL injection attack, a user enters SQL code in a user-accessible location (for example, a data entry field). If application code constructs a SQL statement by directly concatenating user input into the SQL text, the user-entered SQL code is executed against the database. The following listing shows an example of concatenating user input into SQL text. **Do not use this technique**:

```
// assume the variables "username" and "password"
// contain user-entered data
var sql:String =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = '" + username + "' " +
    "    AND password = '" + password + "'";
var statement:SQLStatement = new SQLStatement();
statement.text = sql;
```

Using statement parameters instead of concatenating user-entered values into a statement's text prevents a SQL injection attack. SQL injection can’t happen because the parameter values are treated explicitly as substituted values, rather than becoming part of the literal statement text. The following is the recommended alternative to the previous listing:

```
// assume the variables "username" and "password"
// contain user-entered data
var sql:String =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = :username " +
    "    AND password = :password";
var statement:SQLStatement = new SQLStatement();
statement.text = sql;

// set parameter values
statement.parameters[":username"] = username;
statement.parameters[":password"] = password;
```

Retrieving data from a database

Retrieving data from a database involves two steps. First, you execute a SQL `SELECT` statement, describing the set of data you want from the database. Next, you access the retrieved data and display or manipulate it as needed by your application.

Contents

Executing a SELECT statement

To retrieve existing data from a database, you use a SQLStatement instance. Assign the appropriate SQL `SELECT` statement to the instance's `text` property, then call its `execute()` method. For details on the syntax of the `SELECT` statement, see the appendix "SQL support in local databases" in the Flex 3 Language Reference.

The following example demonstrates executing a `SELECT` statement to retrieve data from a table named "products," using asynchronous execution mode:

```
var selectStmt:SQLStatement = new SQLStatement();

// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;

selectStmt.text = "SELECT itemId, itemName, price FROM products";

// The resultHandler and errorHandler are listener methods are
// described in a subsequent code listing
selectStmt.addEventListener(SQLEvent.RESULT, resultHandler);
selectStmt.addEventListener(SQLErrorEvent.ERROR, errorHandler);

selectStmt.execute();
```

The following example demonstrates executing a `SELECT` statement to retrieve data from a table named "products," using asynchronous execution mode:

```
var selectStmt:SQLStatement = new SQLStatement();

// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;

selectStmt.text = "SELECT itemId, itemName, price FROM products";

// This try..catch block is fleshed out in
// a subsequent code listing
try
{
    selectStmt.execute();
    // accessing the data is shown in a subsequent code listing
}
catch (error:SQLError)
{
    // error handling is shown in a subsequent code listing
}
```

In asynchronous execution mode, when the statement finishes executing, the SQLStatement instance dispatches a `result` event (`SQLEvent.RESULT`) indicating that the statement was run successfully. Alternatively, if a Responder object is passed as an argument in the `execute()` call, the Responder object's result handler function is called. In synchronous execution mode, execution pauses until the `execute()` operation completes, then continues on the next line of code.

Accessing SELECT statement result data

Once the `SELECT` statement has finished executing, the next step is to access the data that was retrieved. Each row of data in the `SELECT` result set becomes an Object instance. That object has properties whose names match the result set's column names. The properties contain the values from the result set's columns. For example, suppose a `SELECT` statement specifies a result set with three columns named "itemId," "itemName," and "price." For each row in the result set, an Object instance is created with properties named `itemId`, `itemName`, and `price`. Those properties contain the values from their respective columns.

The following code listing continues the previous code listing for retrieving data in asynchronous execution mode. It shows how to access the retrieved data within the result event listener method.

```
function resultHandler(event:SQLEvent):void
{
    var result:SQLResult = selectStmt.getResult();
    var numResults:int = result.data.length;
    for (var i:int = 0; i < numResults; i++)
    {
        var row:Object = result.data[i];
        var output:String = "itemId: " + row.itemId;
        output += "; itemName: " + row.itemName;
        output += "; price: " + row.price;
        trace(output);
    }
}

function errorHandler(event:SQLErrorEvent):void
{
    // Information about the error is available in the
    // event.error property, which is an instance of
    // the SQLError class.
}
```

The following code listing expands on the previous code listing for retrieving data in synchronous execution mode. It expands the `try..catch` block in the previous synchronous execution example, showing how to access the retrieved data.

```
try
{
    selectStmt.execute();

    var result:SQLResult = selectStmt.getResult();
    var numResults:int = result.data.length;
    for (var i:int = 0; i < numResults; i++)
    {
        var row:Object = result.data[i];
        var output:String = "itemId: " + row.itemId;
        output += "; itemName: " + row.itemName;
        output += "; price: " + row.price;
        trace(output);
    }
}
catch (error:SQLError)
{
    // Information about the error is available in the
    // error variable, which is an instance of
    // the SQLError class.
}
```

As the preceding code listings show, the result objects are contained in an array that is available as the `data` property of a SQLResult instance. If you're using asynchronous execution with an event listener, to retrieve that SQLResult instance you call the SQLStatement instance's `getResult()` method. If you specify a Responder argument in the `execute()` call, the SQLResult instance is passed to the result handler function as an argument. In synchronous execution mode, you call the SQLStatement instance's `getResult()` method any time after the `execute()` method call. In any case, once you have the SQLResult object you can access the result rows using the `data` array property.

The following code listing defines a SQLStatement instance whose text is a `SELECT` statement. The statement retrieves rows containing the `firstName` and `lastName` column values of all the rows of a table named `employees`. This example uses asynchronous execution mode. When the execution completes, the `selectResult()` method is called, and the resulting rows of data are accessed using `SQLStatement.getResult()` and displayed using the `trace()` method. Note that this listing assumes there is a SQLConnection instance named `conn` that has already been instantiated and is already connected to the database. It also assumes that the "employees" table has already been created and populated with data.

```
import flash.data.SQLConnection;
import flash.data.SQLResult;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var selectStmt:SQLStatement = new SQLStatement();
selectStmt.sqlConnection = conn;

// define the SQL text
var sql:String =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;

// register listeners for the result and error events
selectStmt.addEventListener(SQLEvent.RESULT, selectResult);
selectStmt.addEventListener(SQLErrorEvent.ERROR, selectError);

// execute the statement
selectStmt.execute();

function selectResult(event:SQLEvent):void
{
    // access the result data
    var result:SQLResult = selectStmt.getResult();
    var numRows:int = result.data.length;
    for (var i:int = 0; i < numRows; i++)
    {
        var output:String = "";
        for (var columnName:String in result.data[i])
        {
            output += columnName + ": " + result.data[i][columnName] + "; ";
        }
        trace("row[" + i.toString() + "]\t", output);
    }
}

function selectError(event:SQLErrorEvent):void
{
    trace("Error message:", event.error.message);
    trace("Details:", event.error.details);
}
```

The following code listing demonstrates the same techniques as the preceding one, but uses synchronous execution mode. The example defines a SQLStatement instance whose text is a `SELECT` statement. The statement retrieves rows containing the `firstName` and `lastName` column values of all the rows of a table named `employees`. The resulting rows of data are accessed using `SQLStatement.getResult()` and displayed using the `trace()` method. Note that this listing assumes there is a SQLConnection instance named `conn` that has already been instantiated and is already connected to the database. It also assumes that the "employees" table has already been created and populated with data.

```
import flash.data.SQLConnection;
import flash.data.SQLResult;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var selectStmt:SQLStatement = new SQLStatement();
selectStmt.sqlConnection = conn;

// define the SQL text
var sql:String =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;

try
{
    // execute the statement
    selectStmt.execute();

    // access the result data
    var result:SQLResult = selectStmt.getResult();
    var numRows:int = result.data.length;
    for (var i:int = 0; i < numRows; i++)
    {
        var output:String = "";
        for (var columnName:String in result.data[i])
        {
            output += columnName + ": " + result.data[i][columnName] + "; ";
        }
        trace("row[" + i.toString() + "]\t", output);
    }
}
catch (error:SQLError)
{
    trace("Error message:", error.message);
    trace("Details:", error.details);
}
```

Defining the data type of SELECT result data

By default, each row returned by a `SELECT` statement is created as an Object instance with properties named for the result set's column names and with the value of each column as the value of its associated property. However, before executing a SQL `SELECT` statement, you can set the `itemClass` property of the SQLStatement instance to a class. By setting the `itemClass` property, each row returned by the `SELECT` statement is created as an instance of the designated class. The runtime assigns result column values to property values by matching the column names in the `SELECT` result set to the names of the properties in the `itemClass` class.

Any class assigned as an `itemClass` property value must have a constructor that does not require any parameters. In addition, the class must have a single property for each column returned by the `SELECT` statement. It is considered an error if a column in the `SELECT` list does not have a matching property name in the `itemClass` class.

Retrieving SELECT results in parts

By default, a `SELECT` statement execution retrieves all the rows of the result set at one time. Once the statement completes, you usually process the retrieved data in some way, such as creating objects or displaying the data on the screen. If the statement returns a large number of rows, processing all the data at once can be demanding for the computer, which in turn will cause the user interface to not redraw itself.

You can improve the perceived performance of your application by instructing the runtime to return a specific number of result rows at a time. Doing so causes the initial result data to return more quickly. It also allows you to divide the result rows into sets, so that the user interface is updated after each set of rows is processed. Note that it's only practical to use this technique in asynchronous execution mode.

To retrieve `SELECT` results in parts, specify a value for the `SQLStatement.execute()` method's first parameter (the `prefetch` parameter). The `prefetch` parameter indicates the number of rows to retrieve the first time the statement executes. When you call a SQLStatement instance's `execute()` method, specify a `prefetch` parameter value and only that many rows will be retrieved:

```
var stmt:SQLStatement = new SQLStatement();
stmt.sqlConnection = conn;
stmt.text = "SELECT ...";
stmt.addEventListener(SQLEvent.RESULT, selectResult);
stmt.execute(20); // only the first 20 rows (or fewer) are returned
```

The statement dispatches the `result` event, indicating that the first set of result rows is available. The resulting SQLResult instance's `data` property contains the rows of data, and its `complete` property indicates whether there are additional result rows to retrieve. To retrieve additional result rows, call the SQLStatement instance's `next()` method. Like the `execute()` method, the `next()` method's first parameter is used to indicate how many rows to retrieve the next time the result event is dispatched.

```
function selectResult(event:SQLEvent):void
{
    var result:SQLResult = stmt.getResult();
    if (result.data != null)
    {
        // ... loop through the rows or perform other processing ...
        if (!result.complete)
        {
            stmt.next(20); // retrieve the next 20 rows
        }
        else
        {
            stmt.removeEventListener(SQLEvent.RESULT, selectResult);
        }
    }
}
```

The SQLStatement dispatches a `result` event each time the `next()` method returns a subsequent set of result rows. Consequently, the same listener function can be used to continue processing results (from `next()` calls) until all the rows are retrieved.

For more information, see the language reference descriptions for the `SQLStatement.execute()` method (the `prefetch` parameter description) and the `SQLStatement.next()` method.

Inserting data

Retrieving data from a database involves executing a SQL `INSERT` statement. Once the statement has finished executing, you can access the primary key for the newly inserted row if the key was generated by the database.

Contents

Executing an INSERT statement

To add data to a table in a database, you create and execute a SQLStatement instance whose text is a SQL `INSERT` statement.

The following example uses a SQLStatement instance to add a row of data to the already-existing employees table. This example demonstrates inserting data using asynchronous execution mode. Note that this listing assumes that there is a SQLConnection instance named `conn` that has already been instantiated and is already connected to a database. It also assumes that the "employees" table has already been created.

```
import flash.data.SQLConnection;
import flash.data.SQLResult;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var insertStmt:SQLStatement = new SQLStatement();
insertStmt.sqlConnection = conn;

// define the SQL text
var sql:String =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;

// register listeners for the result and failure (status) events
insertStmt.addEventListener(SQLEvent.RESULT, insertResult);
insertStmt.addEventListener(SQLErrorEvent.ERROR, insertError);

// execute the statement
insertStmt.execute();

function insertResult(event:SQLEvent):void
{
    trace("INSERT statement succeeded");
}

function insertError(event:SQLErrorEvent):void
{
    trace("Error message:", event.error.message);
    trace("Details:", event.error.details);
}
```

The following example adds a row of data to the already-existing employees table, using synchronous execution mode. Note that this listing assumes that there is a SQLConnection instance named `conn` that has already been instantiated and is already connected to a database. It also assumes that the "employees" table has already been created.

```
import flash.data.SQLConnection;
import flash.data.SQLResult;
import flash.data.SQLStatement;
import flash.events.SQLErrorEvent;
import flash.events.SQLEvent;

// ... create and open the SQLConnection instance named conn ...

// create the SQL statement
var insertStmt:SQLStatement = new SQLStatement();
insertStmt.sqlConnection = conn;

// define the SQL text
var sql:String =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;

try
{
    // execute the statement
    insertStmt.execute();
    trace("INSERT statement succeeded");
}
catch (error:SQLError)
{
    trace("Error message:", error.message);
    trace("Details:", error.details);
}
```

Retrieving a database-generated primary key of an inserted row

Often after inserting a row of data into a table, your code needs to know a database-generated primary key or row identifier value for the newly inserted row. For example, once you insert a row in one table, you might want to add rows in a related table. In that case you would want to insert the primary key value as a foreign key in the related table. The primary key of a newly inserted row can be retrieved using the SQLResult object generated by the statement execution. This is the same object that's used to access result data after a `SELECT` statement is executed. As with any SQL statement, when the execution of an `INSERT` statement completes the runtime creates a SQLResult instance. You access the SQLResult instance by calling the SQLStatement object's `getResult()` method if you're using an event listener or if you're using synchronous execution mode. Alternatively, if you're using asynchronous execution mode and you pass a Responder instance to the execute() call, the SQLResult instance is passed as an argument to the result handler function. In any case, the SQLResult instance has a property, `lastInsertRowID`, that contains the row identifier of the most-recently inserted row if the executed SQL statement is an `INSERT` statement.

The following example demonstrates accessing the primary key of an inserted row in asynchronous execution mode:

```
insertStmt.text = "INSERT INTO ...";
insertStmt.addEventListener(SQLEvent.RESULT, resultHandler);
insertStmt.execute();

private function resultHandler(event:SQLEvent):void
```

```
{
    // get the primary key
    var result:SQLResult = insertStmt.getResult();
    var primaryKey:Number = result.lastInsertRowID;
    // do something with the primary key
}
```

The following example demonstrates accessing the primary key of an inserted row in synchronous execution mode:

```
insertStmt.text = "INSERT INTO ...";
insertStmt.addEventListener(SQLEvent.RESULT, resultHandler);
try
{
    insertStmt.execute();

    // get the primary key
    var result:SQLResult = insertStmt.getResult();
    var primaryKey:Number = result.lastInsertRowID;
    // do something with the primary key
}
catch (error:SQLError)
{
    // respond to the error
}
```

Note that the row identifier may or may not be the value of the column that is designated as the primary key column in the table definition, according to the following rule:

- If the table is defined with a primary key column whose affinity (column data type) is `INTEGER`, the `lastInsertRowID` property contains the value that was inserted into that row (or the value generated by the runtime if it's an `AUTOINCREMENT` column).
- If the table is defined with multiple primary key columns (a composite key) or with a single primary key column whose affinity is not `INTEGER`, behind the scenes the database generates a row identifier value for the row. That generated value is the value of the `lastInsertRowID` property.
- The value is always the row identifier of the most-recently inserted row. If an `INSERT` statement causes a trigger to fire which in turn inserts a row, the `lastInsertRowID` property contains the row identifier of the last row inserted by the trigger rather than the row created by the `INSERT` statement. Consequently, if you want to have an explicitly defined primary key column whose value is available after an `INSERT` command through the `SQLResult.lastInsertRowID` property, the column must be defined as an `INTEGER PRIMARY KEY` column. Note, however, that even if your table does not include an explicit `INTEGER PRIMARY KEY` column, it is equally acceptable to use the database-generated row identifier as a primary key for your table in the sense of defining relationships with related tables. The row identifier column value is available in any SQL statement by using one of the special column names `ROWID`, `_ROWID_`, or `OID`. You can create a foreign key column in a related table and use the row identifier value as the foreign key column value just as you would with an explicitly declared `INTEGER PRIMARY KEY` column. In that sense, if you are using an arbitrary primary key rather than a natural key, and as long as you don't mind the runtime generating the primary key value for you, it makes little difference whether you use an `INTEGER PRIMARY KEY` column or the system-generated row identifier as a table's primary key for defining a foreign key relationship with between two tables.

For more information about primary keys and generated row identifiers, see the sections titled "CREATE TABLE" and "Expressions" in the appendix "SQL support in local databases" in the Flex 3 Language Reference.

Changing or deleting data

The process for executing other data manipulation operations is identical to the process used to execute a SQL `SELECT` or `INSERT` statement. Simply substitute a different SQL statement in the SQLStatement instance's `text` property:

- To change existing data in a table, use an `UPDATE` statement.
- To delete one or more rows of data from a table, use a `DELETE` statement.

For descriptions of these statements, see the appendix "SQL support in local databases" in the Flex 3 Language Reference.

Working with multiple databases

Use the `SQLConnection.attach()` method to open a connection to an additional database on a SQLConnection instance that already has an open database. You give the attached database a name using the name parameter in the `attach()` method call. When writing statements to manipulate that database, you can then use that name in a prefix (using the form `database-name.table-name`) to qualify any table names in your SQL statements, indicating to the runtime that the table can be found in the named database.

You can execute a single SQL statement that includes tables from multiple databases that are connected to the same SQLConnection instance. If a transaction is created on the SQLConnection instance, that transaction applies to all SQL statements that are executed using the SQLConnection instance. This is true regardless of which attached database the statement runs on.

Alternatively, you can also create multiple SQLConnection instances in an application, each of which is connected to one or multiple databases. However, if you do use multiple connections to the same database keep in mind that a database transaction isn't shared across SQLConnection instances. Consequently, if you connect to the same database file using multiple SQLConnection instances, you can't rely on both connections' data changes being applied in the expected manner. For example, if two `UPDATE` or `DELETE` statements are run against the same database through different SQLConnection instances, and an application error occurs after one operation takes place, the database data could be left in an intermediate state that would not be reversible and might affect the integrity of the database (and consequently the application).

Handling database errors

In general, database error handling is like other runtime error handling. You should write code that is prepared for errors that may occur, and respond to the errors rather than leave it up to the runtime to do so. In a general sense, the possible database errors can be divided into three categories: connection errors, SQL syntax errors, and constraint errors.

Contents

Connection errors

Most database errors are connection errors, and they can occur during any operation. Although there are strategies for preventing connection errors, there is rarely a simple way to gracefully recover from a connection error if the database is a critical part of your application.

Most connection errors have to do with how the runtime interacts with the operating system, the file system, and the database file. For example, a connection error occurs if the user doesn't have permission to create a database file in a particular location on the file system. The following strategies help to prevent connection errors:

Use user-specific database files Rather than using a single database file for all users who use the application on a single computer, give each user their own database file. The file should be located in a directory that's associated with the user's account. For example, it could be in the application's storage directory, the user's documents folder, the user's desktop, and so forth.

Consider different user types Test your application with different types of user accounts, on different operating systems. Don't assume that the user has administrator permission on the computer. Also, don't assume that the individual who installed the application is the user who's running the application.

Consider various file locations If you allow a user to specify where to save a database file or select a file to open, consider the possible file locations that the users might use. In addition, consider defining limits on where users can store (or from where they can open) database files. For example, you might only allow users to open files that are within their user account's storage location.

If a connection error occurs, it most likely happens on the first attempt to create or open the database. This means that the user is unable to do any database-related operations in the application. For certain types of errors, such as read-only or permission errors, one possible recovery technique is to copy the database file to a different location. The application can copy the database file to a different location where the user does have permission to create and write to files, and use that location instead.

Syntax errors

A syntax error occurs when a SQL statement is incorrectly formed, and the application attempts to execute the statement. Because local database SQL statements are created as strings, compile-time SQL syntax checking is not possible. All SQL statements must be executed to check their syntax. Use the following strategies to prevent SQL syntax errors:

Test all SQL statements thoroughly If possible, while developing your application test your SQL statements separately before encoding them as statement text in the application code. In addition, use a code-testing approach such as unit testing to create a set of tests that exercise every possible option and variation in the code.

Use statement parameters and avoid concatenating (dynamically generating) SQL Using parameters, and avoiding dynamically built SQL statements, means that the same SQL statement text is used each time a statement is executed. Consequently, it's much easier to test your statements and limit the possible variation. If you must dynamically generate a SQL statement, keep the dynamic parts of the statement to a minimum. Also, carefully validate any user input to make sure it won't cause syntax errors.

To recover from a syntax error, an application would need complex logic to be able to examine a SQL statement and correct its syntax. By following the previous guidelines for preventing syntax errors, your code can identify any potential run-time sources of SQL syntax errors (such as user input used in a statement). To recover from a syntax error, provide guidance to the user. Indicate what to correct to make the statement execute properly.

Constraint errors

Constraint errors occur when an `INSERT` or `UPDATE` statement attempts to add data to a column. The error happens if the new data violates one of the defined constraints for the table or column. The set of possible constraints includes:

Unique constraint Indicates that across all the rows in a table, there cannot be duplicate values in one column. Alternatively, when multiple columns are combined in a unique constraint, the combination of values in those columns must not be duplicated. In other words, in terms of the specified unique column or columns, each row must be distinct.

Primary key constraint In terms of the data that a constraint allows and doesn't allow, a primary key constraint is identical to a unique constraint.

Not null constraint Specifies that a single column cannot store a `NULL` value and consequently that in every row, that column must have a value.

Check constraint Allows you to specify an arbitrary constraint on one or more tables. A common check constraint is a rule that define that a column's value must be within certain bounds (for example, that a numeric column's value must be larger than 0). Another common type of check constraint specifies relationships between column values (for example, that a column's value must be different from the value of another column in the same row).

Data type (column affinity) constraint The runtime enforces the data type of columns' values, and an error occurs if an attempt is made to store a value of the incorrect type in a column. However, in many conditions values are converted to match the column's declared data type. See "Working with database data types" on page 230 for more information.

The runtime does not enforce constraints on foreign key values. In other words, foreign key values aren't required to match an existing primary key value.

In addition to the predefined constraint types, the runtime SQL engine supports the use of triggers. A trigger is similar to an event handler. It is a predefined set of instructions that are carried out when a certain action happens. For example, a trigger could be defined that runs when data is inserted into or deleted from a particular table. One possible use of a trigger is to examine data changes and cause an error to occur if specified conditions aren't met. Consequently, a trigger can serve the same purpose as a constraint, and the strategies for preventing and recovering from constraint errors also apply to trigger-generated errors. However, the error id for trigger-generated errors is different from the error id for constraint errors.

The set of constraints that apply to a particular table is determined while you're designing an application. Consciously designing constraints makes it easier to design your application to prevent and recover from constraint errors. However, constraint errors are difficult to systematically predict and prevent. Prediction is difficult because constraint errors don't appear until application data is added. Constraint errors occur with data that is added to a database after it's created. These errors are often a result of the relationship between new data and data that already exists in the database. The following strategies can help you avoid many constraint errors:

Carefully plan database structure and constraints The purpose of constraints is to enforce application rules and help protect the integrity of the database's data. When you're planning your application, consider how to structure your database to support your application. As part of that process, identify rules for your data, such as whether certain values are required, whether a value has a default, whether duplicate values are allowed, and so forth. Those rules guide you in defining database constraints.

Explicitly specify column names An `INSERT` statement can be written without explicitly specifying the columns into which values are to be inserted, but doing so is an unnecessary risk. By explicitly naming the columns into which values are to be inserted, you can allow for automatically generated values, columns with default values, and columns that allow `NULL` values. In addition, by doing so you can ensure that all `NOT NULL` columns have an explicit value inserted.

Use default values Whenever you specify a `NOT NULL` constraint for a column, if at all possible specify a default value in the column definition. Application code can also provide default values. For example, your code can check if a String variable is `null` and assign it a value before using it to set a statement parameter value.

Validate user-entered data Check user-entered data ahead of time to make sure that it obeys limits specified by constraints, especially in the case of `NOT NULL` and `CHECK` constraints. Naturally, a `UNIQUE` constraint is more difficult to check for because doing so would require executing a `SELECT` query to determine whether the data is unique.

Use triggers You can write a trigger that validates (and possibly replaces) inserted data or takes other actions to correct invalid data. This validation and correction can prevent a constraint error from occurring.

In many ways constraint errors are more difficult to prevent than other types of errors. Fortunately, there are several strategies to recover from constraint errors in ways that don't make the application unstable or unusable:

Use conflict algorithms When you define a constraint on a column, and when you create an `INSERT` or `UPDATE` statement, you have the option of specifying a conflict algorithm. A conflict algorithm defines the action the database takes when a constraint violation occurs. There are several possible actions the database engine can take. The database engine can end a single statement or a whole transaction. It can ignore the error. It can even remove old data and replace it with the data that the code is attempting to store. For more information see the section "ON CONFLICT (conflict algorithms)" in the appendix "SQL support in local databases" in the Flex 3 Language Reference.

Provide corrective feedback The set of constraints that can affect a particular SQL command can be identified ahead of time. Consequently, you can anticipate constraint errors that a statement could cause. With that knowledge, you can build application logic to respond to a constraint error. For example, suppose an application includes a data entry form for entering new products. If the product name column in the database is defined with a `UNIQUE` constraint, the action of inserting a new product row in the database could cause a constraint error. Consequently, the application is designed to anticipate a constraint error. When the error happens, the application alerts the user, indicating that the specified product name is already in use and asking the user to choose a different name. Another possible response is to allow the user to view information about the other product with the same name.

Working with database data types

When a table is created in a database, the SQL statement for creating the table defines the affinity, or data type, for each column in the table. Although affinity declarations can be omitted, it's a good idea to explicitly declare column affinity in your `CREATE TABLE` SQL statements.

As a general rule, any object that you store in a database using an `INSERT` statement is returned as an instance of the same data type when you execute a `SELECT` statement. However, the data type of the retrieved value can be different depending on the affinity of the database column in which the value is stored. When a value is stored in a column, if its data type doesn't match the column's affinity, the database attempts to convert the value to match the column's affinity. For example, if a database column is declared with `NUMERIC` affinity, the database attempts to convert inserted data into a numeric storage class (`INTEGER` or `REAL`) before storing the data. The database throws an error if the data can't be converted. According to this rule, if the String "12345" is inserted into a `NUMERIC` column, the database automatically converts it to the integer value 12345 before storing it in the database. When it's retrieved with a `SELECT` statement, the value is returned as an instance of a numeric data type (such as Number) rather than as a String instance.

The best way to avoid undesirable data type conversion is to follow two rules. First, define each column with the affinity that matches the type of data that it is intended to store. Next, only insert values whose data type matches the defined affinity. Following these rules provides two benefits. When you insert the data it isn't converted unexpectedly (possibly losing its intended meaning as a result). In addition, when you retrieve the data it is returned with its original data type.

For more information about the available column affinity types and using data types in SQL statements, see the section "Data type support" in the appendix "SQL support in local databases" in the Flex 3 Language Reference.

Using synchronous and asynchronous database operations

Previous sections have described common database operations such as retrieving, inserting, updating, and deleting data, as well as creating a database file and tables and other objects within a database. The examples have demonstrated how to perform these operations both asynchronously and synchronously.

As a reminder, in asynchronous execution mode, you instruct the database engine to perform an operation. The database engine then works in the background while the application keeps running. When the operation finishes the database engine dispatches an event to alert you to that fact. The key benefit of asynchronous execution is that the runtime performs the database operations in the background while the main application code continues executing. This is especially valuable when the operation takes a notable amount of time to run.

On the other hand, in synchronous execution mode operations don't run in the background. You tell the database engine to perform an operation. The code pauses at that point while the database engine does its work. When the operation completes, execution continues with the next line of your code.

A single database connection can't execute some operations or statements synchronously and others asynchronously. You specify whether a SQLConnection operates in synchronous or asynchronous when you open the connection to the database. If you call `SQLConnection.open()` the connection operates in synchronous execution mode, and if you call `SQLConnection.openAsync()` the connection operates in asynchronous execution mode. Once a SQLConnection instance is connected to a database using `open()` or `openAsync()`, it is fixed to synchronous or asynchronous execution.

Contents

Using synchronous database operations

There is little difference in the actual code that you use to execute and respond to operations when using synchronous execution, compared to the code for asynchronous execution mode. The key differences between the two approaches fall into two areas. The first is executing an operation that depends on another operation (such as `SELECT` result rows or the primary key of the row added by an `INSERT` statement). The second area of difference is in handling errors.

Contents

Writing code for synchronous operations

The key difference between synchronous and asynchronous execution is that in synchronous mode you write the code as a single series of steps. In contrast, in asynchronous code you register event listeners and often divide operations among listener methods. When a database is connected in synchronous execution mode, you can execute a series of database operations in succession within a single code block. The following example demonstrates this technique:

```
var conn:SQLConnection = new SQLConnection();
var dbFile:File = File.applicationStorageDirectory.resolvePath("DBSample.db");
```

```
// open the database
conn.open(dbFile, OpenMode.UPDATE);

// start a transaction
conn.begin();

// add the customer record to the database
var insertCustomer:SQLStatement = new SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
    "INSERT INTO customers (firstName, lastName) " +
    "VALUES ('Bob', 'Jones')";

insertCustomer.execute();
var customerId:Number = insertCustomer.getResult().lastInsertRowID;

// add a related phone number record for the customer
var insertPhoneNumber:SQLStatement = new SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
    "INSERT INTO customerPhoneNumbers (customerId, number) " +
    "VALUES (:customerId, '800-555-1234')";
insertPhoneNumber.parameters[":customerId"] = customerId;

insertPhoneNumber.execute();

// commit the transaction
conn.commit();
```

As you can see, you call the same methods to perform database operations whether you're using synchronous or asynchronous execution. The key differences between the two approaches are executing an operation that depends on another operation and handling errors.

Executing an operation that depends on another operation

When you're using synchronous execution mode, you don't need to write code that listens for an event to determine when an operation completes. Instead, you can assume that if an operation in one line of code completes successfully, execution continues with the next line of code. Consequently, to perform an operation that depends on the success of another operation, simply write the dependent code immediately following the operation on which it depends. For instance, to code an application to begin a transaction, execute an `INSERT` statement, retrieve the primary key of the inserted row, insert that primary key into another row of a different table, and finally commit the transaction, the code can all be written as a series of statements. The following example demonstrates these operations:

```
var conn:SQLConnection = new SQLConnection();
var dbFile:File = File.applicationStorageDirectory.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, SQLMode.UPDATE);

// start a transaction
conn.begin();

// add the customer record to the database
var insertCustomer:SQLStatement = new SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
    "INSERT INTO customers (firstName, lastName) " +
    "VALUES ('Bob', 'Jones')";

insertCustomer.execute();
var customerId:Number = insertCustomer.getResult().lastInsertRowID;

// add a related phone number record for the customer
var insertPhoneNumber:SQLStatement = new SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
    "INSERT INTO customerPhoneNumbers (customerId, number) " +
    "VALUES (:customerId, '800-555-1234')";
insertPhoneNumber.parameters[":customerId"] = customerId;

insertPhoneNumber.execute();

// commit the transaction
conn.commit();
```

Handling errors with synchronous execution

In synchronous execution mode, you don't listen for an error event to determine that an operation has failed. Instead, you surround any code that could trigger errors in a set of `try..catch..finally` code blocks. You wrap the error-throwing code in the `try` block. Write the actions to perform in response to each type of error in separate `catch` blocks. Place any code that you want to always execute regardless of success or failure (for example, closing a database connection that's no longer needed) in a `finally` block. The following example demonstrates using `try..catch..finally` blocks for error handling. It builds on the previous example by adding error handling code:

```
var conn:SQLConnection = new SQLConnection();
var dbFile:File = File.applicationStorageDirectory.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, SQLMode.UPDATE);

// start a transaction
conn.begin();

try
{
    // add the customer record to the database
    var insertCustomer:SQLStatement = new SQLStatement();
    insertCustomer.sqlConnection = conn;
    insertCustomer.text =
        "INSERT INTO customers (firstName, lastName)" +
        "VALUES ('Bob', 'Jones')";

    insertCustomer.execute();
    var customerId:Number = insertCustomer.getResult().lastInsertRowID;

    // add a related phone number record for the customer
    var insertPhoneNumber:SQLStatement = new SQLStatement();
    insertPhoneNumber.sqlConnection = conn;
    insertPhoneNumber.text =
        "INSERT INTO customerPhoneNumbers (customerId, number)" +
        "VALUES (:customerId, '800-555-1234')";
    insertPhoneNumber.parameters[":customerId"] = customerId;

    insertPhoneNumber.execute();
    // if we've gotten to this point without errors, commit the transaction
    conn.commit();
}
catch (error:SQLError)
{
    // rollback the transaction
    conn.rollback();
}
```

Understanding the asynchronous execution model

One common concern about using asynchronous execution mode is the assumption that you can't start executing a SQLStatement instance if another SQLStatement is currently executing against the same database connection. In fact, this assumption isn't correct. While a SQLStatement instance is executing you can't change the `text` property of the statement. However, if you use a separate SQLStatement instance for each different SQL statement that you want to execute, you can call the `execute()` method of a SQLStatement while another SQLStatement instance is still executing, without causing an error.

Internally, when you're executing database operations using asynchronous execution mode, each database connection (each SQLConnection instance) has its own queue or list of operations that it is instructed to perform. The runtime executes each operation in sequence, in the order they are added to the queue. When you create a SQLStatement instance and call its `execute()` method, that statement execution operation is added to the queue for the connection. If no operation is currently executing on that SQLConnection instance, the statement begins executing in the background. Suppose that within the same block of code you create another SQLStatement instance and also call that method's `execute()` method. That second statement execution operation is added to the queue behind the first statement. As soon as the first statement finishes executing, the runtime moves to the next operation in the queue. The processing of subsequent operations in the queue happens in the background, even while the `result` event for the first operation is being dispatched in the main application code. The following code demonstrates this technique:

```
// Using asynchronous execution mode

var stmt1:SQLStatement = new SQLStatement();
stmt1.sqlConnection = conn;
// ... Set statement text and parameters, and register event listeners ...
stmt1.execute();
// At this point stmt1's execute() operation is added to conn's execution queue.

var stmt2:SQLStatement = new SQLStatement();
stmt2.sqlConnection = conn;
// ... Set statement text and parameters, and register event listeners ...
stmt2.execute();
// At this point stmt2's execute() operation is added to conn's execution queue.
// When stmt1 finishes executing, stmt2 will immediately begin executing
// in the background.
```

There is an important side effect of the database automatically executing subsequent queued statements. If a statement depends on the outcome of another operation, you can't add the statement to the queue (in other words, you can't call its `execute()` method) until the first operation completes. This is because once you've called the second statement's `execute()` method, you can't change the statement's `text` or `parameters` properties. In that case you must wait for the event indicating that the first operation completes before starting the next operation. For instance, if you want to execute a statement in the context of a transaction, the statement execution depends on the operation of opening the transaction. After calling the `SQLConnection.begin()` method to open the transaction, you need to wait for the SQLConnection instance to dispatch its `begin` event. Only then can you call the SQLStatement instance's `execute()` method. In this example the simplest way to organize the application to ensure that the operations are executed properly is to create a method that's registered as a listener for the `begin` event. The code to call the `SQLStatement.execute()` method is placed within that listener method.

Strategies for working with SQL databases

There are various ways that an application can access and work with a local SQL database. The application design can vary in terms of how the application code is organized, the sequence and timing of how operations are performed, and so on. The techniques you choose can have an impact on how easy it is to develop your application. They can affect how easy it is to modify the application in future updates. They can also affect how well the application performs from the users' perspective.

Contents

Distributing a pre-populated database

When you use an AIR local SQL database in your application, the application expects a database with a certain structure of tables, columns, and so forth. Some applications also expect certain data to be pre-populated in the database file. One way to ensure that the database has the proper structure is to create the database within the application code. When the application loads it checks for the existence of its database file in a particular location. If the file doesn't exist, the application executes a set of commands to create the database file, create the database structure, and populate the tables with the initial data.

The code that creates the database and its tables is frequently complex. It is often only used once in the installed lifetime of the application, but still adds to the size and complexity of the application. As an alternative to creating the database, structure, and data programmatically, you can distribute a pre-populated database with your application. To distribute a predefined database, include the database file in the application's AIR package.

Like all files that are included in an AIR package, a bundled database file is installed in the application directory (the directory represented by the `File.applicationDirectory` property). However, files in that directory are read only. Use the file from the AIR package as a "template" database. The first time a user runs the application, copy the original database file into the user's application storage directory (or another location), and use that database within the application.

Improving database performance

Several techniques that are built into Adobe AIR allow you to improve the performance of database operations in your application.

Contents

- Minimize runtime processing
- Avoid schema changes

In addition to the techniques described here, the way a SQL statement is written can also affect database performance. Frequently, there are multiple ways to write a SQL `SELECT` statement to retrieve a particular result set. In some cases, the different approaches require more or less effort from the database engine. This aspect of improving database performance—designing SQL statements for better performance—is not covered in the Adobe AIR documentation.

Use one SQLStatement instance for each SQL statement

Before any SQL statement is executed, the runtime prepares (compiles) it to determine the steps that are performed internally to carry out the statement. When you call `SQLStatement.execute()` on a SQLStatement instance that hasn't executed previously, the statement is automatically prepared before it is executed. On subsequent calls to the `execute()` method, as long as the `SQLStatement.text` property hasn't changed the statement is still prepared. Consequently, it executes faster.

In order to gain the maximum benefit from reusing statements, if values need to change between statement executions, use statement parameters to customize your statement. (Statement parameters are specified using the `SQLStatement.parameters` associative array property.) Unlike changing the SQLStatement instance's `text` property, if you change the values of statement parameters the runtime isn't required to prepare the statement again. For more information about using parameters in statements, see "Using parameters in statements" on page 216.

Because preparing and executing a statement is an operation that is potentially demanding, a good strategy is to preload initial data and then execute other statements in the background. Load the data that the application needs first. When the initial start-up operations of your application have completed, or at another "idle" time in the application, execute other statements. For instance, if your application doesn't access the database at all in order to display its initial screen, wait until that screen displays, then open the database connection, and finally create the SQLStatement instances and execute any that you can. Alternatively, suppose when your application starts up it immediately displays some data, such as the result of a particular query. In that case, go ahead and execute the SQLStatement instance for that query. After the initial data is loaded and displayed, create SQLStatement instances for other database operations and if possible execute other statements that are needed later.

When you're reusing a SQLStatement instance, your application needs to keep a reference to the SQLStatement instance once it has been prepared. To keep a reference to the instance, declare the variable as a class-scope variable rather than a function-scope variable. One good way to do this is to structure your application so that a SQL statement is wrapped in a single class. A group of statements that are executed in combination can also be wrapped in a single class. By defining the SQLStatement instance or instances as member variables of the class, they persist as long as the instance of the wrapper class exists in the application. At a minimum, you can simply define a variable containing the SQLStatement instance outside of a function so that the instance persists in memory. For example, declare the SQLStatement instance as a member variable in an ActionScript class or as a non-function variable in a JavaScript file. You can then set the statement's parameter values and call its `execute()` method when you want to actually run the query.

Group multiple operations in a transaction

Suppose you're executing a large number of SQL statements that involve adding or changing data (`INSERT` or `UPDATE` statements). You can get a significant increase in performance by executing all the statements within an explicit transaction. If you don't explicitly begin a transaction, each of the statements runs in its own automatically created transaction. After each transaction (each statement) finishes executing, the runtime writes the resulting data to the database file on the disk. On the other hand, consider what happens if you explicitly create a transaction and execute the statements in the context of that transaction. The runtime makes all the changes in memory, then writes all the changes to the database file at one time when the transaction is committed. Writing the data to disk is usually the most time-intensive part of the operation. Consequently, writing to the disk one time rather than once per SQL statement can improve performance significantly.

Minimize runtime processing

Using the following techniques can prevent unneeded work on the part of the database engine and make applications perform better:

- Always explicitly specify database names along with table names in a statement. (Use "main" if it's the main database). For example, use `SELECT employeeId FROM` **`main.`**`employees` rather than `SELECT employeId FROM employees`. Explicitly specifying the database name prevents the runtime from having to check each database to find the matching table. It also prevents the possibility of having the runtime choose the wrong database. Follow this rule even if a SQLConnection is only connected to a single database, because behind the scenes the SQLConnection is also connected to a temporary database that is accessible through SQL statements.
- Always explicitly specify column names in a `SELECT` or `INSERT` statement.
- Break up the rows returned by a `SELECT` statement that retrieves a large number of rows: see "Retrieving SELECT results in parts" on page 223.

Avoid schema changes

If possible, avoid changing the schema (table structure) of a database once you've added data into the database's tables. Normally a database file is structured with the table definitions at the start of the file. When you open a connection to a database, the runtime loads those definitions. When you add data to database tables, that data is added to the file after the table definition data. However, if you make schema changes such as adding a column to a table or adding a new table, the new table definition data is mixed in with the table data in the database file. If the table definition data is not all at the start of the database file, it takes longer to open a connection to the database as the runtime reads the table definition data from different parts of the file.

If you do need to make schema changes, you can call the `SQLConnection.compact()` method after completing the changes. This operation restructures the database file so that the table definition data is located together at the start of the file. However, the `compact()` operation can be time-intensive, especially as a database file grows larger.

Best practices for working with local SQL databases

The following list is a set of suggested techniques you can use to improve the performance, security, and ease of maintenance of your applications when working with local SQL databases. For additional techniques for improving database applications, see "Improving database performance" on page 235.

Contents

- Use statement parameters
- Use constants for column and parameter names

Pre-create database connections

Even if your application doesn't execute any statements when it first loads, instantiate a SQLConnection object and call its `open()` or `openAsync()` method ahead of time (such as after the initial application startup) to avoid delays when running statements. See "Connecting to a database" on page 214.

Reuse database connections

If you access a certain database throughout the execution time of your application, keep a reference to the SQLConnection instance, and reuse it throughout the application, rather than closing and reopening the connection. See "Connecting to a database" on page 214.

Favor asynchronous execution mode

When writing data-access code, it can be tempting to execute operations synchronously rather than asynchronously, because using synchronous operations frequently requires shorter and less complex code. However, as described in "Using synchronous and asynchronous database operations" on page 231, synchronous operations can have a performance impact that is obvious to users and detrimental to their experience with an application. The amount of time a single operation takes varies according to the operation and particularly the amount of data it involves. For instance, a SQL `INSERT` statement that only adds a single row to the database takes less time than a `SELECT` statement that retrieves thousands of rows of data. However, when you're using synchronous execution to perform multiple operations, the operations are usually strung together. Even if the time each single operation takes is very short, the application is frozen until all the synchronous operations finish. As a result, the cumulative time of multiple operations strung together may be enough to stall your application.

Use asynchronous operations as a standard approach, especially with operations that involve large numbers of rows. There is a technique for dividing up the processing of large sets of `SELECT` statement results, described in "Retrieving SELECT results in parts" on page 223. However, this technique can only be used in asynchronous execution mode. Only use synchronous operations when you can't achieve certain functionality using asynchronous programming, when you've considered the performance trade-offs that your application's users will face, and when you've tested your application so that you know how your application's performance is affected. Using asynchronous execution can involve more complex coding. However, remember that you only have to write the code once, but the application's users have to use it repeatedly, fast or slow.

In many cases, by using a separate SQLStatement instance for each SQL statement to be executed, multiple SQL operations can be queued up at one time, which makes asynchronous code like synchronous code in terms of how the code is written. For more information, see "Understanding the asynchronous execution model" on page 234.

Use separate SQL statements and don't change the SQLStatement's text property

For any SQL statement that is executed more than once in an application, create a separate SQLStatement instance for each SQL statement. Use that SQLStatement instance each time that SQL command executes. For example, suppose you are building an application that includes four different SQL operations that are performed multiple times. In that case, create four separate SQLStatement instances and call each statement's `execute()` method to run it. Avoid the alternative of using a single SQLStatement instance for all SQL statements, redefining its `text` property each time before executing the statement. See "Use one SQLStatement instance for each SQL statement" on page 236 for more information.

Use statement parameters

Use SQLStatement parameters—never concatenate user input into statement text. Using parameters makes your application more secure because it prevents the possibility of SQL injection attacks. It makes it possible to use objects in queries (rather than only SQL literal values). It also makes statements run more efficiently because they can be reused without needing to be recompiled each time they're executed. See "Using parameters in statements" on page 216 for more information.

Use constants for column and parameter names

When you don't specify an `itemClass` for a SQLStatement, to avoid spelling errors, define String constants containing a table's column names. Use those constants in the statement text and for the property names when retrieving values from result objects. Also use constants for parameter names.

Chapter 22: Storing encrypted data

The Adobe® AIR™ runtime provides a persistent encrypted local store for each AIR application installed on a user's computer. This lets you save and retrieve data that is stored on the user's local hard drive in an encrypted format that cannot easily be deciphered by other applications or users. A separate encrypted local store is used for each AIR application, and each AIR application uses a separate encrypted local store for each user.

You may want to use the encrypted local store to store information that must be secured, such as login credentials for web services.

AIR uses DPAPI on Windows and KeyChain on Mac OS to associate the encrypted local store to each application and user. The encrypted local store uses AES-CBC 128-bit encryption.

Information in the encrypted local store is only available to AIR application content in the application security sandbox.

Use the `setItem()` and `removeItem()` static methods of the EncryptedLocalStore class to store and retrieve data from the local store. The data is stored in a hash table, using strings as keys, with the data stored as byte arrays.

For example, the following code stores a string in the encrypted local store:

```
var str:String = "Bob";
var bytes:ByteArray = new ByteArray();
bytes.writeUTFBytes(str);
EncryptedLocalStore.setItem("firstName", bytes);

var storedValue:ByteArray = EncryptedLocalStore.getItem("firstName");
trace(storedValue.readUTFBytes(storedValue.length)); // "Bob"
```

The third parameter of the `setItem()` method, the `stronglyBound` parameter, is optional. When this parameter is set to `true`, the encrypted local store provides a higher level of security, by binding the stored item to the storing AIR application's digital signature and bits, as well as to the application's publisher ID when:

```
var str:String = "Bob";
var bytes:ByteArray = new ByteArray();
bytes.writeUTFBytes(str);
EncryptedLocalStore.setItem("firstName", bytes, true);
```

For an item that is stored with `stronglyBound` set to `true`, subsequent calls to `getItem()` only succeed if the calling AIR application is identical to the storing application (if no data in files in the application directory have changed). If the calling AIR application is different from the storing application, the application throws an Error exception when you call `getItem()` for a strongly bound item. If you update your application, it will not be able to read strongly bound data previously written to the encrypted local store.

By default, an AIR application cannot read the encrypted local store of another application. The `stronglyBound` setting provides extra binding (to the data in the application bits) that prevents an attacker application from attempting to read from your application's encrypted local store by trying to hijack your application's publisher ID.

You can delete a value from the encrypted local store by using the `EncryptedLocalStore.removeItem()` method, as in the following example:

```
EncryptedLocalStore.removeItem("firstName");
```

You can clear all data from the encrypted local store by calling the `EncryptedLocalStore.reset()` method, as in the following example:

```
EncryptedLocalStore.reset();
```

When debugging an application in the AIR Debug Launcher (ADL), the application uses a different encrypted local store than the one used in the installed version of the application.

The encrypted local store has a maximum supported total capacity of 10 MB.

When you uninstall an AIR application, the uninstaller does not delete data stored in the encrypted local store.

Encrypted local store data is put in a subdirectory of the user's application data directory; the subdirectory path is Adobe/AIR/ELS/ followed by the application ID.

Part 7: HTML content

Chapter 23: About the HTML environment

Adobe® AIR™ uses WebKit (*www.webkit.org*), also used by the Safari web browser, to parse, layout, and render HTML and JavaScript content. Using the AIR APIs in HTML content is optional. You can program in the content of an HTMLLoader object or HTML window entirely with HTML and JavaScript. Most existing HTML pages and applications should run with few changes (assuming they use HTML, CSS, DOM, and JavaScript features compatible with WebKit).

Because AIR applications run directly on the desktop, with full access to the file system, the security model for HTML content is more stringent than the security model of a typical web browser. In AIR, only content loaded from the application installation directory is placed in the *application sandbox*. The application sandbox has the highest level of privilege and allows access to the AIR APIs. AIR places other content into isolated sandboxes based on where that content came from. Files loaded from the file system go into a local sandbox. Files loaded from the network using the http: or https: protocols go into a sandbox based on the domain of the remote server. Content in these non-application sandboxes is prohibited from accessing any AIR API and runs much as it would in a typical web browser.

Contents

See also

- "Programming in HTML and JavaScript" on page 258
- "Handling HTML-related events" on page 274
- "Scripting the HTML Container" on page 279

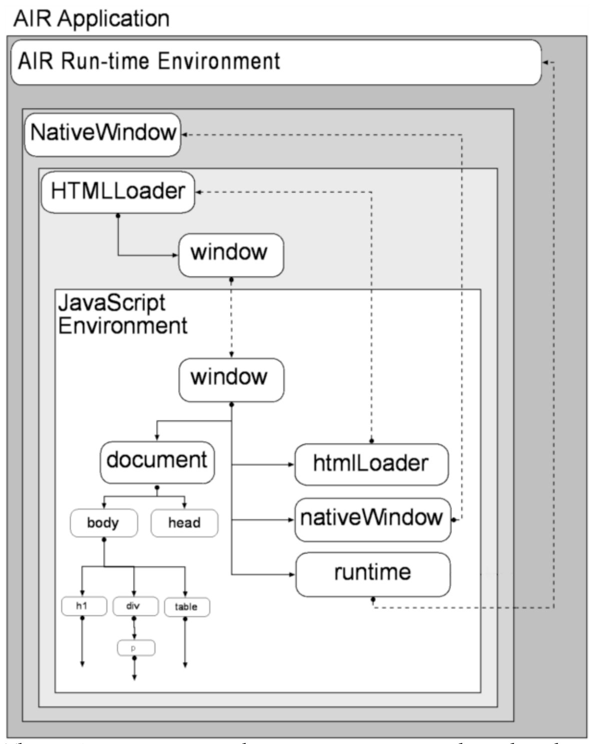
Overview of the HTML environment

Adobe AIR provides a complete browser-like JavaScript environment with an HTML renderer, document object model, and JavaScript interpreter. The JavaScript environment is represented by the AIR HTMLLoader class. In HTML windows, an HTMLLoader object contains all HTML content, and is, in turn, contained within a NativeWindow object. In SWF content, the HTMLLoader class, which extends the Sprite class, can be added to the display list of a stage like any other display object. The ActionScript™ properties of the class are described in "Scripting the HTML Container" on page 279 and also in the *Flex 3 ActionScript Language Reference*. In the Flex framework, the AIR HTMLLoader class is wrapped in a mx:HTML component. The mx:HTML component extends the UIComponent class, so it can be used directly with other Flex containers. The JavaScript environment within the mx:HTML component is otherwise identical.

Contents

About the JavaScript environment and its relationship to AIR

The following diagram illustrates the relationship between the JavaScript environment and the AIR run-time environment. Although only a single native window is shown, an AIR application can contain multiple windows. (And a single window can contain multiple HTMLLoader objects.)



The JavaScript environment has its own Document and Window objects. JavaScript code can interact with the AIR run-time environment through the `runtime`, `nativeWindow`, and `htmlLoader` properties. ActionScript code can interact with the JavaScript environment through the `window` property of an HTMLLoader object, which is a reference to the JavaScript Window object. In addition, both ActionScript and JavaScript objects can listen for events dispatched by both AIR and JavaScript objects.

The `runtime` property provides access to AIR API classes, allowing you to create new AIR objects as well as access class (also called static) members. To access an AIR API, you add the name of the class, with package, to the `runtime` property. For example, to create a File object, you would use the statement:

```
var file = new window.runtime.filesystem.File();
```

***Note:** The AIR SDK provides a JavaScript file, `AIRAliases.js`, that defines more convenient aliases for the most commonly used AIR classes. When you import this file, you can use the shorter form `air.Class` instead of `window.runtime.package.Class`. For example, you could create the File object with `new air.File()`.*

The NativeWindow object provides properties for controlling the desktop window. From within an HTML page, you can access the containing NativeWindow object with the `window.nativeWindow` property.

The HTMLLoader object provides properties, methods, and events for controlling how content is loaded and rendered. From within an HTML page, you can access the parent HTMLLoader object with the `window.htmlLoader` property.

Important: *Only pages installed as part of an application have the* `htmlLoader`*,* `nativeWindow`*, or* `runtime` *properties and only when loaded as the top-level document. These properties are not added when a document is loaded into a frame or iframe. (A child document can access these properties on the parent document as long as it is in the same security sandbox. For example, a document loaded in a frame could access the* `runtime` *property of its parent with* `parent.runtime`*.)*

About security

AIR executes all code within a security sandbox based on the domain of origin. Application content, which is limited to content loaded from the application installation directory, is placed into the *application* sandbox. Access to the run-time environment and the AIR APIs are only available to HTML and JavaScript running within this sandbox. At the same time, most dynamic evaluation and execution of JavaScript is blocked in the application sandbox after all handlers for the page `load` event have returned.

You can map an application page into a non-application sandbox by loading the page into a frame or iframe and setting the AIR-specific `sandboxRoot` and `documentRoot` attributes of the frame. By setting the `sandboxRoot` value to an actual remote domain, you can enable the sandboxed content to cross-script content in that domain. Mapping pages in this way can be useful when loading and scripting remote content, such as in a *mash-up* application.

Another way to allow application and non-application content to cross-script each other, and the only way to give non-application content access to AIR APIs, is to create a *sandbox bridge*. A *parent-to-child* bridge allows content in a child frame, iframe, or window to access designated methods and properties defined in the application sandbox. Conversely, a *child-to-parent* bridge allows application content to access designated methods and properties defined in the sandbox of the child. Sandbox bridges are established by setting the `parentSandboxBridge` and `childSandboxBridge` properties of the window object. For more information, see "HTML security" on page 75 and "HTML frame and iframe elements" on page 253.

About plug-ins and embedded objects

AIR supports the Adobe® Acrobat® plug-in. Users must have Acrobat or Adobe® Reader® 8.1 (or better) to display PDF content. The HTMLLoader object provides a property for checking whether a user's system can display PDF. SWF file content can also be displayed within the HTML environment, but this capability is built in to AIR and does not use an external plug-in.

No other Webkit plug-ins are supported in AIR.

See also

- "HTML security" on page 75
- "HTML Sandboxes" on page 247
- "HTML frame and iframe elements" on page 253
- "JavaScript Window object" on page 251
- "The XMLHttpRequest object" on page 247
- "Adding PDF content" on page 293

AIR and Webkit extensions

Adobe AIR uses the open source Webkit engine, also used in the Safari web browser. AIR adds several extensions to allow access to the runtime classes and objects as well as for security. In addition, Webkit itself adds features not included in the W3C standards for HTML, CSS, and JavaScript.

Only the AIR additions and the most noteworthy Webkit extensions are covered here; for additional documentation on non-standard HTML, CSS, and JavaScript, see www.webkit.org and developer.apple.com. For standards information, see the W3C website. Mozilla also provides a valuable general reference on HTML, CSS, and DOM topics (of course, the Webkit and Mozilla engines are not identical).

Note: *AIR does not support the following standard and extended WebKit features: the JavaScript Window object* `print()` *method; plug-ins, except Acrobat or Adobe Reader 8.1+; Scalable Vector Graphics (SVG), the CSS* `opacity` *property.*

Contents

JavaScript in AIR

AIR makes several changes to the typical behavior of common JavaScript objects. Many of these changes are made to make it easier to write secure applications in AIR. At the same time, these differences in behavior mean that some common JavaScript coding patterns, and existing web applications using those patterns, might not always execute as expected in AIR. For information on correcting these types of issues, see "Avoiding security-related JavaScript errors" on page 260.

Contents

HTML Sandboxes

AIR places content into isolated sandboxes according to the origin of the content. The sandbox rules are consistent with the same-origin policy implemented by most web browsers, as well as the rules for sandboxes implemented by the Adobe Flash Player. In addition, AIR provides a new *application* sandbox type to contain and protect application content. See "Sandboxes" on page 73 for more information on the types of sandboxes you may encounter when developing AIR applications.

Access to the run-time environment and AIR APIs are only available to HTML and JavaScript running within the application sandbox. At the same time, however, dynamic evaluation and execution of JavaScript, in its various forms, is largely restricted within the application sandbox for security reasons. These restrictions are in place whether or not your application actually loads information directly from a server. (Even file content, pasted strings, and direct user input may be untrustworthy.)

The origin of the content in a page determines the sandbox to which it is consigned. Only content loaded from the application directory (the installation directory referenced by the `app:` URL scheme) is placed in the application sandbox. Content loaded from the file system is placed in the *local-with-filesystem* or the *local-trusted* sandbox, which allows access and interaction with content on the local file system, but not remote content. Content loaded from the network is placed in a remote sandbox corresponding to its domain of origin.

To allow an application page to interact freely with content in a remote sandbox, the page can be mapped to the same domain as the remote content. For example, if you write an application that displays map data from an Internet service, the page of your application that loads and displays content from the service could be mapped to the service domain. The attributes for mapping pages into a remote sandbox and domain are new attributes added to the frame and iframe HTML elements.

To allow content in a non-application sandbox to safely use AIR features, you can set up a parent sandbox bridge. To allow application content to safely call methods and access properties of content in other sandboxes, you can set up a child sandbox bridge. Safety here means that remote content cannot accidentally get references to objects, properties, or methods that are not explicitly exposed. Only simple data types, functions, and anonymous objects can be passed across the bridge. However, you must still avoid explicitly exposing potentially dangerous functions. If, for example, you exposed an interface that allowed remote content to read and write files anywhere on a user's system, then you might be giving remote content the means to do considerable harm to your users.

JavaScript eval() function

Use of the `eval()` function is restricted within the application sandbox once a page has finished loading. Some uses are permitted so that JSON-formatted data can be safely parsed, but any evaluation that results in executable statements results in an error. "Code restrictions for content in different sandboxes" on page 77 describes the allowed uses of the `eval()` function.

Function constructors

In the application sandbox, function constructors can be used before a page has finished loading. After all page `load` event handlers have finished, new functions cannot be created.

Loading external scripts

HTML pages in the application sandbox cannot use the `script` tag to load JavaScript files from outside of the application directory. For a page in your application to load a script from outside the application directory, the page must be mapped to a non-application sandbox.

The XMLHttpRequest object

AIR provides an XMLHttpRequest (XHR) object that applications can use to make data requests. The following example illustrates a simple data request:

```
xmlhttp = new XMLHttpRequest();
xmlhttp.open("GET", "http:/www.example.com/file.data", true);
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4) {
        //do something with data...
    }
}
xmlhttp.send(null);
```

In contrast to a browser, AIR allows content running in the application sandbox to request data from any domain. The result of an XHR that contains a JSON string can be evaluated into data objects unless the result also contains executable code. If executable statements are present in the XHR result, an error is thrown and the evaluation attempt fails.

To prevent accidental injection of code from remote sources, synchronous XHRs return an empty result if made before a page has finished loading. Asynchronous XHRs will always return after a page has loaded.

By default, AIR blocks cross-domain XMLHttpRequests in non-application sandboxes. A parent window in the application sandbox can choose to allow cross-domain requests in a child frame containing content in a non-application sandbox by setting `allowCrossDomainXHR`, an attribute added by AIR, to `true` in the containing frame or iframe element:

```
<iframe id="mashup"
    src="http://www.example.com/map.html"
    allowCrossDomainXHR="true"
</iframe>
```

Note: *When convenient, the AIR URLStream class can also be used to download data.*

If you dispatch an XMLHttpRequest to a remote server from a frame or iframe containing application content that has been mapped to a remote sandbox, make sure that the mapping URL does not mask the server address used in the XHR. For example, consider the following iframe definition, which maps application content into a remote sandbox for the example.com domain:

```
<iframe id="mashup"
    src="http://www.example.com/map.html"
    documentRoot="app:/sandbox/"
    sandboxRoot="http://www.example.com/"
    allowCrossDomainXHR="true"
</iframe>
```

Because the `sandboxRoot` attribute remaps the root URL of the www.example.com address, all requests are loaded from the application directory and not the remote server. Requests are remapped whether they derive from page navigation or from an XMLHttpRequest.

To avoid accidentally blocking data requests to your remote server, map the `sandboxRoot` to a subdirectory of the remote URL rather than the root. The directory does not have to exist. For example, to allow requests to the www.example.com to load from the remote server rather than the application directory, change the previous iframe to the following:

```
<iframe id="mashup"
    src="http://www.example.com/map.html"
    documentRoot="app:/sandbox/"
    sandboxRoot="http://www.example.com/air/"
    allowCrossDomainXHR="true"
</iframe>
```

In this case, only content in the `air` subdirectory is loaded locally.

For more information on sandbox mapping see "HTML frame and iframe elements" on page 253 and "HTML security" on page 75.

The Canvas object

The Canvas object defines an API for drawing geometric shapes such as lines, arcs, ellipses, and polygons. To use the canvas API, you first add a canvas element to the document and then draw into it using the JavaScript Canvas API. In most other respects, the Canvas object behaves like an image.

The following example draws a triangle using a Canvas object:

```
<html>
<body>
<canvas id="triangleCanvas" style="width:40px; height:40px;"></canvas>
<script>
    var canvas = document.getElementById("triangleCanvas");
    var context = canvas.getContext("2d");
    context.lineWidth = 3;
    context.strokeStyle = "#457232";
    context.beginPath();
        context.moveTo(5,5);
        context.lineTo(35,5);
        context.lineTo(20,35);
        context.lineTo(5,5);
        context.lineTo(6,5);
    context.stroke();
</script>
</body>
</html>
```

For more documentation on the Canvas API, see the Safari JavaScript Reference from Apple. Note that the Webkit project recently began changing the Canvas API to standardize on the HTML 5 Working Draft proposed by the Web Hypertext Application Technology Working Group (WHATWG) and W3C. As a result, some of the documentation in the Safari JavaScript Reference may be inconsistent with the version of the canvas present in AIR.

Cookies

In AIR applications, only content in remote sandboxes (content loaded from http: and https: sources) can use cookies (the `document.cookie` property). In the application sandbox, AIR APIs provide other means for storing persistent data (such as the EncryptedLocalStore and FileStream classes).

The Clipboard object

The WebKit Clipboard API is driven with the following events: `copy`, `cut`, and `paste`. The event object passed in these events provides access to the clipboard through the `clipboardData` property. Use the following methods of the `clipboardData` object to read or write clipboard data:

Method	**Description**
clearData(mimeType)	Clears the clipboard data. Set the `mimeType` parameter to the MIME type of the data to clear.
getData(mimeType)	Get the clipboard data. This method can only be called in a handler for the `paste` event. Set the `mimeType` parameter to the MIME type of the data to return.
setData(mimeType, data)	Copy data to the clipboard. Set the `mimeType` parameter to the MIME type of the data.

JavaScript code outside the application sandbox can only access the clipboard through theses events. However, content in the application sandbox can access the system clipboard directly using the AIR Clipboard class. For example, you could use the following statement to get text format data on the clipboard:

```
var clipping = air.Clipboard.generalClipboard.getData("text/plain",
                                air.ClipboardTransferMode.ORIGINAL_ONLY);
```

The valid data MIME types are:

MIME type	Value
Text	"text/plain"
HTML	"text/html"
URL	"text/uri-list"
Bitmap	"image/x-vnd.adobe.air.bitmap"
File list	"application/x-vnd.adobe.air.file-list"

Important: *Only content in the application sandbox can access file data present on the clipboard. If non-application content attempts to access a file object from the clipboard, a security error is thrown.*

Drag and Drop

Drag-and-drop gestures into and out of HTML produce the following DOM events: `dragstart`, `drag`, `dragend`, `dragenter`, `dragover`, `dragleave`, and `drop`. The event object passed in these events provides access to the dragged data through the `dataTransfer` property. The `dataTransfer` property references an object that provides the same methods as the `clipboardData` object associated with a clipboard event. For example, you could use the following function to get text format data from a `drop` event:

```
function onDrop(dragEvent){
    return dragEvent.dataTransfer.getData("text/plain",
            air.ClipboardTransferMode.ORIGINAL_ONLY);
}
```

The `dataTransfer` object has the following important members:

Member	Description
clearData(mimeType)	Clears the data. Set the `mimeType` parameter to the MIME type of the data representation to clear.
getData(mimeType)	Get the dragged data. This method can only be called in a handler for the `drop` event. Set the `mimeType` parameter to the MIME type of the data to get.
setData(mimeType, data)	Set the data to be dragged. Set the `mimeType` parameter to the MIME type of the data.
types	An array of strings containing the MIME types of all data representations currently available in the `dataTransfer` object.
effectsAllowed	Specifies whether the data being dragged can be copied, moved, linked, or some combination thereof. Set the `effectsAllowed` property in the handler for the `dragstart` event.
dropEffect	Specifies which of the allowed drop effects are supported by a drag target. Set the `dropEffect` property in the handler for the `dragEnter` event. During the drag, the cursor changes to indicate which effect would occur if the user released the mouse. If no `dropEffect` is specified, an `effectsAllowed` property effect is chosen. The copy effect has priority over the move effect, which itself has priority over the link effect. The user can modify the default priority using the keyboard.

innerHTML and outerHTML properties

AIR places security restrictions on the use of the `innerHTML` and `outerHTML` properties for content running in the application sandbox. Before the page load event, as well as during the execution of any load event handlers, use of the `innerHTML` and `outerHTML` properties is unrestricted. However, once the page has loaded, you can only use `innerHTML` or `outerHTML` properties to add static content to the document. Any statement in the string assigned to `innerHTML` or `outerHTML` that evaluates to executable code is ignored. For example, if you include an event callback attribute in an element definition, the event listener is not added. Likewise, embedded `<script>` tags are not evaluated. For more information, see the "HTML security" on page 75.

Document.write() and Document.writeln() methods

Use of the `write()` and `writeln()` methods is not restricted in the application sandbox before the `load` event of the page. However, once the page has loaded, calling either of these methods does not clear the page or create a new one. In a non-application sandbox, as in most web browsers, calling `document.write()` or `writeln()` after a page has finished loading clears the current page and opens a new, blank one.

Document.designMode property

Set the `document.designMode` property to a value of `on` to make all elements in the document editable. Built-in editor support includes text editing, copy, paste, and drag-and-drop. Setting `designMode` to `on` is equivalent to setting the `contentEditable` property of the `body` element to `true`. You can use the `contentEditable` property on most HTML elements to define which sections of a document are editable. See "HTML contentEditable attribute" on page 256 for additional information.

unload events (for body and frameset objects)

In the top-level `frameset` or `body` tag of a window (including the main window of the application), do not use the `unload` event to respond to the window (or application) being closed. Instead, use `exiting` event of the NativeApplication object (to detect when an application is closing). Or use the `closing` event of the NativeWindow object (to detect when a window is closing). For example, the following JavaScript code displays a message (`"Goodbye."`) when the user closes the application:

```
var app = air.NativeApplication.nativeApplication;
app.addEventListener(air.Event.EXITING, closeHandler);
function closeHandler(event)
{
    alert("Goodbye.");
}
```

However, scripts *can* successfully respond to the `unload` event caused by navigation of a frame, iframe, or top-level window content.

***Note:** These limitations may be removed in a future version of Adobe AIR.*

JavaScript Window object

The Window object remains the global object in the JavaScript execution context. In the application sandbox, AIR adds new properties to the JavaScript Window object to provide access to the built-in classes of AIR, as well as important host objects. In addition, some methods and properties behave differently depending on whether they are within the application sandbox or not.

Window.runtime property The `runtime` property allows you to instantiate and use the built-in runtime classes from within the application sandbox. These classes include the AIR and Flash Player APIs (but not, for example, the Flex framework). For example, the following statement creates an AIR file object:

```
var preferencesFile = new window.runtime.flash.filesystem.File();
```

The `AIRAliases.js` file, provided in the AIR SDK, contains alias definitions that allow you to shorten such references. For example, when `AIRAliases.js` is imported into a page, a File object can be created with the following statement:

```
var preferencesFile = new air.File();
```

The `window.runtime` property is only defined for content within the application sandbox and only for the parent document of a page with frames or iframes.

See "Using the AIRAliases.js file" on page 264.

Window.nativeWindow property The `nativeWindow` property provides a reference to the underlying native window object. With this property, you can script window functions and properties such as screen position, size, and visibility, and handle window events such as closing, resizing, and moving. For example, the following statement closes the window:

```
window.nativeWindow.close();
```

***Note:** The window control features provided by the NativeWindow object overlap the features provided by the JavaScript Window object. In such cases, you can use whichever method you find most convenient.*

The `window.nativeWindow` property is only defined for content within the application sandbox and only for the parent document of a page with frames or iframes.

Window.htmlLoader property The `htmlLoader` property provides a reference to the AIR HTMLLoader object that contains the HTML content. With this property, you can script the appearance and behavior of the HTML environment. For example, you can use the `htmlLoader.paintsDefaultBackground` property to determine whether the control paints a default, white background:

```
window.htmlLoader.paintsDefaultBackground = false;
```

***Note:** The HTMLLoader object itself has a `window` property, which references the JavaScript Window object of the HTML content it contains. You can use this property to access the JavaScript environment through a reference to the containing HTMLLoader.*

The `window.htmlLoader` property is only defined for content within the application sandbox and only for the parent document of a page with frames or iframes.

Window.parentSandboxBridge and Window.childSandboxBridge properties The `parentSandboxBridge` and `childSandboxBridge` properties allow you to define an interface between a parent and a child frame. For more information, see "Cross-scripting content in different security sandboxes" on page 270.

Window.setTimeout() and Window.setInterval() functions AIR places security restrictions on use of the `setTimeout()` and `setInterval()` functions within the application sandbox. You cannot define the code to be executed as a string when calling `setTimeout()` or `setInterval()`. You must use a function reference. For more information, see "setTimeout() and setInterval()" on page 262.

Window.open() function When called by code running in a non-application sandbox, the `open()` method only opens a window when called as a result of user interaction (such as a mouse click or keypress). In addition, the window title is prefixed with the application title (to prevent windows opened by remote content from impersonating windows opened by the application). For more information, see the "Restrictions on calling the JavaScript window.open() method" on page 80.

air.NativeApplication object

The NativeApplication object provides information about the application state, dispatches several important application-level events, and provides useful functions for controlling application behavior. A single instance of the NativeApplication object is created automatically and can be accessed through the class-defined `NativeApplication.nativeApplication` property.

To access the object from JavaScript code you could use:

```
var app = window.runtime.flash.desktop.NativeApplication.nativeApplication;
```

Or, if the `AIRAliases.js` script has been imported, you could use the shorter form:

```
var app = air.NativeApplication.nativeApplication;
```

The NativeApplication object can only be accessed from within the application sandbox. “Interacting with the operating system” on page 307 describes the NativeApplication object in detail.

The JavaScript URL scheme

Execution of code defined in a JavaScript URL scheme (as in `href="javascript:alert('Test')"`) is blocked within the application sandbox. No error is thrown.

Extensions to HTML

AIR and WebKit define a few non-standard HTML elements and attributes, including:

- “HTML frame and iframe elements” on page 253
- “HTML Canvas element” on page 255
- “HTML element event handlers” on page 255

HTML frame and iframe elements

AIR adds new attributes to the frame and iframe elements of content in the application sandbox:

sandboxRoot attribute The `sandboxRoot` attribute specifies an alternate, non-application domain of origin for the file specified by the frame `src` attribute. The file is loaded into the non-application sandbox corresponding to the specified domain. Content in the file and content loaded from the specified domain can cross-script each other.

Important: *If you set the value of `sandboxRoot` to the base URL of the domain, all requests for content from that domain are loaded from the application directory instead of the remote server (whether that request results from page navigation, from an XMLHttpRequest, or from any other means of loading content).*

documentRoot attribute The `documentRoot` attribute specifies the local directory from which to load URLs that resolve to files within the location specified by `sandboxRoot`.

When resolving URLs, either in the frame `src` attribute, or in content loaded into the frame, the part of the URL matching the value specified in `sandboxRoot` is replaced with the value specified in `documentRoot`. Thus, in the following frame tag:

```
<iframe     src="http://www.example.com/air/child.html"
            documentRoot="app:/sandbox/"
            sandboxRoot="http://www.example.com/air/"/>
```

`child.html` is loaded from the `sandbox` subdirectory of the application installation folder. Relative URLs in `child.html` are resolved based on `sandbox` directory. Note that any files on the remote server at `www.example.com/air` are not accessible in the frame, since AIR would attempt to load them from the `app:/sandbox/` directory.

allowCrossDomainXHR attribute Include `allowCrossDomainXHR="allowCrossDomainXHR"` in the opening frame tag to allow content in the frame to make XMLHttpRequests to any remote domain. By default, non-application content can only make such requests to its own domain of origin. There are serious security implications involved in allowing cross-domain XHRs. Code in the page is able to exchange data with any domain. If malicious content is somehow injected into the page, any data accessible to code in the current sandbox can be compromised. Only enable cross-domain XHRs for pages that you create and control and only when cross-domain data loading is truly necessary. Also, carefully validate all external data loaded by the page to prevent code injection or other forms of attack.

***Important:** If the `allowCrossDomainXHR` attribute is included in a frame or iframe element, cross-domain XHRs are enabled (unless the value assigned is "0" or starts with the letters "f" or "n"). For example, setting `allowCrossDomainXHR` to `"deny"` would still enable cross-domain XHRs. Leave the attribute out of the element declaration altogether if you do not want to enable cross-domain requests.*

ondominitialize attribute Specifies an event handler for the `dominitialize` event of a frame. This event is an AIR-specific event that fires when the window and document objects of the frame have been created, but before any scripts have been parsed or document elements created.

The frame dispatches the `dominitialize` event early enough in the loading sequence that any script in the child page can reference objects, variables, and functions added to the child document by the `dominitialize` handler. The parent page must be in the same sandbox as the child to directly add or access any objects in a child document. However, a parent in the application sandbox can establish a sandbox bridge to communicate with content in a non-application sandbox.

The following examples illustrate use of the iframe tag in AIR:

Place `child.html` in a remote sandbox, without mapping to an actual domain on a remote server:

```
<iframe     src="http://localhost/air/child.html"
            documentRoot="app:/sandbox/"
            sandboxRoot="http://localhost/air/"/>
```

Place `child.html` in a remote sandbox, allowing XMLHttpRequests only to `www.example.com`:

```
<iframe     src="http://www.example.com/air/child.html"
            documentRoot="app:/sandbox/"
            sandboxRoot="http://www.example.com/air/"/>
```

Place `child.html` in a remote sandbox, allowing XMLHttpRequests to any remote domain:

```
<iframe     src="http://www.example.com/air/child.html"
            documentRoot="app:/sandbox/"
            sandboxRoot="http://www.example.com/air/"
            allowCrossDomainXHR="allowCrossDomainXHR"/>
```

Place `child.html` in a local-with-file-system sandbox:

```
<iframe     src="file:///templates/child.html"
            documentRoot="app:/sandbox/"
            sandboxRoot="app-storage:/templates/"/>
```

Place `child.html` in a remote sandbox, using the `dominitialize` event to establish a sandbox bridge:

```
<html>
<head>
<script>
var bridgeInterface = {};
bridgeInterface.testProperty = "Bridge engaged";
function engageBridge(){
    document.getElementById("sandbox").parentSandboxBridge = bridgeInterface;
}
</script>
</head>
<body>
<iframe id="sandbox"
            src="http://www.example.com/air/child.html"
            documentRoot="app:/"
            sandboxRoot="http://www.example.com/air/"
            ondominitialize="engageBridge()"/>
</body>
</html>
```

The following `child.html` document illustrates how child content can access the parent sandbox bridge :

```
<html>
```

```
    <head>
        <script>
            document.write(window.parentSandboxBridge.testProperty);
        </script>
    </head>
    <body></body>
</html>
```

For more information, see "Cross-scripting content in different security sandboxes" on page 270 and "HTML security" on page 75.

HTML Canvas element

Defines a drawing area for use with the Webkit Canvas API. Graphics commands cannot be specified in the tag itself. To draw into the canvas, call the canvas drawing methods through JavaScript.

```
<canvas id="drawingAtrium" style="width:300px; height:300px;"></canvas>
```

See also

"The Canvas object" on page 249

HTML element event handlers

DOM objects in AIR and Webkit dispatch some events not found in the standard DOM event model. The following table lists the related event attributes you can use to specify handlers for these events:

Callback attribute name	Description
oncontextmenu	Called when a context menu is invoked, such as through a right-click or command-click on selected text.
oncopy	Called when a selection in an element is copied.
oncut	Called when a selection in an element is cut.
ondominitialize	Called when the DOM of a document loaded in a frame or iframe is created, but before any DOM elements are created or scripts parsed.
ondrag	Called when an element is dragged.
ondragend	Called when a drag is released.
ondragenter	Called when a drag gesture enters the bounds of an element.
ondragleave	Called when a drag gesture leaves the bounds of an element.
ondragover	Called continuously while a drag gesture is within the bounds of an element.
ondragstart	Called when a drag gesture begins.
ondrop	Called when a drag gesture is released while over an element.
onerror	Called when an error occurs while loading an element.
oninput	Called when text is entered into a form element.
onpaste	Called when an item is pasted into an element.
onscroll	Called when the content of a scrollable element is scrolled.
onselectstart	Called when a selection begins.

HTML contentEditable attribute

You can add the `contentEditable` attribute to any HTML element to allow users to edit the content of the element. For example, the following example HTML code sets the entire document as editable, except for first `p` element:

```
<html>
<head/>
<body contentEditable="true">
    <h1>de Finibus Bonorum et Malorum</h1>
    <p contentEditable="false">Sed ut perspiciatis unde omnis iste natus error.</p>
    <p>At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis.</p>
</body>
</html>
```

***Note:** If you set the `document.designMode` property to `on`, then all elements in the document are editable, regardless of the setting of `contentEditable` for an individual element. However, setting `designMode` to `off`, does not disable editing of elements for which `contentEditable` is `true`. See "Document.designMode property" on page 251 for additional information.*

See also

- Apple Safari HTML Reference (http://developer.apple.com/documentation/AppleApplications/Reference/SafariHTMLRef/)

Extensions to CSS

WebKit supports several extended CSS properties. The following table lists the extended properties for which support is established. Additional non-standard properties are available in WebKit, but are not fully supported in AIR, either because they are still under development in WebKit, or because they are experimental features that may be removed in the future.

CSS property name	Values	Description
-webkit-border-horizontal-spacing	Non-negative unit of length	Specifies the horizontal component of the border spacing.
-webkit-border-vertical-spacing	Non-negative unit of length	Specifies the vertical component of the border spacing.
-webkit-line-break	after-white-space, normal	Specifies the line break rule to use for Chinese, Japanese, and Korean (CJK) text.
-webkit-margin-bottom-collapse	collapse, discard, separate	Defines how the bottom margin of a table cell collapses.
-webkit-margin-collapse	collapse, discard, separate	Defines how the top and bottom margins of a table cell collapses.
-webkit-margin-start	Any unit of length.	The width of the starting margin. For left-to-right text, this property overrides the left margin. For right-to-left text, this property overrides the right margin.
-webkit-margin-top-collapse	collapse, discard, separate	Defines how the top margin of a table cell collapses.
-webkit-nbsp-mode	normal, space	Defines the behavior of non-breaking spaces within the enclosed content.

CSS property name	Values	Description
-webkit-padding-start	Any unit of length	Specifies the width of the starting padding. For left-to-right text, this property overrides the left padding value. For right-to-left text, this property overrides the right padding value.
-webkit-rtl-ordering	logical, visual	Overrides the default handling of mixed left-to-right and right-to-left text.
-webkit-text-fill-color	Any named color or numeric color value	Specifies the text fill color.
-webkit-text-security	circle, disc, none, square	Specifies the replacement shape to use in a password input field.
-webkit-user-drag	• auto — Default behavior • element — The entire element is dragged • none — The element cannot be dragged	Overrides the automatic drag behavior.
-webkit-user-modify	read-only, read-write, read-write-plain-text-only	Specifies whether the content of an element can be edited.
-webkit-user-select	• auto — Default behavior • none — The element cannot be selected • text — Only text in the element can be selected	Specifies whether a user can select the content of an element.

For more information, see the Apple Safari CSS Reference (http://developer.apple.com/documentation/AppleApplications/Reference/SafariCSSRef/).

Chapter 24: Programming in HTML and JavaScript

A number of programming topics are unique to developing Adobe® AIR™ applications with HTML and JavaScript. The following information is important whether you are programming an HTML-based AIR application or programming a SWF-based AIR application that runs HTML and JavaScript using the HTMLLoader class (or mx:HTML Flex™ component).

Contents

About the HTMLLoader class

The HTMLLoader class of Adobe AIR defines the display object that can display HTML content in an AIR application. SWF-based applications can add an HTMLLoader control to an existing window or create an HTML window that automatically contains a HTMLLoader object with `HTMLLoader.createRootWindow()`. The HTMLLoader object can be accessed through the JavaScript `window.htmlLoader` property from within the loaded HTML page.

Contents

Loading HTML content from a URL

The following code loads a URL into an HTMLLoader object and sets the object as a child of a Sprite object:

```
var container:Sprite;
var html:HTMLLoader = new HTMLLoader;
html.width = 400;
html.height = 600;
var urlReq:URLRequest = new URLRequest("http://www.adobe.com/");
```

```
html.load(urlReq);
container.addChild(html);
```

An HTMLLoader object's `width` and `height` properties are both set to 0 by default. You will want to set these dimensions when adding an HTMLLoader object to the stage. The HTMLLoader dispatches several events as a page loads. You can use these events to determine when it is safe to interact with the loaded page. These events are described in "Handling HTML-related events" on page 274.

***Note:** In the Flex framework, only classes that extend the UIComponent class can be added as children of a Flex Container components. For this reason, you cannot directly add an HTMLLoader as a child of a Flex Container component; however you can use the Flex mx:HTML control, you can build a custom class that extends UIComponent and contains an HTMLLoader as a child of the UIComponent, or you can add the HTMLLoader as a child of a UIComponent and add the UIComponent to the Flex container.* For more information, see "Using the Flex AIR components" on page 38.

You can also render HTML text by using the TextField class, but its capabilities are limited. The Adobe® Flash® Player's TextField class supports a subset of HTML markup, but because of size limitations, its capabilities are limited. (The HTMLLoader class included in Adobe AIR is not available in Flash Player.)

Loading HTML content from a string

The `loadString()` method of an HTMLLoader object loads a string of HTML content into the HTMLLoader object:

```
var html:HTMLLoader = new HTMLLoader();
var htmlStr:String = "<html><body>Hello <b>world</b>.</body></html>";
html.loadString(htmlStr);
```

Content loaded via the `loadString()` method is put in the application security sandbox, giving it full access to AIR APIs.

Important security rules when using HTML in AIR applications

The files you install with the AIR application have access to the AIR APIs. For security reasons, content from other sources do not. For example, this restriction prevents content from a remote domain (such as http://example.com) from reading the contents the user's desktop directory (or worse).

Because there are security loopholes that can be exploited through calling the `eval()` function (and related APIs), content installed with the application, by default, is restricted from using these methods. However, some Ajax frameworks use the calling the `eval()` function and related APIs.

To properly structure content to work in an AIR application, you must take the rules for the security restrictions on content from different sources into account. Content from different sources is placed in separate security classifications, called sandboxes (see "Sandboxes" on page 73). By default, content installed with the application is installed in a sandbox known as the *application* sandbox, and this grants it access to the AIR APIs. The application sandbox is generally the most secure sandbox, with restrictions designed to prevent the execution of untrusted code.

The runtime allows you to load content installed with your application into a sandbox other than the application sandbox. Content in non-application sandboxes operates in a security environment similar to that of a typical web browser. For example, code in non-application sandboxes can use `eval()` and related methods (but at the same time is not allowed to access the AIR APIs). The runtime includes ways to have content in different sandboxes communicate securely (without exposing AIR APIs to non-application content, for example). For details, see "Cross-scripting content in different security sandboxes" on page 270.

If you call code that is restricted from use in a sandbox for security reasons, the runtime dispatches a JavaScript error: "Adobe AIR runtime security violation for JavaScript code in the application security sandbox."

To avoid this error, follow the coding practices described in the next section, "Avoiding security-related JavaScript errors" on page 260.

For more information, see "HTML security" on page 75.

Avoiding security-related JavaScript errors

If you call code that is restricted from use in a sandbox due to these security restrictions, the runtime dispatches a JavaScript error: "Adobe AIR runtime security violation for JavaScript code in the application security sandbox." To avoid this error, follow these coding practices.

Contents

Causes of security-related JavaScript errors

Code executing in the application sandbox is restricted from most operations that involve evaluating and executing strings once the document `load` event has fired and any `load` event handlers have exited. Attempting to use the following types of JavaScript statements that evaluate and execute potentially insecure strings generates JavaScript errors:

- eval() function
- setTimeout() and setInterval()
- Function constructor

In addition, the following types of JavaScript statements fail without generating an unsafe JavaScript error:

- javascript: URLs
- Event callbacks assigned through onevent attributes in innerHTML and outerHTML statements
- Loading JavaScript files from outside the application installation directory
- document.write() and document.writeln()
- Synchronous XMLHttpRequests before the `load` event or during a `load` event handler
- Dynamically created script elements

Note: *In some restricted cases, evaluation of strings is permitted. See "Code restrictions for content in different sandboxes" on page 77 for more information.*

Adobe maintains a list of Ajax frameworks known to support the application security sandbox, at http://www.adobe.com/go/airappsandboxframeworks.

The following sections describe how to rewrite scripts to avoid these unsafe JavaScript errors and silent failures for code running in the application sandbox.

Mapping application content to a different sandbox

In most cases, you can rewrite or restructure an application to avoid security-related JavaScript errors. However, when rewriting or restructuring is not possible, you can load the application content into a different sandbox using the technique described in "Loading application content into a non-application sandbox" on page 271. If that content also must access AIR APIs, you can create a sandbox bridge, as described in "Setting up a sandbox bridge interface" on page 272.

eval() function

In the application sandbox, the `eval()` function can only be used before the page `load` event or during a `load` event handler. After the page has loaded, calls to `eval()` will not execute code. However, in the following cases, you can rewrite your code to avoid the use of `eval()`.

Assigning properties to an object

Instead of parsing a string to build the property accessor:

```
eval("obj." + propName + " = " + val);
```

access properties with bracket notation:

```
obj[propName] = val;
```

Creating a function with variables available in context

Replace statements such as the following:

```
function compile(var1, var2){
    eval("var fn = function(){ this."+var1+"(var2) }");
    return fn;
}
```

with:

```
function compile(var1, var2){
    var self = this;
    return function(){ self[var1](var2) };
}
```

Creating an object using the name of the class as a string parameter

Consider a hypothetical JavaScript class defined with the following code:

```
var CustomClass = 
    {
        Utils:
        {
            Parser: function(){ alert('constructor') }
        },
        Data:
```

```
        {

        }
    };
```

```
var constructorClassName = "CustomClass.Utils.Parser";
```

The simplest way to create a instance would be to use `eval()`:

```
var myObj;
eval('myObj=new ' + constructorClassName +'()')
```

However, you could avoid the call to `eval()` by parsing each component of the class name and building the new object using bracket notation:

```
function getter(str)
{
    var obj = window;
    var names = str.split('.');
    for(var i=0;i<names.length;i++){
        if(typeof obj[names[i]]=='undefined'){
            var undefstring = names[0];
            for(var j=1;j<=i;j++)
                undefstring+="."+names[j];
            throw new Error(undefstring+" is undefined");
        }
        obj = obj[names[i]];
    }
    return obj;
}
```

To create the instance, use:

```
try{
    var Parser = getter(constructorClassName);
    var a = new Parser();
    }catch(e){
        alert(e);
}
```

setTimeout() and setInterval()

Replace the string passed as the handler function with a function reference or object. For example, replace a statement such as:

```
setTimeout("alert('Timeout')", 10);
```

with:

```
setTimeout(alert('Timeout'), 10);
```

Or, when the function requires the `this` object to be set by the caller, replace a statement such as:

```
this.appTimer = setInterval("obj.customFunction();", 100);
```

with the following:

```
var _self = this;
this.appTimer = setInterval(function(){obj.customFunction.apply(_self);}, 100);
```

Function constructor

Calls to `new Function(param, body)` can be replaced with an inline function declaration or used only before the page `load` event has been handled.

javascript: URLs

The code defined in a link using the javascript: URL scheme is ignored in the application sandbox. No unsafe JavaScript error is generated. You can replace links using javascript: URLs, such as:

```
<a href="javascript:code()">Click Me</a>
```

with:

```
<a href="#" onclick="code()">Click Me</a>
```

Event callbacks assigned through `onevent` attributes in innerHTML and outerHTML statements

When you use innerHTML or outerHTML to add elements to the DOM of a document, any event callbacks assigned within the statement, such as `onclick` or `onmouseover`, are ignored. No security error is generated. Instead, you can assign an `id` attribute to the new elements and set the event handler callback functions using the `addEventListener()` method.

For example, given a target element in a document, such as:

```
<div id="container"></div>
```

Replace statements such as:

```
document.getElementById('container').innerHTML = 
    '<a href="#" onclick="code()">Click Me.</a>';
```

with:

```
document.getElementById('container').innerHTML = '<a href="#" id="smith">Click Me.</a>';
document.getElementById('smith').addEventListener("click", function() { code(); });
```

Loading JavaScript files from outside the application installation directory

Loading script files from outside the application sandbox is not permitted. No security error is generated. All script files that run in the application sandbox must be installed in the application directory. To use external scripts in a page, you must map the page to a different sandbox. See “Loading application content into a non-application sandbox” on page 271.

document.write() and document.writeln()

Calls to `document.write()` or `document.writeln()` are ignored after the page `load` event has been handled. No security error is generated. As an alternative, you can load a new file, or replace the body of the document using DOM manipulation techniques.

Synchronous XMLHttpRequests before the `load` event or during a `load` event handler

Synchronous XMLHttpRequests initiated before the page `load` event or during a `load` event handler do not return any content. Asynchronous XMLHttpRequests can be initiated, but do not return until after the `load` event. After the `load` event has been handled, synchronous XMLHttpRequests behave normally.

Dynamically created script elements

Dynamically created script elements, such as when created with innerHTML or `document.createElement()` method are ignored.

See also

- "HTML security" on page 75

Accessing AIR API classes from JavaScript

In addition to the standard and extended elements of Webkit, HTML and JavaScript code can access the host classes provided by the runtime. These classes let you access the advanced features that AIR provides, including:

- Access to the file system
- Use of local SQL databases
- Control of application and window menus
- Access to sockets for networking
- Use of user-defined classes and objects
- Sound capabilities

For example, the AIR file API includes a File class, contained in the flash.filesystem package. You can create a File object in JavaScript as follows:

```
var myFile = new window.runtime.flash.filesystem.File();
```

The `runtime` object is a special JavaScript object, available to HTML content running in AIR in the application sandbox. It lets you access runtime classes from JavaScript. The `flash` property of the `runtime` object provides access to the flash package. In turn, the `flash.filesystem` property of the `runtime` object provides access to the flash.filesystem package (and this package includes the File class). Packages are a way of organizing classes used in ActionScript.

Note: *The `runtime` property is not automatically added to windows loaded in a frame or iframe. However, as long as the child document is in the application sandbox, the child can access the `runtime` property of the parent.*

Because the package structure of the runtime classes would require developers to type long strings of JavaScript code strings to access each class (as in `window.runtime.flash.desktop.NativeApplication`), the AIR SDK includes an AIRAliases.js file that lets you access runtime classes much more easily (for instance, by simply typing `air.NativeApplication`).

The AIR API classes are discussed throughout this guide. Other classes from the Flash Player API, which may be of interest to HTML developers, are described in the *Adobe AIR Language Reference for HTML Developers*. Action-Script is the language used in SWF (Flash Player) content. However, JavaScript and ActionScript syntax are similar. (They are both based on versions of the ECMAScript language.) All built-in classes are available in both JavaScript (in HTML content) and ActionScript (in SWF content).

Note: *JavaScript code cannot use the Dictionary, XML, and XMLList classes, which are available in ActionScript.*

For more information, see:

- "ActionScript 3.0 classes, packages, and namespaces " on page 6
- "ActionScript basics for JavaScript developers" on page 4

Using the AIRAliases.js file

The runtime classes are organized in a package structure, as in the following:

- `window.runtime.flash.desktop.NativeApplication`
- `window.runtime.flash.desktop.ClipboardManager`

- `window.runtime.flash.filesystem.FileStream`
- `window.runtime.flash.data.SQLDatabase`

Included in the AIR SDK is an AIRAliases.js file that provide "alias" definitions that let you access the runtime classes with less typing. For example, you can access the classes listed above by simply typing the following:

- `air.NativeApplication`
- `air.Clipboard`
- `air.FileStream`
- `air.SQLDatabase`

This list is just a short subset of the classes in the AIRAliases.js file. The complete list of classes and package-level functions is provided in the *Adobe AIR Language Reference for HTML Developers*.

In addition to commonly used runtime classes, the AIRAliases.js file includes aliases for commonly used package-level functions: `window.runtime.trace()`, `window.runtime.flash.net.navigateToURL()`, and `window.runtime.flash.net.sendToURL()`, which are aliased as `air.trace()`, `air.navigateToURL()`, and `air.sendToURL()`.

To use the AIRAliases.js file, include the following `script` reference in your HTML page:

```
<script src="AIRAliases.js"></script>
```

Adjust the path in the `src` reference, as needed.

Important: *Except where noted, the JavaScript example code in this documentation assumes that you have included the AIRAliases.js file in your HTML page.*

About URLs in AIR

In HTML content running in AIR, you can use any of the following URL schemes in defining `src` attributes for `img`, `frame`, `iframe`, and `script` tags, in the `href` attribute of a `link` tag, or anywhere else you can provide a URL.

URL scheme	Description	Example
file	A path relative to the root of the file system.	`file:///c:/AIR Test/test.txt`
app	A path relative to the root directory of the installed application.	`app:/images`
app-storage	A path relative to the application store directory. For each installed application, AIR defines a unique application store directory, which is a useful place to store data specific to that application.	`app-storage:/settings/prefs.xml`
http	A standard HTTP request.	`http://www.adobe.com`
https	A standard HTTPS request.	`https://secure.example.com`

For more information about using URL schemes in AIR, see "Using AIR URL schemes in URLs" on page 326.

Many of AIR APIs, including the File, Loader, URLStream, and Sound classes, use a URLRequest object rather than a string containing the URL. The URLRequest object itself is initialized with a string, which can use any of the same url schemes. For example, the following statement creates a URLRequest object that can be used to request the Adobe home page:

```
var urlReq = new air.URLRequest("http://www.adobe.com/");
```

For information about URLRequest objects see "URL requests and networking" on page 324.

Making ActionScript objects available to JavaScript

JavaScript in the HTML page loaded by an HTMLLoader object can call the classes, objects, and functions defined in the ActionScript execution context using the `window.runtime`, `window.htmlLoader`, and `window.nativeWindow` properties of the HTML page. You can also make ActionScript objects and functions available to JavaScript code by creating references to them within the JavaScript execution context.

Contents

A basic example of accessing JavaScript objects from ActionScript

The following example illustrates how to add properties referencing ActionScript objects to the global window object of an HTML page:

```
var html:HTMLLoader = new HTMLLoader();
var foo:String = "Hello from container SWF."
function helloFromJS(message:String):void {
    trace("JavaScript says:", message);
}
var urlReq:URLRequest = new URLRequest("test.html");
html.addEventListener(Event.COMPLETE, loaded);
html.load(urlReq);

function loaded(e:Event):void{
    html.window.foo = foo;
    html.window.helloFromJS = helloFromJS;
}
```

The HTML content (in a file named test.html) loaded into the HTMLLoader object in the previous example can access the `foo` property and the `helloFromJS()` method defined in the parent SWF file:

```
<html>
    <script>
        function alertFoo() {
            alert(foo);
        }
    </script>
    <body>
        <button onClick="alertFoo()">
            What is foo?
        </button>
        <p><button onClick="helloFromJS('Hi.')">
            Call helloFromJS() function.
        </button></p>
    </body>
</html>
```

When accessing the JavaScript context of a loading document, you can use the `htmlDOMInitialize` event to create objects early enough in the page construction sequence that any scripts defined in the page can access them. If you wait for the `complete` event, only scripts in the page that run after the page `load` event can access the added objects.

Making class definitions available to JavaScript

To make the ActionScript classes of your application available in JavaScript, you can assign the loaded HTML content to the application domain containing the class definitions. The application domain of the JavaScript execution context can be set with the `runtimeApplicationDomain` property of the HTMLLoader object. To set the application domain to the primary application domain, for example, set `runtimeApplicationDomain` to `ApplicationDomain.currentDomain`, as shown in the following code:

```
html.runtimeApplicationDomain = ApplicationDomain.currentDomain;
```

Once the `runtimeApplicationDomain` property is set, the JavaScript context shares class definitions with the assigned domain. To create an instance of a custom class in JavaScript, reference the class definition through the `window.runtime` property and use the `new` operator:

```
var customClassObject = new window.runtime.CustomClass();
```

The HTML content must be from a compatible security domain. If the HTML content is from a different security domain than that of the application domain you assign, the page uses a default application domain instead. For example, if you load a remote page from the Internet, you could not assign `ApplicationDomain.currentDomain` as the application domain of the page.

Removing event listeners

When you add JavaScript event listeners to objects outside the current page, including runtime objects, objects in loaded SWF content, and even JavaScript objects running in other pages, you should always remove those event listeners when the page unloads. Otherwise, the event listener dispatches the event to a handler function that no longer exists. If this happens, you will see the following error message: “The application attempted to reference a JavaScript object in an HTML page that is no longer loaded." Removing unneeded event listeners also lets AIR reclaim the associated memory. For more information, see “Removing event listeners in HTML pages that navigate” on page 278.

Accessing HTML DOM and JavaScript objects from ActionScript

Once the HTMLLoader object dispatches the `complete` event, you can access all the objects in the HTML DOM (document object model) for the page. Accessible objects include display elements (such as `div` and `p` objects in the page) as well as JavaScript variables and functions. The `complete` event corresponds to the JavaScript page `load` event. Before `complete` is dispatched, DOM elements, variables, and functions may not have been parsed or created. If possible, wait for the `complete` event before accessing the HTML DOM.

For example, consider the following HTML page:

```
<html>
    <script>
        foo = 333;
        function test() {
            return "OK.";
        }
    </script>
    <body>
        <p id="p1">Hi.</p>
    </body>
</html>
```

This simple HTML page defines a JavaScript variable named *foo* and a JavaScript function named *test()*. Both of these are properties of the global `window` object of the page. Also, the `window.document` object includes a named P element (with the ID *p1*), which you can access using the `getElementById()` method. Once the page is loaded (when the HTMLLoader object dispatches the `complete` event), you can access each of these objects from ActionScript, as shown in the following ActionScript code:

```
var html:HTMLLoader = new HTMLLoader();
html.width = 300;
html.height = 300;
html.addEventListener(Event.COMPLETE, completeHandler);
var xhtml:XML =
    <html>
        <script>
            foo = 333;
            function test() {
                return "OK.";
            }
        </script>
        <body>
            <p id="p1">Hi.</p>
        </body>
    </html>;
html.loadString(xhtml.toString());

function completeHandler(e:Event):void {
    trace(html.window.foo); // 333
    trace(html.window.document.getElementById("p1").innerHTML); // Hi.
    trace(html.window.test()); // OK.
}
```

To access the content of an HTML element, use the `innerHTML` property. For example, the previous code uses `html.window.document.getElementById("p1").innerHTML` to get the contents of the HTML element named *p1*.

You can also set properties of the HTML page from ActionScript. For example, the following example sets the contents of the `p1` element and the value of the `foo` JavaScript variable on the page using a reference to the containing HTMLLoader object:

```
html.window.document.getElementById("p1").innerHTML = "Goodbye";
html.window.foo = 66;
```

Using ActionScript libraries within an HTML page

AIR extends the HTML script element so that a page can import ActionScript classes in a compiled SWF file. For example, to import a library named, *myClasses.swf*, located in the `lib` subdirectory of the root application folder, include the following script tag within an HTML file:

```
<script src="lib/myClasses.swf" type="application/x-shockwave-flash"></script>
```

Important: *The type attribute must be* `type="application/x-shockwave-flash"` *for the library to be properly loaded.*

The `lib` directory and `myClasses.swf` file must also be included when the AIR file is packaged.

Access the imported classes through the `runtime` property of the JavaScript Window object:

```
var libraryObject = new window.runtime.LibraryClass();
```

If the classes in the SWF file are organized in packages, you must include the package name as well. For example, if the LibraryClass definition was in a package named *utilities*, you would create an instance of the class with the following statement:

```
var libraryObject = new window.runtime.utilities.LibraryClass();
```

Note: *To compile an ActionScript SWF library for use as part of an HTML page in AIR, use the* `acompc` *compiler.*

Converting Date and RegExp objects

The JavaScript and ActionScript languages both define Date and RegExp classes, but objects of these types are not automatically converted between the two execution contexts. You must convert Date and RegExp objects to the equivalent type before using them to set properties or function parameters in the alternate execution context.

For example, the following ActionScript code converts a JavaScript Date object named `jsDate` to an ActionScript Date object:

```
var asDate:Date = new Date(jsDate.getMilliseconds());
```

The following ActionScript code converts a JavaScript RegExp object named `jsRegExp` to an ActionScript RegExp object:

```
var flags:String = "";
if (jsRegExp.dotAll) flags += "s";
if (jsRegExp.extended) flags += "x";
if (jsRegExp.global) flags += "g";
if (jsRegExp.ignoreCase) flags += "i";
if (jsRegExp.multiline) flags += "m";
var asRegExp:RegExp = new RegExp(jsRegExp.source, flags);
```

Manipulating an HTML stylesheet from ActionScript

Once the HTMLLoader object has dispatched the `complete` event, you can examine and manipulate CSS styles in a page.

For example, consider the following simple HTML document:

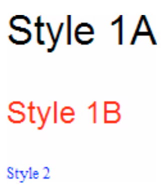
```
<html>
<style>
    .style1A { font-family:Arial; font-size:12px }
    .style1B { font-family:Arial; font-size:24px }
</style>
<style>
    .style2 { font-family:Arial; font-size:12px }
</style>
<body>
    <p class="style1A">
        Style 1A
    </p>
    <p class="style1B">
        Style 1B
    </p>
    <p class="style2">
        Style 2
    </p>
```

```
</body>
</html>
```

After an HTMLLoader object loads this content, you can manipulate the CSS styles in the page via the `cssRules` array of the `window.document.styleSheets` array, as shown here:

```
var html:HTMLLoader = new HTMLLoader( );
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.addEventListener(Event.COMPLETE, completeHandler);
function completeHandler(event:Event):void {
    var styleSheet0:Object = html.window.document.styleSheets[0];
    styleSheet0.cssRules[0].style.fontSize = "32px";
    styleSheet0.cssRules[1].style.color = "#FF0000";
    var styleSheet1:Object = html.window.document.styleSheets[1];
    styleSheet1.cssRules[0].style.color = "blue";
    styleSheet1.cssRules[0].style.font-family = "Monaco";
}
```

This code adjusts the CSS styles so that the resulting HTML document appears like the following:


Style 1A

Style 1B

Style 2

Keep in mind that code can add styles to the page after the HTMLLoader object dispatches the `complete` event.

Cross-scripting content in different security sandboxes

The runtime security model isolates code from different origins. By cross-scripting content in different security sandboxes, you can allow content in one security sandbox to access selected properties and methods in another sandbox.

Contents

AIR security sandboxes and JavaScript code

AIR enforces a same-origin policy that prevents code in one domain from interacting with content in another. All files are placed in a sandbox based on their origin. Ordinarily, content in the application sandbox cannot violate the same-origin principle and cross-script content loaded from outside the application install directory. However, AIR provides two techniques that let you cross-script non-application content.

One technique uses frames or iframes to map application content into a different security sandbox. The code in the application content loaded this way can then interact with content that is actually in that security sandbox. For example, by mapping application content to the *example.com* domain, that content could cross-script pages loaded from example.com.

Since this technique places the application content into a different sandbox, code within that content is also no longer subject to the restrictions on the execution of code in evaluated strings. You can use this sandbox mapping technique to ease these restrictions even when you don't need to cross-script remote content. Mapping content in this way can be especially useful when working with one of the many JavaScript frameworks or with existing code that relies on evaluating strings. However, you should consider and guard against the additional risk that untrusted content could be injected and executed when content is run outside the application sandbox.

At the same time, application content mapped to another sandbox loses its access to the AIR APIs, so the sandbox mapping technique cannot be used to expose AIR functionality to code executed outside the application sandbox.

The second technique lets you create an interface called a *sandbox bridge* between content in a non-application sandbox and its parent document in the application sandbox. The bridge allows the child content to access properties and methods defined by the parent, the parent to access properties and methods defined by the child, or both.

For more information, see "HTML frame and iframe elements" on page 253 and "HTML security" on page 75.

Loading application content into a non-application sandbox

To allow application content to safely cross-script content loaded from outside the application install directory, you can use `frame` or `iframe` elements to load application content into the same security sandbox as the external content. If you do not need to cross-script remote content, but still wish to load a page of your application outside the application sandbox, you can use the same technique, specifying `http://localhost/` or some other innocuous value, as the domain of origin.

AIR adds the new attributes, `sandboxRoot` and `documentRoot`, to the frame element that allow you to specify whether an application file loaded into the frame should be mapped to a non-application sandbox. Files resolving to a path underneath the `sandboxRoot` URL are loaded instead from the `documentRoot` directory. For security purposes, the application content loaded in this way is treated as if it was actually loaded from the `sandboxRoot` URL.

The `sandboxRoot` property specifies the URL to use for determining the sandbox and domain in which to place the frame content. The `file:`, `http:`, or `https:` URL schemes must be used. If you specify a relative URL, the content remains in the application sandbox.

The `documentRoot` property specifies the directory from which to load the frame content. The `file:`, `app:`, or `app-storage:` URL schemes must be used.

The following example maps content installed in the `sandbox` subdirectory of the application to run in the remote sandbox and the `www.example.com` domain:

```
<iframe
    src="http://www.example.com/local/ui.html"
    sandboxRoot="http://www.example.com/local/"
    documentRoot="app:/sandbox/">
</iframe>
```

***Note:** If the sandboxRoot URL maps to a real URL on the remote server, you cannot access content from that URL (or any of its subdirectories) because AIR remaps the request to the local application directory. Requests are remapped whether they derive from page navigation, from an XMLHttpRequest, or from any other means of loading content.*

Setting up a sandbox bridge interface

You can use a sandbox bridge when content in the application sandbox must access properties or methods defined by content in a non-application sandbox, or when non-application content must access properties and methods defined by content in the application sandbox. Create a bridge with the `childSandboxBridge` and `parentSandboxBridge` properties of the `window` object of any child document.

Establishing a child sandbox bridge

The `childSandboxBridge` property allows the child document to expose an interface to content in the parent document. To expose an interface, you set the `childSandbox` property to a function or object in the child document. You can then access the object or function from content in the parent document. The following example shows how a script running in a child document can expose an object containing a function and a property to its parent:

```
var interface = {};
interface.calculatePrice = function(){
    return ".45 cents";
}
interface.storeID = "abc"
window.childSandboxBridge = interface;
```

If this child content was loaded into an iframe assigned an id of "child", you could access the interface from parent content by reading the `childSandboxBridge` property of the frame:

```
var childInterface = document.getElementById("child").contentWindow.childSandboxBridge;
air.trace(childInterface.calculatePrice()); //traces ".45 cents"
air.trace(childInterface.storeID)); //traces "abc"
```

Establishing a parent sandbox bridge

The `parentSandboxBridge` property allows the parent document to expose an interface to content in a child document. To expose an interface, the parent document sets the `parentSandbox` property of the child document to a function or object defined in the parent document. You can then access the object or function from content in the child. The following example shows how a script running in a parent frame can expose an object containing a function to a child document:

```
var interface = {};
interface.save = function(text){
    var saveFile = air.File("app-storage:/save.txt");
    //write text to file
}
document.getElementById("child").contentWindow.parentSandboxBridge = interface;
```

Using this interface, content in the child frame could save text to a file named `save.txt`, but would not have any other access to the file system. The child content could call the save function as follows:

```
var textToSave = "A string.";
window.parentSandboxBridge.save(textToSave);
```

Application content should expose the narrowest interface possible to other sandboxes. Non-application content should be considered inherently untrustworthy since it may be subject to accidental or malicious code injection. You must put appropriate safeguards in place to prevent misuse of the interface you expose through the parent sandbox bridge.

Accessing a parent sandbox bridge during page loading

In order for a script in a child document to access a parent sandbox bridge, the bridge must be set up before the script is run. Window, frame and iframe objects dispatch a `dominitialize` event when a new page DOM has been created, but before any scripts have been parsed, or DOM elements added. You can use the `dominitialize` event to establish the bridge early enough in the page construction sequence that all scripts in the child document can access it.

The following example illustrates how to create a parent sandbox bridge in response to the `dominitialize` event dispatched from the child frame:

```
<html>
<head>
<script>
var bridgeInterface = {};
bridgeInterface.testProperty = "Bridge engaged";
function engageBridge(){
    document.getElementById("sandbox").contentWindow.parentSandboxBridge = bridgeInterface;
}
</script>
</head>
<body>
<iframe id="sandbox"
            src="http://www.example.com/air/child.html"
            documentRoot="app:/"
            sandboxRoot="http://www.example.com/air/"
            ondominitialize="engageBridge()"/>
</body>
</html>
```

The following `child.html` document illustrates how child content can access the parent sandbox bridge:

```
<html>
    <head>
        <script>
            document.write(window.parentSandboxBridge.testProperty);
        </script>
    </head>
    <body></body>
</html>
```

To listen for the `dominitialize` event on a child window, rather than a frame, you must add the listener to the new child window object created by the `window.open()` function:

```
var childWindow = window.open();
childWindow.addEventListener("dominitialize", engageBridge());
childWindow.document.location = "http://www.example.com/air/child.html";
```

In this case, there is no way to map application content into a non-application sandbox. This technique is only useful when `child.html` is loaded from outside the application directory. You can still map application content in the window to a non-application sandbox, but you must first load an intermediate page that itself uses frames to load the child document and map it to the desired sandbox.

If you use the HTMLLoader class `createRootWindow()` function to create a window, the new window is not a child of the document from which `createRootWindow()` is called. Thus, you cannot create a sandbox bridge from the calling window to non-application content loaded into the new window. Instead, you must use load an intermediate page in the new window that itself uses frames to load the child document. You can then establish the bridge from the parent document of the new window to the child document loaded into the frame.

Chapter 25: Handling HTML-related events

An event-handling system allows programmers to respond to user input and system events in a convenient way. The Adobe® AIR™ event model is not only convenient, but also standards-compliant. Based on the Document Object Model (DOM) Level 3 Events Specification, an industry-standard event-handling architecture, the event model provides a powerful, yet intuitive, event-handling tool for programmers.

Contents

HTMLLoader events

An HTMLLoader object dispatches the following ActionScript™ events:

Event	**Description**
htmlDOMInitialize	Dispatched when the HTML document is created, but before any scripts are parsed or DOM nodes are added to the page.
complete	Dispatched when the HTML DOM has been created in response to a load operation, immediately after the `onload` event in the HTML page.
htmlBoundsChanged	Dispatched when one or both of the `contentWidth` and `contentHeight` properties have changed.
locationChange	Dispatched when the location property of the HTMLLoader has changed.
scroll	Dispatched anytime the HTML engine changes the scroll position. Scroll events can be because of navigation to anchor links (# links) in the page or because of calls to the `window.scrollTo()` method. Entering text in a text input or text area can also cause a scroll event.
uncaughtScriptException	Dispatched when a JavaScript exception occurs in the HTMLLoader and the exception is not caught in Java-Script code.

You can also register an ActionScript function for a JavaScript event (such as `onClick`). For details, see "Handling DOM events with ActionScript" on page 275.

Handling DOM events with ActionScript

You can register ActionScript functions to respond to JavaScript events. For example, consider the following HTML content:

```
<html>
<body>
    <a href="#" id="testLink">Click me.</a>
</html>
```

You can register an ActionScript function as a handler for any event in the page. For example, the following code adds the `clickHandler()` function as the listener for the `onclick` event of the `testLink` element in the HTML page:

```
var html:HTMLLoader = new HTMLLoader( );
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.addEventListener(Event.COMPLETE, completeHandler);

function completeHandler(event:Event):void {
    html.window.document.getElementById("testLink").onclick = clickHandler;
}

function clickHandler():void {
    trace("You clicked it!");
}
```

You can also use the `addEventListener()` method to register for these events. For example, you could replace the `completeHandler()` method in the previous example with the following code:

```
function completeHandler(event:Event):void {
    var testLink:Object = html.window.document.getElementById("testLink");
    testLink.addEventListener("click", clickHandler);
}
```

When a listener refers to a specific DOM element, it is good practice to wait for the parent HTMLLoader to dispatch the `complete` event before adding the event listeners. HTML pages often load multiple files and the HTML DOM is not fully built until all the files are loaded and parsed. The HTMLLoader dispatches the `complete` event when all elements have been created.

Responding to uncaught JavaScript exceptions

Consider the following HTML:

```
<html>
<head>
    <script>
        function throwError() {
            var x = 400 * melbaToast;
        }
    </script>
</head>
<body>
    <a href="#" onclick="throwError()">Click me.</a>
</html>
```

It contains a JavaScript function, `throwError()`, that references an unknown variable, `melbaToast`:

```
var x = 400 * melbaToast;
```

When a JavaScript operation encounters an illegal operation that is not caught in the JavaScript code with a `try/catch` structure, the HTMLLoader object containing the page dispatches an HTMLUncaughtScriptException-Event event. You can register a handler for this event, as in the following code:

```
var html:HTMLLoader = new HTMLLoader();
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.width = container.width;
html.height = container.height;
container.addChild(html);
html.addEventListener(HTMLUncaughtScriptExceptionEvent.UNCAUGHT_SCRIPT_EXCEPTION,
                        htmlErrorHandler);
function htmlErrorHandler(event:HTMLUncaughtJavaScriptExceptionEvent):void
{
    event.preventDefault();
    trace("exceptionValue:", event.exceptionValue)
    for (var i:int = 0; i < event.stackTrace.length; i++)
    {
        trace("sourceURL:", event.stackTrace[i].sourceURL);
        trace("line:", event.stackTrace[i].line);
        trace("function:", event.stackTrace[i].functionName);
    }
}
```

Within JavaScript, you can handle the same event using the window.htmlLoader property:

```
<html>
<head>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>

    <script>
        function throwError() {
            var x = 400 * melbaToast;
        }

        function htmlErrorHandler(event) {
            event.preventDefault();
            var message = "exceptionValue:" + event.exceptionValue + "\n";
            for (var i = 0; i < event.stackTrace.length; i++){
                message += "sourceURL:" + event.stackTrace[i].sourceURL +"\n";
                message += "line:" + event.stackTrace[i].line +"\n";
                message += "function:" + event.stackTrace[i].functionName + "\n";
            }
            alert(message);
        }

        window.htmlLoader.addEventListener("uncaughtScriptException", htmlErrorHandler);
    </script>
</head>
<body>
    <a href="#" onclick="throwError()">Click me.</a>
</html>
```

The `htmlErrorHandler()` event handler cancels the default behavior of the event (which is to send the JavaScript error message to the AIR trace output), and generates its own output message. It outputs the value of the `exception-Value` of the HTMLUncaughtScriptExceptionEvent object. It outputs the properties of each object in the `stack-Trace` array:

```
exceptionValue: ReferenceError: Can't find variable: melbaToast
sourceURL: app:/test.html
line: 5
function: throwError
sourceURL: app:/test.html
line: 10
function: onclick
```

Handling runtime events with JavaScript

The runtime classes support adding event handlers with the `addEventListener()` method. To add a handler function for an event, call the `addEventListener()` method of the object that dispatches the event, providing the event type and the handling function. For example, to listen for the `closing` event dispatched when a user clicks the window close button on the title bar, use the following statement:

```
window.nativeWindow.addEventListener(air.NativeWindow.CLOSING, handleWindowClosing);
```

Creating an event handler function

The following code creates a simple HTML file that displays information about the position of the main window. A handler function named `moveHandler()`, listens for a move event (defined by the NativeWindowBoundsEvent class) of the main window.

```
<html>
    <script src="AIRAliases.js" />
    <script>
        function init() {
            writeValues();
            window.nativeWindow.addEventListener(air.NativeWindowBoundsEvent.MOVE,
                                                     moveHandler);
        }
        function writeValues() {
            document.getElementById("xText").value = window.nativeWindow.x;
            document.getElementById("yText").value = window.nativeWindow.y;
        }
        function moveHandler(event) {
            air.trace(event.type); // move
            writeValues();
        }
    </script>
    <body onload="init()" />
        <table>
            <tr>
                <td>Window X:</td>
                <td><textarea id="xText"></textarea></td>
            </tr>
            <tr>
                <td>Window Y:</td>
                <td><textarea id="yText"></textarea></td>
            </tr>
        </table>
    </body>
</html>
```

When a user moves the window, the textarea elements display the updated X and Y positions of the window:

Notice that the event object is passed as an argument to the `moveHandler()` method. The event parameter allows your handler function to examine the event object. In this example, you use the event object's `type` property to report that the event is a `move` event.

Removing event listeners in HTML pages that navigate

When HTML content navigates, or when HTML content is discarded because a window that contains it is closed, the event listeners that reference objects on the unloaded page are not automatically removed. When an object dispatches an event to a handler that has already been unloaded, you see the following error message: "The application attempted to reference a JavaScript object in an HTML page that is no longer loaded."

To avoid this error, remove JavaScript event listeners in an HTML page before it goes away. In the case of page navigation (within an HTMLLoader object), remove the event listener during the `unload` event of the `window` object.

For example, the following JavaScript code removes an event listener for an `uncaughtScriptException` event:

```
window.onunload = cleanup;
window.htmlLoader.addEventListener('uncaughtScriptException', uncaughtScriptException);
function cleanup()
{
    window.htmlLoader.removeEventListener('uncaughtScriptException',
                            uncaughtScriptExceptionHandler);
}
```

To prevent the error from occurring when closing windows that contain HTML content, call a cleanup function in response to the `closing` event of the NativeWindow object (`window.nativeWindow`). For example, the following JavaScript code removes an event listener for an `uncaughtScriptException` event:

```
window.nativeWindow.addEventListener(air.Event.CLOSING, cleanup);
function cleanup()
{
    window.htmlLoader.removeEventListener('uncaughtScriptException',
                            uncaughtScriptExceptionHandler);
}
```

You can also prevent this error from occurring by removing an event listener as soon as it runs. For example, the following JavaScript code creates an html window by calling the `createRootWindow()` method of the HTMLLoader class and adds an event listener for the `complete` event. When the `complete` event handler is called, it removes its own event listener using the `removeEventListener()` function:

```
var html = runtime.flash.html.HTMLLoader.createRootWindow(true);
html.addEventListener('complete', htmlCompleteListener);
function htmlCompleteListener()
{
    html.removeEventListener(complete, arguments.callee)
    // handler code..
}
html.load(new runtime.flash.net.URLRequest("second.html"));
```

Removing unneeded event listeners also allows the system garbage collector to reclaim any memory associated with those listeners.

Chapter 26: Scripting the HTML Container

The HTMLLoader class serves as the container for HTML content in Adobe® AIR™. The class provides many properties and methods, inherited from the Sprite class, for controlling the behavior and appearance of the object on the ActionScript™ 3.0 display list. In addition, the class defines properties and methods for such tasks as loading and interacting with HTML content and managing history.

The HTMLHost class defines a set of default behaviors for an HTMLLoader. When you create an HTMLLoader object, no HTMLHost implementation is provided. Thus when HTML content triggers one of the default behaviors, such as changing the window location, or the window title, nothing happens. You can extend the HTMLHost class to define the behaviors desired for your application.

A default implementation of the HTMLHost is provided for HTML windows created by AIR. You can assign the default HTMLHost implementation to another HTMLLoader object by setting the `htmlHost` property of the object using a new HTMLHost object created with the `defaultBehavior` parameter set to `true`.

Note: *In the Adobe® Flex™ Framework, the HTMLLoader object is wrapped by the mx:HTML component. When using Flex, you should use the HTML component.*

Contents

Display properties of HTMLLoader objects

An HTMLLoader object inherits the display properties of the Adobe® Flash® Player Sprite class. You can resize, move, hide, and change the background color, for example. Or you can apply advanced effects like filters, masks, scaling, and rotation. When applying effects, consider the impact on legibility. SWF and PDF content loaded into an HTML page cannot be displayed when some effects are applied.

HTML windows contain an HTMLLoader object that renders the HTML content. This object is constrained within the area of the window, so changing the dimensions, position, rotation, or scale factor does not always produce desirable results.

Contents

- “Considerations when loading SWF or PDF content in an HTML page” on page 281
- “Advanced display properties” on page 281

Basic display properties

The basic display properties of the HTMLLoader allow you to position the control within its parent display object, to set the size, and to show or hide the control. You should not change these properties for the HTMLLoader object of an HTML window.

The basic properties include:

Property	Notes
x, y	Positions the object within its parent container.
width, height	Changes the dimensions of the display area.
visible	Controls the visibility of the object and any content it contains.

Outside of an HTML window, the `width` and `height` properties of an HTMLLoader object default to 0. You must set the width and height before the loaded HTML content can be seen. HTML content is drawn to the HTMLLoader size, laid out according to the HTML and CSS properties in the content. Changing the HTMLLoader size reflows the content.

When loading content into a new HTMLLoader object (with `width` still set to 0), it can be tempting to set the display `width` and `height` of the HTMLLoader using the `contentWidth` and `contentHeight` properties. This technique works for pages that have a reasonable minimum width when laid out according the HTML and CSS flow rules. However, some pages flow into a long and narrow layout in the absence of a reasonable width provided by the HTMLLoader.

***Note:** When you change the width and height of an HTMLLoader object, the scaleX and scaleY values do not change, as would happen with most other types of display objects.*

Transparency of HTMLLoader content

The `paintsDefaultBackground` property of an HTMLLoader object, which is `true` by default, determines whether the HTMLLoader object draws an opaque background. When `paintsDefaultBackground` is `false`, the background is clear. The display object container or other display objects below the HTMLLoader object are visible behind the foreground elements of the HTML content.

If the body element or any other element of the HTML document specifies a background color (using `style="background-color:gray"`, for instance), then the background of that portion of the HTML is opaque and rendered with the specified background color. If you set the `opaqueBackground` property of the HTMLLoader object, and `paintsDefaultBackground` is `false`, then the color set for the `opaqueBackground` is visible.

***Note:** You can use a transparent, PNG-format graphic to provide an alpha-blended background for an element in an HTML document. Setting the opacity style of an HTML element is not supported.*

Scaling HTMLLoader content

Avoid scaling an HTMLLoader object beyond a scale factor of 1.0. Text in HTMLLoader content is rendered at a specific resolution and appears pixelated if the HTMLLoader object is scaled up. To prevent the HTMLLoader, as well as its contents, from scaling when a window is resized, set the `scaleMode` property of the Stage to `StageScaleMode.NO_SCALE`.

Considerations when loading SWF or PDF content in an HTML page

SWF and PDF content loaded into in an HTMLLoader object disappears in the following conditions:

- If you scale the HTMLLoader object to a factor other that 1.0.
- If you set the alpha property of the HTMLLoader object to a value other than 1.0.
- If you rotate the HTMLLoader content.

The content reappears if you remove the offending property setting and remove the active filters.

***Note:** The runtime cannot display SWF or PDF content in transparent windows.*

For more information on loading these types of media in an HTMLLoader, see "Loading SWF content within an HTML page" on page 112 and "Adding PDF content" on page 293.

Advanced display properties

The HTMLLoader class inherits several methods that can be used for special effects. In general, these effects have limitations when used with the HTMLLoader display, but they can be useful for transitions or other temporary effects. For example, if you display a dialog window to gather user input, you could blur the display of the main window until the user closes the dialog. Likewise, you could fade the display out when closing a window.

The advanced display properties include:

Property	**Limitations**
alpha	Can reduce the legibility of HTML content
filters	In an HTML Window, exterior effects are clipped by the window edge
graphics	Shapes drawn with graphics commands appear below HTML content, including the default background. The paintsDefaultBackground property must be false for the drawn shapes to be visible.
opaqueBackground	Does not change the color of the default background. The paintsDefaultBack-ground property must be false for this color layer to be visible.
rotation	The corners of the rectangular HTMLLoader area can be clipped by the window edge. SWF and PDF content loaded in the HTML content is not displayed.
scaleX, scaleY	The rendered display can appear pixelated at scale factors greater than 1. SWF and PDF content loaded in the HTML content is not displayed.
transform	Can reduce legibility of HTML content. The HTML display can be clipped by the window edge. SWF and PDF content loaded in the HTML content is not displayed if the transform involves rotation, scaling, or skewing.

The following example illustrates how to set the `filters` array to blur the entire HTML display:

```
var html:HTMLLoader = new HTMLLoader();
var urlReq:URLRequest = new URLRequest("http://www.adobe.com/");
html.load(urlReq);
html.width = 800;
html.height = 600;

var blur:BlurFilter = new BlurFilter(8);
var filters:Array = [blur];
html.filters = filters;
```

Scrolling HTML content

The HTMLLoader class includes the following properties that let you control the scrolling of HTML content:

Property	Description
contentHeight	The height, in pixels, of the HTML content.
contentWidth	The width, in pixels, of the HTML content.
scrollH	The horizontal scroll position of the HTML content within the HTMLLoader object.
scrollV	The vertical scroll position of the HTML content within the HTMLLoader object.

The following code sets the `scrollV` property so that HTML content is scrolled to the bottom of the page:

```
var html:HTMLLoader = new HTMLLoader();
html.addEventListener(Event.HTML_BOUNDS_CHANGE, scrollHTML);

const SIZE:Number = 600;
html.width = SIZE;
html.height = SIZE;

var urlReq:URLRequest = new URLRequest("http://www.adobe.com");
html.load(urlReq);
this.addChild(html);

function scrollHTML(event:Event):void
{
    html.scrollV = html.contentHeight - SIZE;
}
```

The HTMLLoader does not include horizontal and vertical scroll bars. You can implement scroll bars in ActionScript or by using a Flex component. The Flex HTML component automatically includes scroll bars for HTML content. You can also use the `HTMLLoader.createRootWindow()` method to create a window that contains an HTMLLoader object with scroll bars (see “Creating windows with scrolling HTML content” on page 289).

Accessing the HTML history list

As new pages are loaded in an HTMLLoader object, the runtime maintains a history list for the object. The history list corresponds to the `window.history` object in the HTML page. The HTMLLoader class includes the following properties and methods that let you work with the HTML history list:

Class member	Description
historyLength	The overall length of the history list, including back and forward entries.
historyPosition	The current position in the history list. History items before this position represent “back” navigation, and items after this position represent “forward” navigation.
historyAt()	Returns the URLRequest object corresponding to the history entry at the specified position in the history list.

Class member	Description
historyBack()	Navigates back in the history list, if possible.
historyForward()	Navigates back in the history list, if possible.
historyGo()	Navigates the indicated number of steps in the browser history. Navigates forward if positive, backward if negative. Navigating to zero reloads the page. Specifying a position beyond the end navigates to the end of the list.

Items in the history list are stored as objects of type HistoryListItem. The HistoryListItem class has the following properties:

Property	Description
isPost	Set to `true` if the HTML page includes POST data.
originalUrl	The original URL of the HTML page, before any redirects.
title	The title of the HTML page.
url	The URL of the HTML page.

Setting the user agent used when loading HTML content

The HTMLLoader class has a `userAgent` property, which lets you set the user agent string used by the HTMLLoader. Set the `userAgent` property of the HTMLLoader object before calling the `load()` method. If you set this property on the HTMLLoader instance, then the `userAgent` property of the URLRequest passed to the `load()` method is *not* used.

You can set the default user agent string used by all HTMLLoader objects in an application domain by setting the `URLRequestDefaults.userAgent` property. The static URLRequestDefaults properties apply as defaults for all URLRequest objects, not only URLRequests used with the `load()` method of HTMLLoader objects. Setting the `userAgent` property of an HTMLLoader overrides the default `URLRequestDefaults.userAgent` setting.

If you do not set a user agent value for either the `userAgent` property of the HTMLLoader object or for `URLRequestDefaults.userAgent`, then the default AIR user agent value is used. This default value varies depending on the runtime operating system (such as Mac OS or Windows), the runtime language, and the runtime version, as in the following two examples:

- `"Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/420+ (KHTML, like Gecko) AdobeAIR/1.0"`
- `"Mozilla/5.0 (Windows; U; en) AppleWebKit/420+ (KHTML, like Gecko) AdobeAIR/1.0"`

Setting the character encoding to use for HTML content

An HTML page can specify the character encoding it uses by including `meta` tag, such as the following:

```
meta http-equiv="content-type" content="text/html" charset="ISO-8859-1";
```

Override the page setting to ensure that a specific character encoding is used by setting the `textEncodingOverride` property of the HTMLLoader object:

```
var html:HTMLLoader = new HTMLLoader();
html.textEncodingOverride = "ISO-8859-1";
```

Specify the character encoding for the HTMLLoader content to use when an HTML page does not specify a setting with the `textEncodingFallback` property of the HTMLLoader object:

```
var html:HTMLLoader = new HTMLLoader();
html.textEncodingFallback = "ISO-8859-1";
```

The `textEncodingOverride` property overrides the setting in the HTML page. And the `textEncodingOverride` property and the setting in the HTML page override the `textEncodingFallback` property.

Set the `textEncodingOverride` property or the `textEncodingFallback` property before loading the HTML content.

Defining browser-like user interfaces for HTML content

JavaScript provides several APIs for controlling the window displaying the HTML content. In AIR, these APIs can be overridden by implementing a custom HTMLHost class.

Contents

About extending the HTMLHost class

If, for example, your application presents multiple HTMLLoader objects in a tabbed interface, you may want title changes made by the loaded HTML pages to change the label of the tab, not the title of the main window. Similarly, your code could respond to a `window.moveTo()` call by repositioning the HTMLLoader object in its parent display object container, by moving the window that contains the HTMLLoader object, by doing nothing at all, or by doing something else entirely.

The AIR HTMLHost class controls the following JavaScript properties and methods:

- `window.status`
- `window.document.title`
- `window.location`
- `window.blur()`
- `window.close()`
- `window.focus()`
- `window.moveBy()`
- `window.moveTo()`

- `window.open()`
- `window.resizeBy()`
- `window.resizeTo()`

When you create an HTMLLoader object using `new HTMLLoader()`, the listed JavaScript properties or methods are not enabled. The HTMLHost class provides a default, browser-like implementation of these JavaScript APIs. You can also extend the HTMLHost class to customize the behavior. To create an HTMLHost object supporting the default behavior, set the `defaultBehaviors` parameter to true in the HTMLHost constructor:

```
var defaultHost:HTMLHost = new HTMLHost(true);
```

When you create an HTML window in AIR with the HTMLLoader class `createRootWindow()` method, an HTMLHost instance supporting the default behaviors is assigned automatically. You can change the host object behavior by assigning a different HTMLHost implementation to the `htmlHost` property of the HTMLLoader, or you can assign `null` to disable the features entirely.

***Note:** AIR assigns a default HTMLHost object to the initial window created for an HTML-based AIR application and any windows created by the default implementation of the JavaScript `window.open()` method.*

Example: Extending the HTMLHost class

The following example shows how to customize the way that an HTMLLoader object affects the user interface, by extending the HTMLHost class:

1 Create a class that extends the HTMLHost class (a subclass).

2 Override methods of the new class to handle changes in the user interface-related settings. For example, the following class, CustomHost, defines behaviors for calls to `window.open()` and changes to `window.document.title`. Calls to `window.open()` open the HTML page in a new window, and changes to `window.document.title` (including the setting of the `<title>` element of an HTML page) set the title of that window.

```
package
{
    import flash.html.*;
    import flash.display.StageScaleMode;
    import flash.display.NativeWindow;
    import flash.display.NativeWindowInitOptions;

    public class CustomHost extends HTMLHost
    {
        import flash.html.*;
        override public function
            createWindow(windowCreateOptions:HTMLWindowCreateOptions):HTMLLoader
        {
            var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
            var bounds:Rectangle = new Rectangle(windowCreateOptions.x,
                                            windowCreateOptions.y,
                                            windowCreateOptions.width,
                                            windowCreateOptions.height);
            var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
                                        windowCreateOptions.scrollBarsVisible, bounds);
            htmlControl.htmlHost = new HTMLHostImplementation();
            if(windowCreateOptions.fullscreen){
                htmlControl.stage.displayState =
                    StageDisplayState.FULL_SCREEN_INTERACTIVE;
            }
            return htmlControl;
        }
        override public function updateTitle(title:String):void
```

```
            {
                htmlLoader.stage.nativeWindow.title = title;
            }
        }
    }
```

3 In the code that contains the HTMLLoader (not the code of the new subclass of HTMLHost), create an object of the new class. Assign the new object to the `htmlHost` property of the HTMLLoader. The following Flex code uses the CustomHost class defined in the previous step:

```
    <?xml version="1.0" encoding="utf-8"?>
    <mx:WindowedApplication
        xmlns:mx="http://www.adobe.com/2006/mxml"
        layout="vertical"
        applicationComplete="init()">
        <mx:Script>
            <![CDATA[
                import flash.html.HTMLLoader;
                import CustomHost;
                private function init():void
                {
                    var html:HTMLLoader = new HTMLLoader();
                    html.width = container.width;
                    html.height = container.height;
                    var urlReq:URLRequest = new URLRequest("Test.html");
                    html.htmlHost = new CustomHost();
                    html.load(urlReq);
                    container.addChild(html);
                }
            ]]>
        </mx:Script>
        <mx:UIComponent id="container" width="100%" height="100%"/>
    </mx:WindowedApplication>
```

To test the code described here, include an HTML file with the following content in the application directory:

```
<html>
    <head>
        <title>Test</title>
    </head>
    <script>
        function openWindow()
        {
            window.runtime.trace("in");
            document.title = "foo"
            window.open('Test.html');
            window.runtime.trace("out");
        }
    </script>
    <body>
        <a href="#" onclick="openWindow()">window.open('Test.html')</a>
    </body>
</html>
```

Handling changes to the window.location property

Override the `locationChange()` method to handle changes of the URL of the HTML page. The `locationChange()` method is called when JavaScript in a page changes the value of `window.location`. The following example simply loads the requested URL:

```
override public function updateLocation(locationURL:String):void
{
    htmlLoader.load(new URLRequest(locationURL));
}
```

***Note:** You can use the htmlLoader property of the HTMLHost object to reference the current HTMLLoader object.*

Handling JavaScript calls to window.moveBy(), window.moveTo(), window.resizeTo(), window.resizeBy()

Override the `set windowRect()` method to handle changes in the bounds of the HTML content. The `set windowRect()` method is called when JavaScript in a page calls `window.moveBy()`, `window.moveTo()`, `window.resizeTo()`, or `window.resizeBy()`. The following example simply updates the bounds of the desktop window:

```
override public function set windowRect(value:Rectangle):void
{
    htmlLoader.stage.nativeWindow.bounds = value;
}
```

Handling JavaScript calls to window.open()

Override the `createWindow()` method to handle JavaScript calls to `window.open()`. Implementations of the `createWindow()` method are responsible for creating and returning a new HTMLLoader object. Typically, you would display the HTMLLoader in a new window, but creating a window is not required.

The following example illustrates how to implement the `createWindow()` function using the `HTMLLoader.createRootWindow()` to create both the window and the HTMLLoader object. You can also create a NativeWindow object separately and add the HTMLLoader to the window stage.

```
override public function
createWindow(windowCreateOptions:HTMLWindowCreateOptions):HTMLLoader{
    var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
    var bounds:Rectangle = new Rectangle(windowCreateOptions.x, windowCreateOptions.y,
                              windowCreateOptions.width, windowCreateOptions.height);
    var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
                                  windowCreateOptions.scrollBarsVisible, bounds);
    htmlControl.htmlHost = new HTMLHostImplementation();
    if(windowCreateOptions.fullscreen){
        htmlControl.stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
    }
    return htmlControl;
}
```

***Note:** This example assigns the custom HTMLHost implementation to any new windows created with `window.open()`. You can also use a different implementation or set the `htmlHost` property to null for new windows, if desired.*

The object passed as a parameter to the `createWindow()` method is an HTMLWindowCreateOptions object. The HTMLWindowCreateOptions class includes properties that report the values set in the `features` parameter string in the call to `window.open()`:

HTMLWindowCreateOptions property	Corresponding setting in the features string in the JavaScript call to window.open()
fullscreen	fullscreen
height	height
locationBarVisible	location

HTMLWindowCreateOptions property	Corresponding setting in the features string in the JavaScript call to window.open()
menuBarVisible	menubar
resizeable	resizable
scrollBarsVisible	scrollbars
statusBarVisible	status
toolBarVisible	toolbar
width	width
x	left or screenX
y	top or screenY

The HTMLLoader class does not implement all the features that can be specified in the feature string. Your application must provide scroll bars, location bars, menu bars, status bars, and tool bars when appropriate.

The other arguments to the JavaScript `window.open()` method are handled by the system. A `createWindow()` implementation should not load content in the HTMLLoader object, or set the window title.

Handling JavaScript calls to window.close()

Override the `windowClose()` to handle JavaScript calls to `window.close()` method. The following example closes the desktop window when the `window.close()` method is called:

```
override public function windowClose():void
{
    htmlLoader.stage.nativeWindow.close();
}
```

JavaScript calls to `window.close()` do not have to close the containing window. You could, for example, remove the HTMLLoader from the display list, leaving the window (which may have other content) open, as in the following code:

```
override public function windowClose():void
{
    htmlLoader.parent.removeChild(htmlLoader);
}
```

Handling changes of the window.status property

Override the `updateStatus()` method to handle JavaScript changes to the value of `window.status`. The following example traces the status value:

```
override public function updateStatus(status:String):void
{
    trace(status);
}
```

The requested status is passed as a string to the `updateStatus()` method.

The HTMLLoader object does not provide a status bar.

Handling changes of the window.document.title property

override the `updateTitle()` method to handle JavaScript changes to the value of `window.document.title`. The following example changes the window title and appends the string, "Sample," to the title:

```
override public function updateTitle(title:String):void
{
    htmlLoader.stage.nativeWindow.title = title + " - Sample";
}
```

When `document.title` is set on an HTML page, the requested title is passed as a string to the `updateTitle()` method.

Changes to `document.title` do not have to change the title of the window containing the HTMLLoader object. You could, for example, change another interface element, such as a text field.

Handling JavaScript calls to window.blur() and window.focus()

Override the `windowBlur()` and `windowFocus()` methods to handle JavaScript calls to `window.blur()` and `window.focus()`, as shown in the following example:

```
override public function windowBlur():void
{
    htmlLoader.alpha = 0.5;
}
override public function windowFocus():void
{
    htmlLoader.alpha = 1.0;
    NativeApplication.nativeApplication.activate(htmlLoader.stage.nativeWindow);
}
```

***Note:** AIR does not provide an API for deactivating a window or application.*

Creating windows with scrolling HTML content

The HTMLLoader class includes a static method, `HTMLLoader.createRootWindow()`, which lets you open a new window (represented by a NativeWindow object) that contains an HTMLLoader object and define some user interface settings for that window. The method takes four parameters, which let you define the user interface:

Parameter	Description
visible	A Boolean value that specifies whether the window is initially visible (`true`) or not (`false`).
windowInitOptions	A NativeWindowInitOptions object. The NativeWindowInitOptions class defines initialization options for a NativeWindow object, including the following: whether the window is minimizable, maximizable, or resizable, whether the window has system chrome or custom chrome, whether the window is transparent or not (for windows that do not use system chrome), and the type of window.
scrollBarsVisible	Whether there are scroll bars (`true`) or not (`false`).
bounds	A Rectangle object defining the position and size of the new window.

For example, the following code uses the `HTMLLoader.createRootWindow()` method to create a window with HTMLLoader content that uses scrollbars:

```
var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
var bounds:Rectangle = new Rectangle(10, 10, 600, 400);
var html2:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions, true, bounds);
var urlReq2:URLRequest = new URLRequest("http://www.example.com");
```

```
html2.load(urlReq2);
html2.stage.nativeWindow.activate();
```

***Note:** Windows created by calling `createRootWindow()` directly in JavaScript remain independent from the opening HTML window. The JavaScript Window `opener` and `parent` properties, for example, are `null`. However, if you call `createRootWindow()` indirectly by overriding the HTMLHost `createWindow()` method to call `createRootWindow()`, then `opener` and `parent` do reference the opening HTML window.*

Creating subclasses of the HTMLLoader class

You can create a subclass of the HTMLLoader class, to create new behaviors. For example, you can create a subclass that defines default event listeners for HTMLLoader events (such as those events dispatched when HTML is rendered or when the user clicks a link).

The following example extends the HTMLHost class to provide *normal* behavior when the JavaScript `window.open()` method is called. The example then defines a subclass of HTMLLoader that uses the custom HTMLHost implementation class:

```
package
{
    import flash.html.HTMLLoader;
    public class MyHTMLHost extends HTMLHost
    {
        public function MyHTMLHost()
        {
            super(false);
        }
        override public function createWindow(opts:HTMLWindowCreateOptions):void
        {
            var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
            var bounds:Rectangle = new Rectangle(opts.x, opts.y, opts.width, opts.height);
            var html:HTMLLoader = HTMLLoader.createRootWindow(true,
                                                            initOptions,
                                                            opts.scrollBarsVisible,
                                                            bounds);
            html.stage.nativeWindow.orderToFront();
            return html
    }
}
```

The following defines a subclass of the HTMLLoader class that assigns a MyHTMLHost object to its `htmlHost` property:

```
package
{
    import flash.html.HTMLLoader;
    import MyHTMLHost;
    import HTMLLoader;
    public class MyHTML extends HTMLLoader
    {
        public function MyHTML()
        {
            super();
            htmlHost = new MyHTMLHost();
        }
    }
}
```

For details on the HTMLHost class and the `HTMLLoader.createRootWindow()` method used in this example, see "Defining browser-like user interfaces for HTML content" on page 284.

Part 8: Rich media content

Chapter 27: Adding PDF content

Applications running in Adobe® AIR™ can render not only SWF and HTML content, but also PDF content. AIR applications render PDF content using the HTMLLoader class, the WebKit engine, and the Adobe® Reader® browser plug-in. In an AIR application, PDF content can either stretch across the full height and width of your application or alternatively as a portion of the interface. The Adobe Reader browser plug-in controls display of PDF files in an AIR application, so modifications to the Reader toolbar interface (such as those for position, anchoring, and visibility) persist in subsequent viewing of PDF files in both AIR applications and the browser.

Important: *In order to render PDF content in AIR, the user must have Adobe Reader or Adobe® Acrobat® version 8.1 or higher installed.*

Contents

Detecting PDF Capability

If the user does not have an installed version of Adobe Reader or Adobe Acrobat 8.1 or higher, PDF content is not displayed in an AIR application. To detect if a user can render PDF content, first check the `HTMLLoader.pdfCapability` property. This property is set to one of the following constants of the HTMLPDFCapability class:

Constant	**Description**
HTMLPDFCapability.STATUS_OK	A sufficient version (8.1 or greater) of Adobe Reader is detected and PDF content can be loaded into an HTMLLoader object.
HTMLPDFCapability.ERROR_INSTALLED_READER_NOT_FOUND	No version of Adobe Reader is detected. An HTMLLoader object cannot display PDF content.
HTMLPDFCapability.ERROR_INSTALLED_READER_TOO_OLD	Adobe Reader has been detected, but the version is too old. An HTMLConrol object cannot display PDF content.
HTMLPDFCapability.ERROR_PREFERRED_READER_TOO_OLD	A sufficient version (8.1 or later) of Adobe Reader is detected, but the version of Adobe Reader that is set up to handle PDF content is older than Reader 8.1. An HTMLConrol object cannot display PDF content.

Note: *On Windows, if Adobe Acrobat or Adobe Reader version 7.x or above is currently running on the user's system, that version is used even if a later version that supports loading PDF loaded is installed. In this case, if the value of the* `pdfCampability` *property is* `HTMLPDFCapability.STATUS_OK`*, when an AIR application attempts to load PDF content, the older version of Acrobat or Reader displays an alert (and no exception is thrown in the AIR application). If this is a possible situation for your end users, consider providing them with instructions to close Acrobat while running your application. You may want to display these instructions if the PDF content does not load within an acceptable time frame.*

The following code detects whether a user can display PDF content in an AIR application, and if not traces the error code that corresponds to the HTMLPDFCapability error object:

```
if(HTMLLoader.pdfCapability == HTMLPDFCapability.STATUS_OK)
{
    trace("PDF content can be displayed");
}
else
{
    trace("PDF cannot be displayed. Error code:", HTMLLoader.pdfCapability);
}
```

Loading PDF content

You can add a PDF to an AIR application by creating an HTMLLoader instance, setting its dimensions, and loading the path of a PDF.

The following example loads a PDF from an external site. Replace the URLRequest with the path to an available external PDF.

```
var request:URLRequest = new URLRequest("http://www.example.com/test.pdf");
pdf = new HTMLLoader();
pdf.height = 800;
pdf.width = 600;
pdf.load(request);
container.addChild(pdf);
```

You can also load content from file URLs and AIR-specific URL schemes, such as app and app-storage. For example, the following code loads the test.pdf file in the PDFs subdirectory of the application directory:

app:/js_api_reference.pdf

For more information on AIR URL schemes, see “Using AIR URL schemes in URLs” on page 326.

Scripting PDF content

You can use JavaScript to control PDF content just as you can in a web page in the browser.

JavaScript extensions to Acrobat provide the following features, among others:

- Controlling page navigation and magnification
- Processing forms within the document
- Controlling multimedia events

Full details on JavaScript extensions for Adobe Acrobat are provided at the Adobe Acrobat Developer Center at http://www.adobe.com/devnet/acrobat/javascript.html.

HTML-PDF communication basics

JavaScript in an HTML page can send a message to JavaScript in PDF content by calling the `postMessage()` method of the DOM object representing the PDF content. For example, consider the following embedded PDF content:

```
<object id="PDFObj" data="test.pdf" type="application/pdf" width="100%" height="100%"/>
```

The following JavaScript code in the containing HTML content sends a message to the JavaScript in the PDF file:

```
pdfObject = document.getElementById("PDFObj");
```

```
pdfObject.postMessage(["testMsg", "hello"]);
```

The PDF file can include JavaScript for receiving this message. You can add JavaScript code to PDF files in some contexts, including the document-, folder-, page-, field-, and batch-level contexts. Only the document-level context, which defines scripts that are evaluated when the PDF document opens, is discussed here.

A PDF file can add a `messageHandler` property to the `hostContainer` object. The `messageHandler` property is an object that defines handler functions to respond to messages. For example, the following code defines the function to handle messages received by the PDF file from the host container (which is the HTML content embedding the PDF file):

```
this.hostContainer.messageHandler = {onMessage: myOnMessage};

function myOnMessage(aMessage)
{
    if(aMessage[0] == "testMsg")
    {
        app.alert("Test message: " + aMessage[1]);
    }
    else
    {
        app.alert("Error");
    }
}
```

JavaScript code in the HTML page can call the `postMessage()` method of the PDF object contained in the page. Calling this method sends a message (`"Hello from HTML"`) to the document-level JavaScript in the PDF file:

```
<html>
    <head>
    <title>PDF Test</title>
    <script>
        function init()
        {
            pdfObject = document.getElementById("PDFObj");
            try {
                  pdfObject.postMessage(["alert", "Hello from HTML"]);
            }
            catch (e)
            {
                alert( "Error: \n name = " + e.name + "\n message = " + e.message );
            }
        }
    </script>
    </head>
    <body onload='init()'>
        <object
            id="PDFObj"
            data="test.pdf"
            type="application/pdf"
            width="100%" height="100%"/>
    </body>
</html>
```

For a more advanced example, and for information on using Acrobat 8 to add JavaScript a PDF file, see Cross-scripting PDF content in Adobe AIR.

Scripting PDF content from ActionScript

ActionScript code (in SWF content) cannot directly communicate with JavaScript in PDF content. However, ActionScript can communicate with the JavaScript in the HTML page loaded in an HTMLLoader object that loads PDF content, and that JavaScript code can communicate with the JavaScript in the loaded PDF file. For more information, see "Programming in HTML and JavaScript" on page 258.

Known limitations for PDF content in AIR

PDF content in Adobe AIR has the following limitations:

- PDF content does not display in a window (a NativeWindow object) that is transparent (where the `transparent` property is set to `true`).
- The display order of a PDF file operates differently than other display objects in an AIR application. Although PDF content clips correctly according to HTML display order, it will always sit on top of content in the AIR application's display order.
- PDF content does not display in a window that is in full-screen mode (when the `displayState` property of the Stage is set to `StageDisplayState.FULL_SCREEN` or `StageDisplayState.FULL_SCREEN_INTERACTIVE`).
- The visual properties of an HTMLLoader object that contains a PDF file cannot be changed. Changing an HTMLLoader object's `filters`, `alpha`, `rotation`, or `scaling` properties render the PDF file invisible until the properties are reset.
- The `scaleMode` property of the Stage object of the NativeWindow object containing the PDF content must be set to `StageScaleMode.NO_SCALE`.
- Clicking links to content within the PDF file update the scroll position of the PDF content. Clicking links to content outside the PDF file redirect the HTMLLoader object that contains the PDF (even if the target of a link is a new window).
- PDF commenting workflows do not function in AIR 1.0.

See also

- Cross-scripting PDF content in Adobe AIR

Chapter 28: Using digital rights management

Adobe® Flash® Media Rights Management Server (FMRMS) provides media publishers the ability to distribute content, specifically FLV and MP4 files, and to recuperate production costs through direct (user-paid) or indirect (advertising-paid) compensation by their consumers. The publishers distribute media as encrypted FLVs that can be downloaded and played in Adobe® Media Player™, or any AIR application that makes use of the digital rights management (DRM) API.

With FMRMS, the content providers can use identity-based licensing to protect their content with user credentials. For example, a consumer wants to view a television program, but does not want to watch the accompanying advertisements. To avoid watching the advertisements, the consumer registers and pays the content publisher a premium. The user can then use their authentication credential to gain access and play the program without the commercials. Another consumer may want to view the content offline while traveling with no internet access. After registering and paying the content publisher for the premium service, the user's authentication credential allows them to access and download the program from the publisher's website. The user can then view the content offline during the permitted period. The content is also protected by the user credentials and cannot be shared with other users.

When a user tries to play a DRM-encrypted file, the application contacts the FMRMS which in turn contacts the content publisher's system through their service provider interface (SPI) to authenticate the user and retrieve the license, a voucher that determines whether the user is allowed access to the content and, if so, for how long. The voucher also determines whether the user can access the content offline and, if so, for how long. As such, user credentials are needed to determine access to the encrypted content.

Identity-based licensing also supports anonymous access. For example, anonymous access can be used by the provider to distribute ad-supported content or to allow free access to the current content for a specified number of days. The archive material might be considered premium content that must be paid for and requires user credentials. The content provider can also specify and restrict the type and version of the player needed for their content.

How to enable your AIR application to play content protected with digital rights management encryption is described here. It is not necessary to understand how to encrypt content using DRM, but it is assumed that you have access to DRM-encrypted content and are communicating with FMRMS to authenticate the user and retrieve the voucher.

For an overview of FMRMS, including creating policies, see the documentation included with FMRMS.

For information on Adobe Media Player, see Adobe Media Player Help available within Adobe Media Player.

Contents

Language Reference

- DRMAuthenticateEvent
- DRMErrorEvent

- DRMStatusEvent
- NetStream

More Information

- Adobe AIR Developer Center for Flex (search for 'digital rights management')

Understanding the encrypted FLV workflow

There are four types of events, StatusEvent. DRMAuthenticateEvent, DRMErrorEvent, and DRMStatusEvent, that may be dispatched when an AIR application attempts to play a DRM-encrypted file. To support these files, the application should add event listeners for handling the DRM events.

The following is the workflow of how the AIR application can retrieve and play the content protected with DRM-encryption:

1 The Application, using a NetStream object, attempts to play an FLV or MP4 file. If the content is encrypted, an `events.StatusEvent` event is dispatched with the code, `DRM.encryptedFLV`, indicating the FLV is encrypted.

__Note:__ If an application does not want to play the DRM-encrypted file, it can listen to the status event dispatched when it encounters an encrypted content, then let the user know that the file is not supported and close the connection.

2 If the file is anonymously encrypted, meaning that all users are allowed to view the content without inputting authentication credentials, the AIR application proceeds to the last step of this workflow. However, if the file requires an identity-based license, meaning that the user credential is required, then the NetStream object generates a DRMAuthenticateEvent event object. The user must provide their authentication credentials before playback can begin.

3 The AIR application must provide some mechanism for gathering the necessary authentication credentials. The `usernamePrompt`, `passwordPrompt`, and `urlPrompt` properties of DRMAuthenticationEvent class, provided by the content server, can be used to instruct the end user with information about the data that is required. You can use these properties in constructing a user interface for retrieving the needed user credentials. For example, the `usernamePrompt` value string may state that the user name must be in the form of an e-mail address.

__Note:__ AIR does not supply a default user interface for gathering authentication credentials. The application developer must write the user interface and handle the DRMAuthenticateEvent events. If the application does not provide an event listener for DRMAuthenticateEvent objects, the DRM-encrypted object remains in a "waiting for credentials" state and the content is therefore not available.

4 Once the application obtains the user credentials, it passes the credentials with the `setDRMAuthenticationCredentials()` method to the NetStream object. This signals to the NetStream object that it should try authenticating the user at the next available opportunity. AIR then passes the credential to the FMRMS for authentication. If the user was authenticated, then the application proceeds to the next step.

If authentication failed, a new DRMAuthenticateEvent event is dispatch and the application returns to step 3. This process repeats indefinitely. The application should provide a mechanism to handle and limit the repeated authentication attempts. For example, the application could allow the user to cancel the attempt which can close the NetStream connection.

5 Once the user was authenticated, or if anonymous encryption was used, then the DRM subsystem retrieves the voucher. The voucher is used to check if the user is authorized to view the content. The information in the voucher can apply to both the authenticated and the anonymous users. For example, both the authenticated and anonymous users may have access to the content for a specified period of time before the content expires or they may not have access to the content because the content provider may not support the version of the viewing application.

If an error has not occurred and the user was authorized to view the content, DRMStatusEvent event object is dispatched and the AIR application begins playback. The DRMStatusEvent object holds the related voucher information, which identifies the user's policy and permissions. For example, it holds information regarding whether the content can be made available offline or when the voucher expires and the content can no longer be viewed. The application can use this data to inform the user of the status of their policy. For example, the application can display the number of remaining days the user has for viewing the content in a status bar.

If the user is allowed offline access, the voucher is cached and the encrypted content is downloaded to the user's machine and made accessible for the duration defined in the offline lease period. The "detail" property in the event contains *"DRM.voucherObtained"*. The application decides where to store the content locally in order for it to be available offline.

All DRM-related errors result in the application dispatching a DRMErrorEvent event object. AIR handles the DRM authentication failure by re-firing the DRMAuthenticationEvent event object. All other error events must be explicitly handled by the application. This includes cases where user inputs valid credentials, but the voucher protecting the encrypted content restricts the access to the content. For example, an authenticated user may still not have access to the content because the rights have not been paid for. This could also occur where two users, both registered members with the same media publisher, are attempting to share content that only one of the members has paid for. The application should inform the user of the error, such as the restrictions to the content, as well as provide an alternative, such as instructions in how to register and pay for the rights to view the content.

Changes to the NetStream class

The NetStream class provides a one-way streaming connection between Flash Player or an AIR application, and either Flash Media Server or the local file system. (The NetStream class also supports progressive download.) A NetStream object is a channel within a NetConnection object. As part of AIR, the NetStream class includes four new DRM-related events:

Event	Description
drmAuthenticate	Defined in the DRMAuthenticateEvent class, this event is dispatched when a NetStream object tries to play a digital rights management (DRM) encrypted content that requires a user credential for authentication before play back. The properties of this event include header, usernamePrompt, passwordPrompt, and urlPrompt properties that can be used in obtaining and setting the user's credentials. This event occurs repeatedly until the NetStream object receives valid user credentials.

Event	Description
drmError	Defined in the DRMErrorEvent class and dispatched when a NetStream object, trying to play a digital rights management (DRM) encrypted file, encounters a DRM-related error. For example, DRM error event object is dispatched when the user authorization fails. This may be because the user has not purchased the rights to view the content or because the content provider does not support the viewing application.
drmStatus	Defined in DRMStatusEvent class, is dispatched when the digital rights management (DRM) encrypted content begins playing (when the user is authenticated and authorized to play the content). The DRMStatusEvent object contains information related to the voucher, such as whether the content can be made available offline or when the voucher expires and the content can no longer be viewed.
status	Defined in events.StatusEvent and only dispatched when the application attempts to play content encrypted with digital rights management (DRM), by invoking the NetStream.play() method. The value of the status code property is "DRM.encryptedFLV".

The NetStream class includes the following DRM-specific methods:

Method	Description
resetDRMVouchers()	Deletes all the locally cached digital rights management (DRM) voucher data for the current content. The application must download the voucher again for the user to be able to access the encrypted content. For example, the following code removes vouchers for a NetStream object: `NetStream.resetDRMVouchers();`
setDRMAuthenticationCredentials()	Passes a set of authentication credentials, namely username, password and authentication type, to the NetStream object for authentication. Valid authentication types are `"drm"` and `"proxy"`. With `"drm"` authentication type, the credentials provided is authenticated against the FMRMS. With `"proxy"` authentication type, the credentials authenticates against the proxy server and must match those required by the proxy server. For example, the proxy option allows the application to authenticate against a proxy server if an enterprise requires such a step before the user can access the Internet. Unless anonymous authentication is used, after the proxy authentication, the user must still authenticate against the FMRMS in order to obtain the voucher and play the content. You can use `setDRMAuthenticationcredentials()` a second time, with "`drm`" option, to authenticate against the FMRMS.

In the following code, username ("administrator"), password ("password") and the "drm" authentication type are set for authenticating the user. The setDRMAuthenticationCredentials() method must provide credentials that match credentials known and accepted by the content provider (the same user credentials that provided permission to view the content). The code for playing the video and making sure that a successful connection to the video stream has been made is not included here.

```
var connection:NetConnection = new NetConnection();
connection.connect(null);

var videoStream:NetStream = new NetStream(connection);

videoStream.addEventListener(DRMAuthenticateEvent.DRM_AUTHENTICATE,
                            drmAuthenticateEventHandler)

private function drmAuthenticateEventHandler(event:DRMAuthenticateEvent):void
{
    videoStream.setDRMAuthenticationCredentials("administrator", "password", "drm");
}
```

Using the DRMStatusEvent class

A NetStream object dispatches a DRMStatusEvent object when the content protected using digital rights management (DRM) begins playing successfully (when the voucher is verified, and when the user is authenticated and authorized to view the content). The DRMStatusEvent is also dispatched for anonymous users if they are permitted access. The voucher is checked to verify whether anonymous user, who do not require authentication, are allowed access to play the content. Anonymous users maybe denied access for a variety of reasons. For example, an anonymous user may not have access to the content because it has expired.

The DRMStatusEvent object contains information related to the voucher, such as whether the content can be made available offline or when the voucher expires and the content can no longer be viewed. The application can use this data to convey the user's policy status and its permissions.

Contents

DRMStatusEvent properties

The DRMStatusEvent class includes the following properties:

Property	Description
detail	A string explaining the context of the status event. In DRM 1.0, the only valid value is DRM.voucherObtained.
isAnonymous	Indicates whether the content, protected with DRM encryption, is available without requiring a user to provide authentication credentials (true) or not (false). A false value means user must provide a username and password that matches the one known and expected by the content provider.
isAvailableOffline	Indicates whether the content, protected with DRM encryption, can be made available offline (true) or not (false). In order for digitally protected content to be available offline, its voucher must be cached to the user's local machine.
offlineLeasePeriod	The remaining number of days that content can be viewed offline.
policies	A custom object that may contain custom DRM properties.
voucherEndDate	The absolute date on which the voucher expires and the content is no longer viewable.

Creating a DRMStatusEvent handler

The following example creates an event handler that outputs the DRM content status information for the NetStream object that originated the event. Add this event handler to a NetStream object that points to DRM-encrypted content.

```
private function drmStatusEventHandler(event:DRMStatusEvent):void
{
    trace(event.toString());
}
```

Using the DRMAuthenticateEvent class

The DRMAuthenticateEvent object is dispatched when a NetStream object tries to play a digital rights management (DRM) encrypted content that requires a user credential for authentication before play back.

The DRMAuthenticateEvent handler is responsible for gathering the required credentials (user name, password, and type) and passing the values to the `NetStream.setDRMAuthenticationCredentials()` method for validation. Each AIR application must provide some mechanism for obtaining user credentials. For example, the application could provide a user with a simple user interface to enter the username and password values, and optionally the type value as well. The AIR application should also provide a mechanism for handling and limiting the repeated authentication attempts.

Contents

DRMAuthenticateEvent properties

The DRMAuthenticateEvent class includes the following properties:

Property	Description
authenticationType	Indicates whether the supplied credentials are for authenticating against the FMRMS ("drm") or a proxy server ("proxy"). For example, the "proxy" option allows the application to authenticate against a proxy server if an enterprise requires such a step before the user can access the Internet. Unless anonymous authentication is used, after the proxy authentication, the user still must authenticate against the FMRMS in order to obtain the voucher and play the content. You can use setDRMAuthenticationcredentials() a second time, with "drm" option, to authenticate against the FMRMS.
header	The encrypted content file header provided by the server. It contains information about the context of the encrypted content.
netstream	The NetStream object that initiated this event.
passwordPrompt	A prompt for a password credential, provided by the server. The string can include instruction for the type of password required.
urlPrompt	A prompt for a URL string, provided by the server. The string can provide the location where the username and password is sent.
usernamePrompt	A prompt for a user name credential, provided by the server. The string can include instruction for the type of user name required. For example, a content provider may require an e-mail address as the user name.

Creating a DRMAuthenticateEvent handler

The following example creates an event handler that passes a set of hard-coded authentication credentials to the NetStream object that originated the event. (The code for playing the video and making sure that a successful connection to the video stream has been made is not included here.)

```
var connection:NetConnection = new NetConnection();
connection.connect(null);

var videoStream:NetStream = new NetStream(connection);

videoStream.addEventListener(DRMAuthenticateEvent.DRM_AUTHENTICATE,
                            drmAuthenticateEventHandler)

private function drmAuthenticateEventHandler(event:DRMAuthenticateEvent):void
{
    videoStream.setDRMAuthenticationCredentials("administrator", "password", "drm");
}
```

Creating an interface for retrieving user credentials

In the case where DRM content requires user authentication, the AIR application usually needs to retrieve the user's authentication credentials via a user interface.

The following is a Flex example of a simple user interface for retrieving user credentials. It consists of a panel object containing two TextInput objects, one for each of the user name and password credentials. The panel also contains a button that launches the `credentials()` method.

```
<mx:Panel x="236.5" y="113" width="325" height="204" layout="absolute" title="Login">
    <mx:TextInput x="110" y="46" id="uName"/>
    <mx:TextInput x="110" y="76" id="pWord" displayAsPassword="true"/>
    <mx:Text x="35" y="48" text="Username:"/>
    <mx:Text x="35" y="78" text="Password:"/>
    <mx:Button x="120" y="115" label="Login" click="credentials()"/>
</mx:Panel>
```

The `credentials()` method is a user-defined method that passes the user name and password values gathered in the simple user interface to the `setDRMAuthenticationCredentials()` method. Once the values are passed, the `credentials()` method resets the values of the TextInput objects.

```
<mx:Script>
    <![CDATA[
        public function credentials():void
        {
            videoStream.setDRMAuthenticationCredentials(uName, pWord, "drm");
            uName.text = "";
            pWord.text = "";
        }
    ]]>
</mx:Script>
```

One way to implement this type of simple interface is to include the panel as part of a new state that originates from the base state when the DRMAuthenticateEvent object is thrown. The following example contains a VideoDisplay object with a source attribute that points to a DRM-protected FLV. In this case, the credentials() method is modified so that it also returns the application to the base state after passing the user credentials and resetting the TextInput object values.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    width="800"
    height="500"
    title="DRM FLV Player"
    creationComplete="initApp()" >

    <mx:states>
        <mx:State name="LOGIN">
            <mx:AddChild position="lastChild">
                    <mx:Panel x="236.5" y="113" width="325" height="204" layout="absolute"
                            title="Login">
                    <mx:TextInput x="110" y="46" id="uName"/>
                    <mx:TextInput x="110" y="76" id="pWord" displayAsPassword="true"/>
                    <mx:Text x="35" y="48" text="Username:"/>
                    <mx:Text x="35" y="78" text="Password:"/>
                    <mx:Button x="120" y="115" label="Login" click="credentials()"/>
                    <mx:Button x="193" y="115" label="Reset" click="uName.text='';
                            pWord.text='';"/>
                </mx:Panel>
            </mx:AddChild>
        </mx:State>
    </mx:states>
    <mx:Script>
        <![CDATA[
            import flash.events.DRMAuthenticateEvent;
            private function initApp():void
            {
                videoStream.addEventListener(DRMAuthenticateEvent.DRM_AUTHENTICATE,
                                        drmAuthenticateEventHandler);
            }

            public function credentials():void
            {
                videoStream.setDRMAuthenticationCredentials(uName, pWord, "drm");
                uName.text = "";
                pWord.text = "";
                currentState='';
            }

            private function drmAuthenticateEventHandler(event:DRMAuthenticateEvent):void
            {
                currentState='LOGIN';
            }
        ]]>
    </mx:Script>

    <mx:VideoDisplay id="video" x="50" y="25" width="700" height="350"
        autoPlay="true"
        bufferTime="10.0"
        source="http://www.example.com/flv/Video.flv" />
</mx:WindowedApplication>
```

Using the DRMErrorEvent class

AIR dispatches a DRMErrorEvent object when a NetStream object, trying to play a digital rights management (DRM) encrypted file, encounters a DRM related error. In the case of invalid user credentials, the DRMAuthenticateEvent object handles the error by repeatedly dispatching until the user enters valid credentials, or the AIR application denies further attempts. The application should listen to any other DRM error events to detect, identify, and handle the DRM-related errors.

If a user enters valid credentials, they still may not be allowed to view the encrypted content, depending on the terms of the DRM voucher. For example, if the user is attempting to view the content in an unauthorized application, that is, an application that is not validated by the publisher of the encrypted content. In this case, a DRMErrorEvent object is dispatched. The error events can also be fired if the content is corrupted or if the application's version does not match what is specified by the voucher. The application must provide appropriate mechanism for handling errors.

Contents

DRMErrorEvent properties

The DRMErrorEvent class includes the following property:

subErrorID	Indicates the minor error ID with more information about the underlying problem.

The following table lists the errors that the DRMErrorEvent object reports:

Major Error Code	Minor Error Code	Error Details	Description
1001	0		User authentication failed.
1002	0		Flash Media Rights Management Server (FMRMS) is not supporting Secure Sockets Layer (SSL).
1003	0		The content has expired and is no longer available for viewing.
1004	0		User authorization failure. This can occur, for example, if the user has not purchased the content and therefore does not have the rights to view it.
1005	0	`Server URL`	Cannot connect to the server.
1006	0		A client update is required, that is, Flash Media Rights Management Server (FMRMS) requires a new digital rights management (DRM) engine.
1007	0		Generic internal failure.
1008	*Detailed decrypting error code*		An incorrect license key.
1009	0		FLV content is corrupted.
1010	0	`publisherID:applicationID`	The ID of the viewing application does not match a valid ID supported by the content publisher.
1011	0		Application version does not match what is specified in the policy.
1012	0		Verification of the voucher associated with the encrypted content failed, indicating that the content may be corrupted.
1013	0		The voucher associated with the encrypted content could not be saved.
1014	0		Verification of the FLV header integrity failed, indicating that the content may be corrupted.

Major Error Code	Minor Error ID	Error Details	Description
3300	*Adobe Policy Server error code*		The application detected an invalid voucher associated with the content.
3301	0		User authentication failed.
3302	0		Secure Sockets Layer (SSL) is not supported by the Flash Media Rights Management Server (FMRMS).
3303	0		The content has expired and is no longer available for viewing.

Major Error Code	Minor Error ID	Error Details	Description
3304	0		User authorization failure. This can occur even if the user is authenticated, for example, if the user has not purchased the rights to view the content.
3305	0	`Server URL`	Cannot connect to the server.
3306	0		A client update is required, that is, Flash Media Rights Management Server (FMRMS) requires a new digital rights management client engine.
3307	0		Generic internal digital rights management failure.
3308	*Detailed decrypting error code*		An incorrect license key.
3309	0		Flash video content is corrupted.
3310	0	`publisherID:applicationID`	The ID of the viewing application does not match a valid ID supported by the content publisher. In other words, the viewing application is not supported by the content provider.
3311	0	*min=x:max=y*	Application version does not match what is specified in the voucher.
3312	0		Verification of the voucher associated with the encrypted content failed, indicating that the content may be corrupted.
3313	0		The voucher associated with the encrypted content could not be saved to Microsafe.
3314	0		Verification of the FLV header integrity failed, indicating that the content may be corrupted.
3315			Remote playback of the DRM protected content is not allowed.

Creating a DRMErrorEvent handler

The following example creates an event handler for the NetStream object that originated the event. It is called when the NetStream encounters an error while attempting to play the DRM-encrypted content. Normally, when an application encounters an error, it performs any number of clean-up tasks, informs the user of the error, and provides options for solving the problem.

```
private function drmErrorEventHandler(event:DRMErrorEvent):void
{
    trace(event.toString());
}
```

Part 9: Interacting with the operating system

Chapter 29: Application launching and exit options

This section discusses options and considerations for launching an installed Adobe® AIR™ application, as well as options and considerations for closing a running application.

Contents

Application invocation

An AIR application is invoked when the user (or the operating system):

- Launches the application from the desktop shell.
- Uses the application as a command on a command line shell.
- Opens a type of file for which the application is the default opening application.
- (Mac OS X) clicks the application icon in the dock taskbar (whether or not the application is currently running).
- Chooses to launch the application from the installer (either at the end of a new installation process, or after double-clicking the AIR file for an already installed application).
- Begins an update of an AIR application when the installed version has signaled that it is handling application updates itself (by including a `<customUpdateUI>true</customUpdateUI>` declaration in the application descriptor file).
- Visits a web page hosting a Flash badge or application that calls `com.adobe.air.AIR launchApplication()` method specifying the identifying information for the AIR application. (The application descriptor must also include a `<allowBrowserInvocation>true</allowBrowserInvocation>` declaration for browser invocation to succeed.) See "Launching an installed AIR application from the browser" on page 338.

Whenever an AIR application is invoked, AIR dispatches an InvokeEvent object of type `invoke` through the singleton NativeApplication object. To allow an application time to initialize itself and register an event listener, `invoke` events are queued instead of discarded. As soon as a listener is registered, all the queued events are delivered.

Note: *When an application is invoked using the browser invocation feature, the NativeApplication object only dispatches an* `invoke` *event if the application is not already running. See "Launching an installed AIR application from the browser" on page 338.*

To receive `invoke` events, call the `addEventListener()` method of the NativeApplication object (`NativeApplication.nativeApplication)`. When an event listener registers for an `invoke` event, it also receives all `invoke` events that occurred before the registration. Queued `invoke` events are dispatched one at a time on a short interval after the call to `addEventListener()` returns. If a new `invoke` event occurs during this process, it may be dispatched before one or more of the queued events. This event queuing allows you to handle any `invoke` events that have occurred before your initialization code executes. Keep in mind that if you add an event listener later in execution (after application initialization), it will still receive all `invoke` events that have occurred since the application started.

Only one instance of an AIR application is started. When an already running application is invoked again, AIR dispatches a new `invoke` event to the running instance. It is the responsibility of an AIR application to respond to an `invoke` event and take the appropriate action (such as opening a new document window).

An `InvokeEvent` object contains any arguments passed to the application, as well as the directory from which the application has been invoked. If the application was invoked because of a file-type association, then the full path to the file is included in the command line arguments. Likewise, if the application was invoked because of an application update, the full path to the update AIR file is provided.

Your application can handle `invoke` events by registering a listener with its NativeApplication object:

```
NativeApplication.nativeApplication.addEventListener(InvokeEvent.INVOKE, onInvokeEvent);
```

And defining an event listener:

```
var arguments:Array;
var currentDir:File;
public function onInvokeEvent(invocation:InvokeEvent):void {
    arguments = invocation.arguments;
    currentDir = invocation.currentDirectory;
}
```

Capturing command line arguments

The command line arguments associated with the invocation of an AIR application are delivered in the `invoke` event dispatched by the NativeApplication object. The `InvokeEvent.arguments` property contains an array of the arguments passed by the operating system when an AIR application is invoked. If the arguments contain relative file paths, you can typically resolve the paths using the `currentDirectory` property.

The arguments passed to an AIR program are treated as white-space delimited strings, unless enclosed in double quotes:

Arguments	Array
tick tock	{tick,tock}
tick "tick tock"	{tick,tick tock}
"tick" “tock”	{tick,tock}
\"tick\" \"tock\"	{"tick","tock"}

The `InvokeEvent.currentDirectory` property contains a File object representing the directory from which the application was launched.

When an application is invoked because a file of a type registered by the application is opened, the native path to the file is included in the command line arguments as a string. (Your application is responsible for opening or performing the intended operation on the file.) Likewise, when an application is programmed to update itself (rather than relying on the standard AIR update user interface), the native path to the AIR file is included when a user double-clicks an AIR file containing an application with a matching application ID.

You can access the file using the `resolve()` method of the `currentDirectory` File object:

```
if((invokeEvent.currentDirectory != null)&&(invokeEvent.arguments.length > 0)){
    dir = invokeEvent.currentDirectory;
    fileToOpen = dir.resolvePath(invokeEvent.arguments[0]);
}
```

You should also validate that an argument is indeed a path to a file.

Example: Invocation event log

The following example demonstrates how to register listeners for and handle the `invoke` event. The example logs all the invocation events received and displays the current directory and command line arguments.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
    invoke="onInvoke(event)" title="Invocation Event Log">
    <mx:Script>
    <![CDATA[
    import flash.events.InvokeEvent;
    import flash.desktop.NativeApplication;

    public function onInvoke(invokeEvent:InvokeEvent):void {
        var now:String = new Date().toTimeString();
        logEvent("Invoke event received: " + now);

        if (invokeEvent.currentDirectory != null){
            logEvent("Current directory=" + invokeEvent.currentDirectory.nativePath);
        } else {
            logEvent("--no directory information available--");
        }

        if (invokeEvent.arguments.length > 0){
            logEvent("Arguments: " + invokeEvent.arguments.toString());
        } else {
            logEvent("--no arguments--");
        }
    }

    public function logEvent(entry:String):void {
        log.text += entry + "\n";
        trace(entry);
    }
    ]]>
    </mx:Script>
    <mx:TextArea id="log" width="100%" height="100%" editable="false"
        valueCommit="log.verticalScrollPosition=log.textHeight;"/>
</mx:WindowedApplication>
```

Launching on login

An AIR application can be set to launch automatically when the current user logs in by setting `NativeApplication.nativeApplication.startAtLogin=true`. Once set, the application automatically starts whenever the user logs in. It continues to launch at start until the setting is changed to `false`, the user manually changes the setting through the operating system, or the application is uninstalled. Launching on login is a run-time setting.

***Note:** The application does not launch when the computer system starts. It launches when the user logs in. The setting only applies to the current user. Also, the application must be installed to successfully set the* `startAtLogin` *property to* `true`*. An error is thrown if the property is set when an application is not installed (such as when it is launched with ADL).*

Browser invocation

Using the browser invocation feature, a website can launch an installed AIR application to be launched from the browser. Browser invocation is only permitted if the application descriptor file sets `allowBrowserInvocation` to `true`:

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

For more information on the application descriptor file, see "Setting AIR application properties" on page 88.

When the application is invoked via the browser, the application's NativeApplication object dispatches a BrowserInvokeEvent object.

To receive BrowserInvokeEvent events, call the `addEventListener()` method of the NativeApplication object (`NativeApplication.nativeApplication`) in the AIR application. When an event listener registers for a BrowserInvokeEvent event, it also receives all BrowserInvokeEvent events that occurred before the registration. These events are dispatched after the call to `addEventListener()` returns, but not necessarily before other BrowserInvokeEvent events that might be received after registration. This allows you to handle BrowserInvokeEvent events that have occurred before your initialization code executes (such as when the application was initially invoked from the browser). Keep in mind that if you add an event listener later in execution (after application initialization) it still receives all BrowserInvokeEvent events that have occurred since the application started.

The BrowserInvokeEvent object includes the following properties:

Property	**Description**
arguments	An array of arguments (strings) to pass to the application.
isHTTPS	Whether the content in the browser uses the https URL scheme (`true`) or not (`false`).

Property	Description
isUserEvent	Whether the browser invocation resulted in a user event (such as a mouse click). In AIR 1.0, this is always set to `true`; AIR requires a user event to the browser invocation feature.
sandboxType	The sandbox type for the content in the browser. Valid values are defined the same as those that can be used in the `Security.sandboxType` property, and can be one of the following: • `Security.APPLICATION` — The content is in the application security sandbox. • `Security.LOCAL_TRUSTED` — The content is in the local-with-filesystem security sandbox. • `Security.LOCAL_WITH_FILE` — The content is in the local-with-filesystem security sandbox. • `Security.LOCAL_WITH_NETWORK` — The content is in the local-with-networking security sandbox. • `Security.REMOTE` — The content is in a remote (network) domain.
securityDomain	The security domain for the content in the browser, such as `"www.adobe.com"` or `"www.example.org"`. This property is only set for content in the remote security sandbox (for content from a network domain). It is not set for content in a local or application security sandbox.

If you use the browser invocation feature, be sure to consider security implications. When a website launches an AIR application, it can send data via the `arguments` property of the BrowserInvokeEvent object. Be careful using this data in any sensitive operations, such as file or code loading APIs. The level of risk depends on what the application is doing with the data. If you expect only a specific website to invoke the application, the application should check the `securityDomain` property of the BrowserInvokeEvent object. You can also require the website invoking the application to use HTTPs, which you can verify by checking the `isHTTPS` property of the BrowserInvokeEvent object.

The application should validate the data passed in. For example, if an application expects to be passed URLs to a specific domain, it should validate that the URLs really do point to that domain. This can prevent an attacker from tricking the application into sending it sensitive data.

No application should use BrowserInvokeEvent arguments that might point to local resources. For example, an application should not create File objects based on a path passed from the browser. If remote paths are expected to be passed from the browser, the application should ensure that the paths do not use the `file://` protocol instead of a remote protocol.

For details on invoking an application from the browser, see "Launching an installed AIR application from the browser" on page 338.

Application termination

The quickest way to terminate an application is to call `NativeApplication.nativeApplication.exit()` and this works fine when your application has no data to save or resources to clean up. Calling `exit()` closes all windows and then terminates the application. However, to allow windows or other components of your application to interrupt the termination process, perhaps to save vital data, dispatch the proper warning events before calling `exit()`.

Another consideration in gracefully shutting down an application is providing a single execution path, no matter how the shut-down process starts. The user (or operating system) can trigger application termination in the following ways:

- By closing the last application window when `NativeApplication.nativeApplication.autoExit` is `true`.
- By selecting the application exit command from the operating system; for example, when the user chooses the exit application command from the default menu. This only happens on Mac OS; Windows does not provide an application exit command through system chrome.
- By shutting down the computer.

When an exit command is mediated through the operating system by one of these routes, the NativeApplication dispatches an `exiting` event. If no listeners cancel the `exiting` event, any open windows are closed. Each window dispatches a `closing` and then a `close` event. If any of the windows cancel the `closing` event, the shut-down process stops.

If the order of window closure is an issue for your application, listen for the `exiting` event from the NativeApplication and close the windows in the proper order yourself. This might be the case, for example, if you have a document window with tool palettes. It might be inconvenient, or worse, if the system closed the palettes, but the user decided to cancel the exit command to save some data. On Windows, the only time you will get the `exiting` event is after closing the last window (when the `autoExit` property of the NativeApplication object is set to `true`).

To provide consistent behavior on all platforms, whether the exit sequence is initiated via operating system chrome, menu commands, or application logic, observe the following good practices for exiting the application:

1 Always dispatch an `exiting` event through the NativeApplication object before calling `exit()` in application code and check that another component of your application doesn't cancel the event.

```
public function applicationExit():void {
    var exitingEvent:Event = new Event(Event.EXITING, false, true);
    NativeApplication.nativeApplication.dispatchEvent(exitingEvent);
    if (!exitingEvent.isDefaultPrevented()) {
        NativeApplication.nativeApplication.exit();
    }
}
```

2 Listen for the application `exiting` event from the `NativeApplication.nativeApplication` object and, in the handler, close any windows (dispatching a `closing` event first). Perform any needed clean-up tasks, such as saving application data or deleting temporary files, after all windows have been closed. Only use synchronous methods during cleanup to ensure that they finish before the application quits.

If the order in which your windows are closed doesn't matter, then you can loop through the `NativeApplication.nativeApplication.openedWindows` array and close each window in turn. If order *does* matter, provide a means of closing the windows in the correct sequence.

```
private function onExiting(exitingEvent:Event):void {
    var winClosingEvent:Event;
    for each (var win:NativeWindow in NativeApplication.nativeApplication.openedWindows)
{
        winClosingEvent = new Event(Event.CLOSING,false,true);
        win.dispatchEvent(winClosingEvent);
        if (!winClosingEvent.isDefaultPrevented()) {
            win.close();
        } else {
            exitingEvent.preventDefault();
        }
    }

    if (!exitingEvent.isDefaultPrevented()) {
        //perform cleanup
    }
}
```

3 Windows should always handle their own clean up by listening for their own `closing` events.

4 Only use one `exiting` listener in your application since handlers called earlier cannot know whether subsequent handlers will cancel the `exiting` event (and it would be unwise to rely on the order of execution).

See also

- "Setting AIR application properties" on page 88
- "Presenting a custom application update user interface" on page 345

Chapter 30: Reading application settings

At runtime, you can get properties of the application descriptor file as well as the publisher ID for an application. These are set in the `applicationDescriptor` and `publisherID` properties of the NativeApplication object.

Contents

Reading the application descriptor file

You can read the application descriptor file of the currently running application, as an XML object, by getting the `applicationDescriptor` property of the NativeApplication object, as in the following:

```
var appXml:XML = NativeApplication.nativeApplication.applicationDescriptor;
```

You can then access the application descriptor data as an XML (E4X) object, as in the following:

```
var appXml:XML = NativeApplication.nativeApplication.applicationDescriptor;
var ns:Namespace = appXml.namespace();
var appId = appXml.ns::id[0];
var appVersion = appXml.ns::version[0];
var appName = appXml.ns::filename[0];
air.trace("appId:", appId);
air.trace("version:", appVersion);
air.trace("filename:", appName);
```

For more information, see "The application descriptor file structure" on page 88.

Getting the application and publisher identifiers

The application and publisher ids together uniquely identify an AIR application. You specify the application ID in the `<id>` element of the application descriptor. The publisher ID is derived from the certificate used to sign the AIR installation package.

The application ID can be read from the NativeApplication object's `id` property, as illustrated in the following code:

```
trace(NativeApplication.nativeApplication.applicationID);
```

The publisher ID can be read from the NativeApplication `publisherID` property:

```
trace(NativeApplication.nativeApplication.publisherID);
```

Note: *When an AIR application is run with ADL, it does not have a publisher ID unless one is temporarily assigned using the* `-pubID` *flag on the ADL command line.*

The publisher ID for an installed application can also be found in the `META-INF/AIR/publisherid` file within the application install directory.

See also

- “The application descriptor file structure” on page 88
- “About AIR publisher identifiers” on page 340

Chapter 31: Working with runtime and operating system information

This section discusses ways that an AIR application can manage operating system file associations, detect user activity, and get information about the Adobe® AIR™ runtime.

Contents

Managing file associations

Associations between your application and a file type must be declared in the application descriptor. During the installation process, the AIR application installer associates the AIR application as the default opening application for each of the declared file types, unless another application is already the default. The AIR application install process does not override an existing file type association. To take over the association from another application, call the `NativeApplication.setAsDefaultApplication()` method at run time.

It is a good practice to verify that the expected file associations are in place when your application starts up. This is because the AIR application installer does not override existing file associations, and because file associations on a user's system can change at any time. When another application has the current file association, it is also a polite practice to ask the user before taking over an existing association.

The following methods of the NativeApplication class let an application manage file associations. Each of the methods takes the file type extension as a parameter:

Method	Description
isSetAsDefaultApplication()	Returns true if the AIR application is currently associated with the specified file type.
setAsDefaultApplication()	Creates the association between the AIR application and the open action of the file type.
removeAsDefaultApplication()	Removes the association between the AIR application and the file type.
getDefaultApplication()	Reports the path of the application that is currently associated with the file type.

AIR can only manage associations for the file types originally declared in the application descriptor. You cannot get information about the associations of a non-declared file type, even if a user has manually created the association between that file type and your application. Calling any of the file association management methods with the extension for a file type not declared in the application descriptor causes the application to throw a runtime exception.

For information about declaring file types in the application descriptor, see "Declaring file type associations" on page 93.

Getting the runtime version and patch level

The NativeApplication object has a `runtimeVersion` property, which is the version of the runtime in which the application is running (a string, such as `"1.0.5"`). The NativeApplication object also has a `runtimePatchLevel` property, which is the patch level of the runtime (a number, such as 2960). The following code uses these properties:

```
trace(NativeApplication.nativeApplication.runtimeVersion);
trace(NativeApplication.nativeApplication.runtimePatchLevel);
```

Detecting AIR capabilities

For a file that is bundled with the Adobe AIR application, the `Security.sandboxType` property is set to the value defined by the `Security.APPLICATION` constant. You can load content (which may or may not contain APIs specific to AIR) based on whether a file is in the Adobe AIR security sandbox, as illustrated in the following code:

```
if (Security.sandboxType == Security.APPLICATION)
{
    // Load SWF that contains AIR APIs
}
else
{
    // Load SWF that does not contain AIR APIs
}
```

All resources that are not installed with the AIR application are assigned to the same security sandboxes as would be assigned by Adobe® Flash® Player in a web browser. Remote resources are put in sandboxes according to their source domains, and local resources are put in the local-with-networking, local-with-filesystem, or local-trusted sandbox.

You can check if the `Capabilities.playerType` static property is set to `"Desktop"` to see if content is executing in the runtime (and not running in Flash Player running in a browser).

For more information, see "AIR security" on page 69.

Tracking user presence

The NativeApplication object dispatches two events that help you detect when a user is actively using a computer. If no mouse or keyboard activity is detected in the interval determined by the `NativeApplication.idleThreshold` property, the NativeApplication dispatches a `userIdle` event. When the next keyboard or mouse input occurs, the NativeApplication object dispatches a `userPresent` event. The `idleThreshold` interval is measured in seconds and has a default value of 300 (5 minutes). You can also get the number of seconds since the last user input from the `NativeApplication.nativeApplication.lastUserInput` property.

The following lines of code set the idle threshold to 2 minutes and listen for both the `userIdle` and `userPresent` events:

```
NativeApplication.nativeApplication.idleThreshold = 120;
NativeApplication.nativeApplication.addEventListener(Event.USER_IDLE,
function(event:Event) {
    trace("Idle");
});
```

```
NativeApplication.nativeApplication.addEventListener(Event.USER_PRESENT,
function(event:Event) {
    trace("Present");
});
```

Note: *Only a single* `userIdle` *event is dispatched between any two* `userPresent` *events.*

Part 10: Networking and communications

Chapter 32: Monitoring network connectivity

Adobe® AIR™ provides the means to check for changes to the network connectivity of the computer on which an AIR application is installed. This information is useful if an application uses data obtained from the network. Also, an application can check the availability of a network service.

Contents

Detecting network connectivity changes

Your AIR application can run in environments with uncertain and changing network connectivity. To help an application manage connections to online resources, Adobe AIR sends a network change event whenever a network connection becomes available or unavailable. The application's NativeApplication object dispatches the network change event. To react to this event, add a listener:

```
NativeApplication.nativeApplication.addEventListener(Event.NETWORK_CHANGE,
onNetworkChange);
```

And define an event handler function:

```
function onNetworkChange(event:Event)
{
    //Check resource availability
}
```

The `Event.NETWORK_CHANGE` event does not indicate a change in all network activity, but only that a network connection has changed. AIR does not attempt to interpret the meaning of the network change. A networked computer may have many real and virtual connections, so losing a connection does not necessarily mean losing a resource. On the other hand, new connections do not guarantee improved resource availability, either. Sometimes a new connection can even block access to resources previously available (for example, when connecting to a VPN).

In general, the only way for an application to determine whether it can connect to a remote resource is to try it. To this end, the service monitoring frameworks in the air.net package provide AIR applications with an event-based means of responding to changes in network connectivity to a specified host.

***Note:** The service monitoring framework detects whether a server responds acceptably to a request. This does not guarantee full connectivity. Scalable web services often use caching and load-balancing appliances to redirect traffic to a cluster of web servers. In this situation, service providers only provide a partial diagnosis of network connectivity.*

Service monitoring basics

The service monitor framework, separate from the AIR framework, resides in the file servicemonitor.swc. In order to use the framework, the servicemonitor.swc file must be included in your build process. Adobe® Flex™ Builder™ 3 includes this automatically.

The ServiceMonitor class implements the framework for monitoring network services and provides a base functionality for service monitors. By default, an instance of the ServiceMonitor class dispatches events regarding network connectivity. The ServiceMonitor object dispatches these events when the instance is created and whenever a network change is detected by Adobe AIR. Additionally, you can set the `pollInterval` property of a ServiceMonitor instance to check connectivity at a specified interval in milliseconds, regardless of general network connectivity events. A ServiceMonitor object does not check network connectivity until the `start()` method is called.

The URLMonitor class, a subclass of the ServiceMonitor class, detects changes in HTTP connectivity for a specified URLRequest.

The SocketMonitor class, also a subclass of the ServiceMonitor class, detects changes in connectivity to a specified host at a specified port.

Detecting HTTP connectivity

The URLMonitor class determines if HTTP requests can be made to a specified address at port 80 (the typical port for HTTP communication). The following code uses an instance of the URLMonitor class to detect connectivity changes to the Adobe website:

```
import air.net.URLMonitor;
import flash.net.URLRequest;
import flash.events.StatusEvent;

var monitor:URLMonitor;
monitor = new URLMonitor(new URLRequest('http://www.adobe.com'));
monitor.addEventListener(StatusEvent.STATUS, announceStatus);
monitor.start();

function announceStatus(e:StatusEvent):void {
    trace("Status change. Current status: " + monitor.available);
}
```

Detecting socket connectivity

AIR applications can also use socket connections for push-model connectivity. Firewalls and network routers typically restrict network communication on unauthorized ports for security reasons. For this reason, developers must consider that users may not have the capability of making socket connections.

Similar to the URLMonitor example, the following code uses an instance of the SocketMonitor class to detect connectivity changes to a socket connection at 6667, a common port for IRC:

```
import air.net.ServiceMonitor;
import flash.events.StatusEvent;

socketMonitor = new SocketMonitor('www.adobe.com',6667);
socketMonitor.addEventListener(StatusEvent.STATUS, socketStatusChange);
socketMonitor.start();

function announceStatus(e:StatusEvent):void {
    trace("Status change. Current status: " + socketMonitor.available);
}
```

Chapter 33: URL requests and networking

The new Adobe AIR functionality related to specifying URL requests is not available to SWF content running in the browser. This functionality is only available to content in the application security sandbox. This section describes the URLRequest features in the runtime, and it discusses networking API changes AIR content.

For other information on using ActionScript™ 3.0 networking and communications capabilities, see *Programming ActionScript 3.0*, delivered with both Adobe® Flash® CS3 and Adobe® Flex™ Builder™ 3.

Contents

Using the URLRequest class

The URLRequest class lets you define more than simply the URL string. AIR adds some new properties to the URLRequest class, which are only available to AIR content running in the application security sandbox. Content in the runtime can define URLs using new URL schemes (in addition to standard schemes like `file` and `http`).

Contents

URLRequest properties

The URLRequest class includes the following properties which are available to content only in the AIR application security sandbox:

Property	Description
followRedirects	Specifies whether redirects are to be followed (`true`, the default value) or not (`false`). This is only supported in the runtime.
manageCookies	Specifies whether the HTTP protocol stack should manage cookies (`true`, the default value) or not (`false`) for this request. This is only supported in the runtime.
authenticate	Specifies whether authentication requests should be handled (`true`) for this request. This is only supported in the runtime. The default is to authenticate requests—this may cause an authentication dialog box to be displayed if the server requires credentials to be shown. You can also set the user name and password—see "Setting URLRequest defaults" on page 325.

Property	Description
cacheResponse	Specifies whether successful response data should be cached for this request. This is only supported in the runtime. The default is to cache the response (`true`).
useCache	Specifies whether the local cache should be consulted before this URLRequest fetches data. This is only supported in the runtime. The default (`true`) is to use the local cached version, if available.
userAgent	Specifies the user-agent string to be used in the HTTP request.

The following properties of a URLRequest object can be set by content in any sandbox (not just the AIR application security sandbox):

Property	Description
contentType	The MIME content type of any data sent with the URL request.
data	An object containing data to be transmitted with the URL request.
digest	A secure "digest" to a cached file to track Adobe® Flash® Player cache.
method	Controls the HTTP request method, such as a GET or POST operation. (Content running in the AIR application security domain can specify strings other than `"GET"` or `"POST"` as the `method` property. Any HTTP verb is allowed and `"GET"` is the default method. See "AIR security" on page 69.)
requestHeaders	The array of HTTP request headers to be appended to the HTTP request.
url	Specifies the URL to be requested.

***Note:** The HTMLLoader class has related properties for settings pertaining to content loaded by an HTMLLoader object. For details, see "About the HTMLLoader class" on page 258.*

Setting URLRequest defaults

The URLRequestDefaults class lets you define default settings for URLRequest objects. For example, the following code sets the default values for the `manageCookies` and `useCache` properties:

```
URLRequestDefaults.manageCookies = false;
URLRequestDefaults.useCache = false;
```

The URLRequestDefaults class includes a `setLoginCredentialsForHost()` method that lets you specify a default user name and password to use for a specific host. The host, which is defined in the hostname parameter of the method, can be a domain, such as `"www.example.com"`, or a domain and a port number, such as `"www.example.com:80"`. Note that `"example.com"`, `"www.example.com"`, and `"sales.example.com"` are each considered unique hosts.

These credentials are only used if the server requires them. If the user has already authenticated (for example, by using the authentication dialog box), then you cannot change the authenticated user by calling the `setLoginCredentialsForHost()` method.

For example, the following code sets the default user name and password to use at www.example.com:

```
URLRequestDefaults.setLoginCredentialsForHost("www.example.com", "Ada", "love1816$X");
```

Each property of URLRequestDefaults settings applies to only the application domain of the content setting the property. However, the `setLoginCredentialsForHost()` method applies to content in all application domains within an AIR application. This way, an application can log in to a host and have *all* content within the application be logged in with the specified credentials.

For more information, see the URLRequestDefaults class in the *Flex 3 Language Reference (http://www.adobe.com/go/learn_flex3_aslr).*

Using AIR URL schemes in URLs

The standard URL schemes, such as the following, are available when defining URLs in any security sandbox in AIR:

`http:` and `https:`

Use these as you would use them in a web browser.

`file:`

Use this to specify a path relative to the root of the file system. For example:

```
file:///c:/AIR Test/test.txt
```

You can also use the following schemes when defining a URL for content running in the application security sandbox:

`app:`

Use this to specify a path relative to the root directory of the installed application (the directory that contains the application descriptor file for the installed application). For example, the following path points to a resources subdirectory of the directory of the installed application:

```
app:/resources
```

When running in the ADL application, the application resource directory is set to the directory that contains the application descriptor file.

`app-storage:`

Use this to specify a path relative to the application store directory. For each installed application, AIR defines a unique application store directory for each user, which is a useful place to store data specific to that application. For example, the following path points to a prefs.xml file in a settings subdirectory of the application store directory:

```
app-storage:/settings/prefs.xml
```

The application storage directory location is based on the user name, the application ID, and the publisher ID:

- On Mac OS—In:

  `/Users/`*user name*`/Library/Preferences/`*applicationID*`.`*publisherID*`/Local Store/`

  For example:

  ```
  /Users/babbage/Library/Preferences/com.example.TestApp.02D88EEED35F84C264A183921344EEA3
  53A629FD.1/Local Store
  ```

- On Windows—In the documents and Settings directory, in:

  user name`/Application Data/`*applicationID*`.`*publisherID*`/Local Store/`

  For example:

  ```
  C:\Documents and Settings\babbage\Application
  Data\com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store
  ```

The URL (and `url` property) for a File object created with `File.applicationStorageDirectory` uses the `app-storage` URL scheme, as in the following:

```
var dir:File = File.applicationStorageDirectory;
dir = dir.resolvePath("preferences");
trace(dir.url); // app-storage:/preferences
```

Using URL schemes in AIR

You can use a URLRequest object that uses any of these URL schemes to define the URL request for a number of different objects, such as a FileStream or a Sound object. You can also use these schemes in HTML content running in AIR; for example, you can use them in the `src` attribute of an `img` tag.

However, you can only use these AIR-specific URL schemes (`app:` and `app-storage:`) in content in the application security sandbox. For more information, see "AIR security" on page 69.

Prohibited URL schemes

Some APIs allow you to launch content in a web browser. For security reasons, some URL schemes are prohibited when using these APIs in AIR. The list of prohibited schemes depends on the security sandbox of the code using the API. For details, see "Opening a URL in the default system web browser" on page 327.

Changes to the URLStream class

The URLStream class provides low-level access to downloading data from URLs. In the runtime, the URLStream class includes a new event: `httpResponseStatus`. Unlike the `httpStatus` event, the `httpResponseStatus` event is delivered before any response data. The `httpResponseStatus` event (defined in the HTTPStatusEvent class) includes a `responseURL` property, which is the URL that the response was returned from, and a `responseHeaders` property, which is an array of URLRequestHeader objects representing the response headers that the response returned.

Opening a URL in the default system web browser

You can use the `navigateToURL()` function to open a URL in the default system web browser. For the URLRequest object you pass as the `request` parameter of this function, only the `url` property is used.

When using the `navigateToURL()` function, URL schemes are permitted based on the security sandbox of the code calling the `navigateToURL()` function.

Some APIs allow you to launch content in a web browser. For security reasons, some URL schemes are prohibited when using these APIs in AIR. The list of prohibited schemes depends on the security sandbox of the code using the API. (For details on security sandboxes, see "AIR security" on page 69.)

Application sandbox

The following schemes are allowed. Use these as you would use them in a web browser.

- `http:`
- `https:`
- `file:`

- `mailto:` — AIR directs these requests to the registered system mail application
- `app:`
- `app-storage:`

All other URL schemes are prohibited.

Remote sandbox

The following schemes are allowed. Use these as you would use them in a web browser.

- `http:`
- `https:`
- `mailto:` — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Local-with-file sandbox

The following schemes are allowed. Use these as you would use them in a web browser.

- `file:`
- `mailto:` — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Local-with-networking sandbox

The following schemes are allowed. Use these as you would use them in a web browser.

- `http:`
- `https:`
- `mailto:` — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Local-trusted sandbox

The following schemes are allowed. Use these as you would use them in a web browser.

- `file:`
- `http:`
- `https:`
- `mailto:` — AIR directs these requests to the registered system mail application

All other URL schemes are prohibited.

Chapter 34: Inter-application communication

The LocalConnection class enables communications between Adobe® AIR™ applications, as well as among AIR applications and SWF content running in the browser.

The `connect()` method of the LocalConnection class uses a `connectionName` parameter to identify applications. In content running in the AIR application security sandbox (content installed with the AIR application), AIR uses the string `app#` followed by the application ID followed by a dot (.) character, followed by the publisher ID for the AIR application (defined in the application descriptor file) in place of the domain used by SWF content running in the browser. For example, a `connectionName` for an application with the application ID `com.example.air.MyApp`, the `connectionName` and the publisher ID B146A943FBD637B68C334022D304CEA226D129B4 resolves to `"app#com.example.air.MyApp:connectionName.B146A943FBD637B68C334022D304CEA226D129B4"`. (For more information, see "Defining the basic application information" on page 89 and "Getting the application and publisher identifiers" on page 315.)

Part 11: Distributing and updating applications

Chapter 35: Distributing, Installing, and Running AIR applications

AIR applications are distributed as a single AIR installation file, which contains the application code and all assets. You can distribute this file through any of the typical means, such as by download, by e-mail, or by physical media such as a CD-ROM. Users can install the application by double-clicking the AIR file. You can use the *seamless install* feature, which lets users install your AIR application (and Adobe® AIR™, if needed) by clicking a single link on a web page.

Before it can be distributed, an AIR installation file must be packaged and signed with a code-signing certificate and private key. Digitally signing the installation file provides assurance that your application has not been altered since it was signed. In addition, a trusted certificate authority, such as Verisign or Thawte, issued the digital certificate, your users can confirm your identity as the publisher and signer. The AIR file is signed when the application is packaged with the AIR Developer Tool (ADT).

For information about how to package an application into an AIR file using Adobe® Flex™ Builder™ 3, see "Packaging AIR applications with Flex Builder" on page 21.

For information about how to package an application into an AIR file using the Adobe® AIR™ SDK, see "Packaging an AIR installation file using the AIR Developer Tool (ADT)" on page 28.

Contents

Installing and running an AIR application from the desktop

You can simply send the AIR file to the recipient. For example, you can send the AIR file as an e-mail attachment or as a link in a web page.

Once the user downloads the AIR application, the user follows these instructions to install it:

1. Double-click the AIR file.

   The Adobe AIR must already be installed on the computer.

2. In the Installation window, leave the default settings selected, and then click Continue.

   In Windows, AIR automatically does the following:

   - Installs the application into the Program Files directory
   - Creates a desktop shortcut for application
   - Creates a Start Menu shortcut
   - Adds an entry for application in the Add / Remove Programs Control Panel

   In the Mac OS, by default the application is added to the Applications directory.

If the application is already installed, the installer gives the user the choice of opening the existing version of the application or updating to the version in the downloaded AIR file. The installer identifies the application using the application ID and publisher ID in the AIR file.

3 When the installation is complete, click Finish.

On Mac OS, to install an updated version of an application, the user needs adequate system privileges to install to the application directory. On Windows, a user needs administrative privileges.

An application can also install a new version via ActionScript or JavaScript. For more information, see "Updating AIR applications" on page 344.

Once the AIR application is installed, a user simply double-clicks the application icon to run it, just like any other desktop application.

- On Windows, double-click the application's icon (which is either installed on the desktop or in a folder) or select the application from the Start menu.
- On Mac OS, double-click the application in the folder in which it was installed. The default installation directory is the /Applications directory.

The AIR *seamless install* feature lets a user install an AIR application by clicking a link in a web page. The AIR *browser invocation* features lets a user run an installed AIR application by clicking a link in a web page. These features are described in the following section.

Installing and running AIR applications from a web page

The seamless install feature lets you embed a SWF file in a web page that lets the user install an AIR application from the browser. If the runtime is not installed, the seamless install feature installs the runtime. The seamless install feature lets users install the AIR application without saving the AIR file to their computer. Included in the Flex SDK is a badge.swf file, which lets you easily use the seamless install feature. For details, see "Using the badge.swf file to install an AIR application" on page 333.

Contents

About customizing the seamless install badge.swf

In addition to using the badge.swf file provided with the SDK, you can create your own SWF file for use in a browser page. Your custom SWF file can interact with the runtime in the following ways:

- It can install an AIR application. See "Installing an AIR application from the browser" on page 337.
- It can check to see if a specific AIR application is installed. See "Checking from a web page if an AIR application is installed" on page 336.
- It can check to see if the runtime is installed. See "Checking if the runtime is installed" on page 336.

- It can launch an installed AIR application on the user's system. See "Launching an installed AIR application from the browser" on page 338.

These capabilities are all provided by calling APIs in a SWF file hosted at adobe.com: air.swf. This section describes how to use and customize the badge.swf file and how to call the air.swf APIs from your own SWF file.

Additionally, a SWF file running in the browser can communicate with a running AIR application by using the LocalConnection class. For more information, see "Inter-application communication" on page 329.

Important: *The features described in this section (and the APIs in the air.swf file) require the end user to have Adobe® Flash® Player 9 update 3 installed in the web browser. You can write code to check the installed version of Flash Player and provide an alternate interface to the user if the required version of Flash Player is not installed. For instance, if an older version of Flash Player is installed, you could provide a link to the download version of the AIR file (instead of using the badge.swf file or the air.swf API to install an application).*

Using the badge.swf file to install an AIR application

Included in the Flex SDK is a badge.swf file which lets you easily use the seamless install feature. The badge.swf can install the runtime and an AIR application from a link in a web page. The badge.swf file and its source code are provided to you for distribution on your website.

The instructions in this section provide information on setting parameters of the badge.swf file provided by Adobe. We also provide the source code for the badge.swf file, which you can customize.

Embedding the badge.swf file in a web page

1 Locate the following files, provided in the samples/badge directory of the Flex SDK, and add them to your web server.

- badge.swf
- default_badge.html
- AC_RunActiveContent.js

2 Open the default_badge.html page in a text editor.

3 In the default_badge.html page, in the `AC_FL_RunContent()` JavaScript function, adjust the `FlashVars` parameter definitions for the following:

Parameter	**Description**
`appname`	The name of the application, displayed by the SWF file when the runtime is not installed.
`appurl`	(Required). The URL of the AIR file to be downloaded. You must use an absolute, not relative, URL.
`airversion`	(Required). For the 1.0 version of the runtime, set this to 1.0.
`imageurl`	The URL of the image (optional) to display in the badge.
`buttoncolor`	The color of the download button (specified as a hex value, such as `FFCC00`).
`messagecolor`	The color of the text message displayed below the button when the runtime is not installed (specified as a hex value, such as `FFCC00`).

4 The minimum size of the badge.swf file is 217 pixels wide by 180 pixels high. Adjust the values of the `width` and `height` parameters of the `AC_FL_RunContent()` function to suit your needs.

5 Rename the default_badge.html file and adjust its code (or include it in another HTML page) to suit your needs.

You can also edit and recompile the badge.swf file. For details, see "Modifying the badge.swf file" on page 334.

Installing the AIR application from a seamless install link in a web page

Once you have added the seamless install link to a page, the user can install the AIR application by clicking the link in the SWF file.

1. Navigate to the HTML page in a web browser that has Flash Player (version 9 update 3 or later) installed.
2. In the web page, click the link in the badge.swf file.
   - If you have installed the runtime, skip to the next step.
   - If you have not installed the runtime, a dialog box is displayed asking whether you would like to install it.

   Install the runtime (see "Adobe AIR installation" on page 2), and then proceed with the next step.
3. In the Installation window, leave the default settings selected, and then click Continue.

   On a Windows computer, AIR automatically does the following:
   - Installs the application into c:\Program Files\
   - Creates a desktop shortcut for application
   - Creates a Start Menu shortcut
   - Adds an entry for application in the Add/Remove Programs Control Panel

   On Mac OS, the installer adds the application to the Applications directory (for example, in the /Applications directory in Mac OS).
4. Select the options you want, and then click the Install button.
5. When the installation is complete, click Finish.

Modifying the badge.swf file

The Flex SDK provides the source files for the badge.swf file. These files are included in the src folder of the SDK:

Source files	Description
badge.fla	The source Flash CS3 file used to compile the badge.swf file. The badge.fla file compiles into a SWF 9 file (which can be loaded in Flash Player).
AIRBadge.as	An ActionScript 3.0 class that defines the base class used in the basdge.fla file.

You can use Flash CS3 to redesign the visual interface of the badge.fla file.

The `AIRBadge()` constructor function, defined in the AIRBadge class, loads the air.swf file hosted at http://airdownload.adobe.com/air/browserapi/air.swf. The air.swf file includes code for using the seamless install feature.

The `onInit()` method (in the AIRBadge class) is invoked when the air.swf file is loaded successfully:

```
private function onInit(e:Event):void {
    _air = e.target.content;
    switch (_air.getStatus()) {
        case "installed" :
            root.statusMessage.text = "";
            break;
        case "available" :
            if (_appName && _appName.length > 0) {
                root.statusMessage.htmlText = "<p align='center'><font color='#"
                        + _messageColor + "'>In order to run " + _appName +
                        ", this installer will also set up Adobe® AIR™.</font></p>";
            } else {
                root.statusMessage.htmlText = "<p align='center'><font color='#"
                        + _messageColor + "'>In order to run this application, "
                        + "this installer will also set up Adobe® AIR™.</font></p>";
            }
            break;
        case "unavailable" :
            root.statusMessage.htmlText = "<p align='center'><font color='#"
                        + _messageColor
                        + "'>Adobe® AIR™ is not available for your system.</font></p>";
            root.buttonBg_mc.enabled = false;
            break;
    }
}
```

The code sets the global `_air` variable to the main class of the loaded air.swf file. This class includes the following public methods, which the badge.swf file accesses to call seamless install functionality:

Method	Description
`getStatus()`	Determines whether the runtime is installed (or can be installed) on the computer. For details, see "Checking if the runtime is installed" on page 336.
`installApplication()`	Installs the specified application on the user's machine. For details, see "Installing an AIR application from the browser" on page 337. • `url`—A string defining the URL. You must use an absolute, not relative, URL path. • `runtimeVersion`—A string indicating the version of the runtime (such as `"1.0.M6")` required by the application to be installed. • `arguments`— Arguments to be passed to the application if it is launched upon installation. The application is launched upon installation if the `allowBrowserInvocation` element is set to `true` in the application descriptor file. (For more information on the application descriptor file, see "Setting AIR application properties" on page 88.) If the application is launched as the result of a seamless install from the browser (with the user choosing to launch upon installation), the application's NativeApplication object dispatches a BrowserInvokeEvent object only if arguments are passed. Consider the security implications of data that you pass to the application. For details, see "Launching an installed AIR application from the browser" on page 338.

The settings for `url` and `runtimeVersion` are passed into the SWF file via the FlashVars settings in the container HTML page.

If the application starts automatically upon installation, you can use LocalConnection communication to have the installed application contact the badge.swf file upon invocation. For details, see "Inter-application communication" on page 329

You may also call the `getApplicationVersion()` method of the air.swf file to check if an application is installed. You can call this method either before the application installation process or after the installation is started. For details, see "Checking from a web page if an AIR application is installed" on page 336.

Loading the air.swf file

You can create your own SWF file that uses the APIs in the air.swf file to interact with the runtime and AIR applications from a web page in a browser. The air.swf file is hosted at http://airdownload.adobe.com/air/browserapi/air.swf. To reference the air.swf APIs from your SWF file, load the air.swf file into the same application domain as your SWF file. The following code shows an example of loading the air.swf file into the application domain of the loading SWF file:

```
var airSWF:Object; // This is the reference to the main class of air.swf
var airSWFLoader:Loader = new Loader(); // Used to load the SWF
var loaderContext:LoaderContext = new LoaderContext();
                                // Used to set the application domain

loaderContext.applicationDomain = ApplicationDomain.currentDomain;
```

```
airSWFLoader.contentLoaderInfo.addEventListener(Event.INIT, onInit);
airSWFLoader.load(new URLRequest("http://airdownload.adobe.com/browserapi/air.swf"), 
                    loaderContext);

function onInit(e:Event):void 
{
    airSWF = e.target.content;
}
```

Once the air.swf file is loaded (when the Loader object's `contentLoaderInfo` object dispatches the `init` event), you can call any of the air.swf APIs. These APIs are described in these sections:

- "Checking if the runtime is installed" on page 336
- "Checking from a web page if an AIR application is installed" on page 336
- "Installing an AIR application from the browser" on page 337
- "Launching an installed AIR application from the browser" on page 338

***Note:** The badge.swf file, provided with the Flex SDK, automatically loads the air.swf file. See "Using the badge.swf file to install an AIR application" on page 333. The instructions in this section apply to creating your own SWF file that loads the air.swf file.*

Checking if the runtime is installed

A SWF file can check if the runtime is installed by calling the `getStatus()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see "Loading the air.swf file" on page 335.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `getStatus()` method as in the following:

```
var status:String = airSWF.getStatus();
```

The `getStatus()` method returns one of the following string values, based on the status of the runtime on the computer:

String value	Description
`"available"`	The runtime can be installed on this computer but currently it is not installed.
`"unavailable"`	The runtime cannot be installed on this computer.
`"installed"`	The runtime is installed on this computer.

The `getStatus()` method throws an error if the required version of Flash Player (version 9 upgrade 3) is not installed in the browser.

Checking from a web page if an AIR application is installed

A SWF file can check if an AIR application (with a matching application ID and publisher ID) is installed by calling the `getApplicationVersion()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see "Loading the air.swf file" on page 335.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `getApplicationVersion()` method as in the following:

```
var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EEED35F84C264A183921344EEA353A629FD.1";
airSWF.getApplicationVersion(appID, pubID, versionDetectCallback);

function versionDetectCallback(version:String):void
{
    if (version == null)
    {
        trace("Not installed.");
        // Take appropriate actions. For instance, present the user with
        // an option to install the application.
    }
    else
    {
        trace("Version", version, "installed.");
        // Take appropriate actions. For instance, enable the
        // user interface to launch the application.
    }
}
```

The `getApplicationVersion()` method has the following parameters:

Parameters	Description
`appID`	The application ID for the application. For details, see "Defining the basic application information" on page 89.
`pubID`	The publisher ID for the application. For details, see "About AIR publisher identifiers" on page 340.
`callback`	A callback function to serve as the handler function. The getApplicationVersion() method operates asynchronously, and upon detecting the installed version (or lack of an installed version), this callback method is invoked. The callback method definition must include one parameter, a string, which is set to the version string of the installed application. If the application is not installed, a null value is passed to the function, as illustrated in the previous code sample.

The `getApplicationVersion()` method throws an error if the required version of Flash Player (version 9 upgrade 3) is not installed in the browser.

Installing an AIR application from the browser

A SWF file can install an AIR application by calling the `installApplication()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see "Loading the air.swf file" on page 335.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `installApplication()` method, as in the following code:

```
var url:String = "http://www.example.com/myApplication.air";
var runtimeVersion:String = "1.0.M6";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.installApplication(url, runtimeVersion, arguments);
```

The `installApplication()` method installs the specified application on the user's machine. This method has the following parameters:

Parameter	Description
`url`	A string defining the URL of the AIR file to install. You must use an absolute, not relative, URL path.
`runtimeVersion`	A string indicating the version of the runtime (such as "1.0") required by the application to be installed.
`arguments`	An array of arguments to be passed to the application if it is launched upon installation. The application is launched upon installation if the `allowBrowserInvocation` element is set to `true` in the application descriptor file. (For more information on the application descriptor file, see "Setting AIR application properties" on page 88.) If the application is launched as the result of a seamless install from the browser (with the user choosing to launch upon installation), the application's NativeApplication object dispatches a BrowserInvokeEvent object only if arguments have been passed. For details, see "Launching an installed AIR application from the browser" on page 338.

The `installApplication()` method can only operate when called in the event handler for a user event, such as a mouse click.

The `installApplication()` method throws an error if the required version of Flash Player (version 9 upgrade 3) is not installed in the browser.

On Mac OS, to install an updated version of an application, the user must have adequate system privileges to install to the application directory (and administrative privileges if the application updates the runtime). On Windows, a user must have administrative privileges.

You may also call the `getApplicationVersion()` method of the air.swf file to check if an application is already installed. You can call this method either before the application installation process begins or after the installation is started. For details, see "Checking from a web page if an AIR application is installed" on page 336. Once the application is running, it can communicate with the SWF content in the browser by using the LocalConnection class. For details, see "Inter-application communication" on page 329.

Launching an installed AIR application from the browser

To use the browser invocation feature (allowing it to be launched from the browser), the application descriptor file of the target application must include the following setting:

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

For more information on the application descriptor file, see "Setting AIR application properties" on page 88.

A SWF file in the browser can launch an AIR application by calling the `launchApplication()` method in the air.swf file loaded from http://airdownload.adobe.com/air/browserapi/air.swf. For details, see "Loading the air.swf file" on page 335.

Once the air.swf file is loaded, the SWF file can call the air.swf file's `launchApplication()` method, as in the following code:

```
var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EEED35F84C264A183921344EEA353A629FD.1";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.launchApplication(appID, pubID, arguments);
```

The `launchApplication()` method is defined at the top level of the air.swf file (which is loaded in the application domain of the user interface SWF file). Calling this method causes AIR to launch the specified application (if it is installed and browser invocation is allowed, via the `allowBrowserInvocation` setting in the application descriptor file). The method has the following parameters:

Parameter	Description
`appID`	The application ID for the application to launch. For details, see "Defining the basic application information" on page 89.
`pubID`	The publisher ID for the application to launch. For details, see "About AIR publisher identifiers" on page 340.
`arguments`	An array of arguments to pass to the application. The NativeApplication object of the application dispatches a BrowserInvokeEvent event that has an arguments property set to this array.

The `launchApplication()` method can only operate when called in the event handler for a user event, such as a mouse click.

The `launchApplication()` method throws an error if the required version of Flash Player (version 9 upgrade 3) is not installed in the browser.

If the `allowBrowserInvocation` element is set to `false` in the application descriptor file, calling the `launchApplication()` method has no effect.

Before presenting the user interface to launch the application, you may want to call the `getApplicationVersion()` method in the air.swf file. For details, see "Checking from a web page if an AIR application is installed" on page 336.

When the application is invoked via the browser invocation feature, the application's NativeApplication object dispatches a BrowserInvokeEvent object. For details, see "Browser invocation" on page 311.

If you use the browser invocation feature, be sure to consider security implications, described in "Browser invocation" on page 311.

Once the application is running, it can communicate with the SWF content in the browser by using the LocalConnection class. For details, see "Inter-application communication" on page 329.

Digitally signing an AIR file

Digitally signing your AIR installation files with a certificate issued by a recognized certificate authority (CA) provides significant assurance to your users that the application they are installing has not been accidentally or maliciously altered and identifies you as the signer (publisher). AIR displays the publisher name during installation when the AIR application has been signed with a certificate that is trusted, or which *chains* to a certificate that is trusted on the installation computer. Otherwise the publisher name is displayed as "Unknown."

Important: *A malicious entity could forge an AIR file with your identity if it somehow obtains your signing keystore file or discovers your private key.*

Contents

Information about code-signing certificates

The security assurances, limitations, and legal obligations involving the use of code-signing certificates are outlined in the Certificate Practice Statements (CPS) and subscriber agreements published by the issuing certificate authority. For more information about the agreements for two of the largest certificate authorities, refer to:

Verisign CPS (http://www.verisign.com/repository/CPS/)

Verisign Subscriber's Agreement (https://www.verisign.com/repository/subscriber/SUBAGR.html)

Thawte CPS (http://www.thawte.com/cps/index.html)

Thawte Code Signing Developer's Agreement (http://www.thawte.com/ssl-digital-certificates/free-guides-whitepapers/pdf/develcertsign.pdf)

About AIR code signing

When an AIR file is signed, a digital signature is included in the installation file. The signature includes a digest of the package, which is used to verify that the AIR file has not been altered since it was signed, and it includes information about the signing certificate, which is used to verify the publisher identity.

AIR uses the public key infrastructure (PKI) supported through the operating system's certificate store to establish whether a certificate can be trusted. The computer on which an AIR application is installed must either directly trust the certificate used to sign the AIR application, or it must trust a chain of certificates linking the certificate to a trusted certificate authority in order for the publisher information to be verified.

If an AIR file is signed with a certificate that does not chain to one of the trusted root certificates (and normally this includes all self-signed certificates), then the publisher information cannot be verified. While AIR can determine that the AIR package has not been altered since it was signed, there is no way to know who actually created and signed the file.

***Note:** A user can choose to trust a self-signed certificate and then any AIR applications signed with the certificate displays the value of the common name field in the certificate as the publisher name. AIR does not provide any means for a user to designate a certificate as trusted. The certificate (not including the private key) must be provided to the user separately and the user must use one of the mechanisms provided by the operating system or an appropriate tool to import the certificate into the proper location in system certificate store.*

About AIR publisher identifiers

As part of the process of building an AIR file, the AIR Developer Tool (ADT) generates a publisher ID. This is an identifier that is unique to the certificate used to build the AIR file. If you reuse the same certificate for multiple AIR applications, they will have the same publisher ID. The publisher ID is used to identify the AIR application in Local-Connection communication (see "Inter-application communication" on page 329). You can identify the publisher ID of an installed application by reading the `NativeApplication.nativeApplication.publisherID` property.

The following fields are used to compute the publisher ID: Name, CommonName, Surname, GivenName, Initials, GenerationQualifier, DNQualifier, CountryName, localityName, StateOrProvinceName, OrganizationName, OrganizationalUnitName, Title, Email, SerialNumber, DomainComponent, Pseudonym, BusinessCategory, StreetAddress, PostalCode, PostalAddress, DateOfBirth, PlaceOfBirth, Gender, CountryOfCitizenship, CountryOf-Residence, and NameAtBirth. If you renew a certificate issued by a certificate authority, or regenerate a self-signed certificate, these fields must be the same for the publisher ID to remain the same. In addition, the root certificate of a CA issued certificate and the public key of a self-signed certificate must be the same.

About Certificate formats

The AIR signing tools accept any keystores accessible through the Java Cryptography Architecture (JCA). This includes file-based keystores such as PKCS12-format files (which typically use a .pfx or .p12 file extension), Java .keystore files, PKCS11 hardware keystores, and the system keystores. The keystore formats that ADT can access depend on the version and configuration of the Java runtime used to run ADT. Accessing some types of keystore, such as PKCS11 hardware tokens, may require the installation and configuration of additional software drivers and JCA plug-ins.

To sign AIR files, you can use an existing class-3, high assurance code signing certificate or you can obtain a new one. For example, any of the following types of certificate from Verisign or Thawte can be used:

- Verisign:
  - Microsoft Authenticode Digital ID
  - Sun Java Signing Digital ID
- Thawte:
  - AIR Developer Certificate
  - Apple Developer Certificate
  - JavaSoft Developer Certificate
  - Microsoft Authenticode Certificate

***Note:** The certificate must be marked for code signing. You typically cannot use an SSL certificate to sign AIR files.*

Timestamps

When you sign an AIR file, the packaging tool queries the server of a timestamp authority to obtain an independently verifiable date and time of signing. The timestamp obtained is embedded in the AIR file. As long as the signing certificate is valid at the time of signing, the AIR file can be installed, even after the certificate has expired. On the other hand, if no timestamp is obtained, the AIR file ceases to be installable when the certificate expires or is revoked.

By default, the AIR packaging tools obtain a timestamp. However, to allow applications to be packaged when the timestamp service is unavailable, you can turn timestamping off. Adobe recommends that all publically distributed AIR files include a timestamp.

The default timestamp authority used by the AIR packaging tools is Geotrust.

Obtaining a certificate

To obtain a certificate, you would normally visit the certificate authority web site and complete the company's procurement process. The tools used to produce the keystore file needed by the AIR tools depend on the type of certificate purchased, how the certificate is stored on the receiving computer, and, in some cases, the browser used to obtain the certificate. For example, to obtain and export a Microsoft Authenticode certificate, Verisign or Thawte require you to use Microsoft Internet Explorer. The certificate can then be exported as a .pfx file directly from the Internet Explorer user interface.

You can generate a self-signed certificate using the Air Development Tool (ADT) used to package AIR installation files. Some third-party tools can also be used.

For instructions on how to generate a self-signed certificate, as well as instructions on signing an AIR file, see "Packaging an AIR installation file using the AIR Developer Tool (ADT)" on page 28. You can also export and sign AIR files using Flex Builder, Dreamweaver, and the AIR update for Flash.

The following example describes how to obtain an AIR Developer Certificate from the Thawte Certificate Authority and prepare it for use with ADT. This example illustrates only one of the many ways to obtain and prepare a code signing certificate for use.

Example: Getting an AIR Developer Certificate from Thawte

To purchase an AIR Developer Certificate, the Thawte web site requires you to use the Mozilla Firefox browser. The private key for the certificate is stored within the browser's keystore. Ensure that the Firefox keystore is secured with a master password and that the computer itself is physically secure. (You can export and remove the certificate and private key from the browser keystore once the procurement process is complete.)

As part of the certificate enrollment process a private/public key pair is generated. The private key is automatically stored within the Firefox keystore. You must use the same computer and browser to both request and retrieve the certificate from Thawte's web site.

1 Visit the Thawte web site and navigate to the Product page for Code Signing Certificates.

2 From the list of Code Signing Certificates, select the Adobe AIR Developer Certificate.

3 Complete the three step enrollment process. You need to provide organizational and contact information. Thawte then performs its identity verification process and may request additional information. After verification is complete, Thawte will send you e-mail with instructions on how to retrieve the certificate.

***Note:** Note: Additional information about the type of documentation required can be found here: https://www.thawte.com/ssl-digital-certificates/free-guides-whitepapers/pdf/enroll_codesign_eng.pdf.*

4 Retrieve the issued certificate from the Thawte site. The certificate is automatically saved to the Firefox keystore.

5 Export a keystore file containing the private key and certificate from the Firefox keystore using the following steps:

Note: *When exporting the private key/cert from Firefox, it is exported in a .p12 (pfx) format which ADT, Flex, Flash, and Dreamweaver can use.*

a Open the Firefox *Certificate Manager* dialog:
On Windows: open Tools -> Options -> Advanced -> Certificates -> Manage Certificates
On Mac OS: open Firefox -> Preferences -> Advanced -> Your Certificates -> View Certificates

b Select the Adobe AIR Code Signing Certificate from the list of certificates and click the **Backup** button.

c Enter a file name and the location to which to export the keystore file and click **Save**.

d If you are using the Firefox master password, you are prompted to enter your password for the software security device in order to export the file. (This password is used only by Firefox.)

e On the *Choose a Certificate Backup Password* dialog box, create a password for the keystore file.

Important: *This password protects the keystore file and is required when the file is used for signing AIR applications.A secure password should be chosen.*

f Click OK. You should receive a successful backup password message. The keystore file containing the private key and certificate is saved with a .p12 file extension (in PKCS12 format)

6 Use the exported keystore file with ADT, Flex Builder, Flash, or Dreamweaver. The password created for the file is required whenever an AIR application is signed.

Important: *The private key and certificate are still stored within the Firefox keystore. While this permits you to export an additional copy of the certificate file, it also provides another point of access that must be protected to maintain the security of your certificate and private key.*

Terminology

This section provides a glossary of some of the key terminology you should understand when making decisions about how to sign your application for public distribution.

Term	**Description**
Certificate Authority (CA)	An entity in a public-key infrastructure network that serves as a trusted third party and ultimately certifies the identity of the owner of a public key. A CA normally issues digital certificates, signed by its own private key, to attest that it has verified the identity of the certificate holder.
Certificate Practice Statement (CPS)	Sets forth the practices and policies of the certificate authority in issuing and verifying certificates. The CPS is part of the contract between the CA and its subscribers and relying parties. It also outlines the policies for identity verification and the level of assurances offered by the certificates they provide.
Certificate Revocation List (CRL)	A list of issued certificates that have been revoked and should no longer be relied upon. AIR checks the CRL at the time an AIR application is signed, and, if no timestamp is present, again when the application is installed.
Certificate chain	A certificate chain is a sequence of certificates in which each certificate in the chain has been signed by the next certificate.
Digital Certificate	A digital document that contains information about the identity of the owner, the owner's public key, and the identity of the certificate itself. A certificate issued by a certificate authority is itself signed by a certificate belonging to the issuing CA.

Term	Description
Digital Signature	An encrypted message or digest that can only be decrypted with the public key half of a public-private key pair. In a PKI, a digital signature contains one or more digital certificates that are ultimately traceable to the certificate authority. A digital signature can be used to validate that a message (or computer file) has not been altered since it was signed (within the limits of assurance provided by the cryptographic algorithm used), and, assuming one trusts the issuing certificate authority, the identity of the signer.
Keystore	A database containing digital certificates and, in some cases, the related private keys.
Java Cryptography Architecture (JCA)	An extensible architecture for managing and accessing keystores. See the Java Cryptography Architecture Reference Guide for more information.
PKCS #11	The Cryptographic Token Interface Standard by RSA Laboratories. A hardware token based keystore.
PKCS #12	The Personal Information Exchange Syntax Standard by RSA Laboratories. A file-based keystore typically containing a private key and its associated digital certificate.
Private Key	The private half of a two-part, public-private key asymmetric cryptography system. The private key must be kept secret and should never be transmitted over a network. Digitally signed messages are encrypted with the private key by the signer.
Public Key	The public half of a two-part, public-private key asymmetric cryptography system. The public key is openly available and is used to decrypt messages encrypted with the private key.
Public Key Infrastructure (PKI)	A system of trust in which certificate authorities attest to the identity of the owners of public keys. Clients of the network rely on the digital certificates issued by a trusted CA to verify the identity of the signer of a digital message (or file).
Time stamp	A digitally signed datum containing the date and time an event occurred. ADT can include a time stamp from an RFC 3161 compliant time server in an AIR package. When present, AIR uses the time stamp to establish the validity of a certificate at the time of signing. This allows an AIR application to be installed after its signing certificate has expired.
Time stamp authority	An authority that issues time stamps. To be recognized by AIR, the time stamp must conform to RFC 3161 and the time stamp signature must chain to a trusted root certificate on the installation machine.

Chapter 36: Updating AIR applications

Users can install or update an AIR application by double-clicking an AIR file on their computer or from the browser (using the seamless install feature), and the Adobe® AIR™ installer application manages the installation, alerting the user if they are updating an already existing application. (See "Distributing, Installing, and Running AIR applications" on page 331.)

However, you can also have an installed application update itself to a new version, using the Updater class. (An installed application may detect that a new version is available to be downloaded and installed.) The Updater class includes an `update()` method that lets you point to an AIR file on the user's computer and update to that version.

Contents

About updating applications

The Updater class (in the flash.desktop package) includes one method, `update()`, which you can use to update the currently running application with a different version. For example, if the user has a version of the AIR file ("Sample_App_v2.air") located on the desktop, the following code updates the application:

```
var updater:Updater = new Updater();
var airFile:File = File.desktopDirectory.resolvePath("Sample_App_v2.air");
var version:String = "2.01";
updater.update(airFile, version);
```

Prior to using the Updater class, the user or the application must download the updated version of the AIR file to the computer. For more information, see "Downloading an AIR file to the user's computer" on page 346.

Results of the method call

When an application in the runtime calls the `update()` method, the runtime closes the application, and it then attempts to install the new version from the AIR file. The runtime checks that the application ID and publisher ID specified in the AIR file matches the application ID and publisher ID for the application calling the `update()` method. (For information on the application ID and publisher ID, see "Setting AIR application properties" on page 88.) It also checks that the version string matches the `version` string passed to the `update()` method. If installation completes successfully, the runtime opens the new version of the application. Otherwise (if the installation cannot complete), it reopens the existing (pre-install) version of the application.

On Mac OS, to install an updated version of an application, the user must have adequate system privileges to install to the application directory. On Windows, a user must have administrative privileges.

If the updated version of the application requires an updated version of the runtime, the new runtime version is installed. To update the runtime, a user must have administrative privileges for the computer.

When testing an application using ADL, calling the `update()` method results in a runtime exception.

About the version string

The string that is specified as the `version` parameter of the `update()` method must match the string in the `version` attribute of the main `application` element of the application descriptor file for the AIR file to be installed. Specifying the `version` parameter is required for security reasons. By requiring the application to verify the version number in the AIR file, the application will not inadvertently install an older version, which might contain a security vulnerability that has been fixed in the currently installed application. The application should also check the version string in the AIR file with version string in the installed application to prevent downgrade attacks.

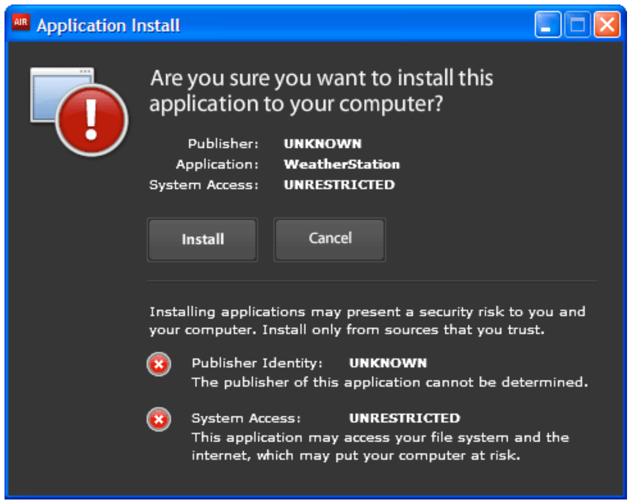
The version string can be of any format. For instance, it can be `"2.01"` or `"version 2"`. The format of this string is left for you, the application developer, to decide. The runtime does not validate the version string; the application code should do this before updating the application.

If an Adobe AIR application downloads an AIR file via the web, it is a good practice to have a mechanism by which the web service can notify the Adobe AIR application of the version being downloaded. The application can then use this string as the `version` parameter of the `update()` method. If the AIR file is obtained by some other means, in which the version of the AIR file is unknown, the AIR application can examine the AIR file to determine the version information. (An AIR file is a ZIP-compressed archive, and the application descriptor file is the second record in the archive.)

For details on the application descriptor file, see "Setting AIR application properties" on page 88.

Presenting a custom application update user interface

AIR includes a default update interface:



Are you sure you want to install this application to your computer?

Publisher: UNKNOWN
Application: WeatherStation
System Access: UNRESTRICTED

Installing applications may present a security risk to you and your computer. Install only from sources that you trust.

Publisher Identity: UNKNOWN
The publisher of this application cannot be determined.

System Access: UNRESTRICTED
This application may access your file system and the internet, which may put your computer at risk.

This interface is always used the first time a user installs a version of an application on a machine. However, you can define your own interface to use for subsequent instances. To do this, specify a `customUpdateUI` element in the application descriptor file for the currently installed application:

```
<customUpdateUI>true</customUpdateUI>
```

When the application is installed and the user opens an AIR file with an application ID and a publisher ID that match the installed application, the runtime opens the application, rather than the default AIR application installer. For more information, see "Providing a custom user interface for application updates" on page 93.

The application can decide, when it is invoked (when the `NativeApplication.nativeApplication` object dispatches an `invoke` event), whether to update the application (using the Updater class). If it decides to update, it can present its own installation interface (which differs from its standard running interface) to the user.

Downloading an AIR file to the user's computer

To use the Updater class, the user or the application must first save an AIR file locally to the user's computer. For example, the following code reads an AIR file from a URL (http://example.com/air/updates/Sample_App_v2.air) and saves the AIR file to the application storage directory:

```
var urlString:String = "http://example.com/air/updates/Sample_App_v2.air";
var urlReq:URLRequest = new URLRequest(urlString);
var urlStream:URLStream = new URLStream();
var fileData:ByteArray = new ByteArray();
urlStream.addEventListener(Event.COMPLETE, loaded);
urlStream.load(urlReq);

function loaded(event:Event):void {
    urlStream.readBytes(fileData, 0, urlStream.bytesAvailable);
    writeAirFile();
}

function writeAirFile():void {
    var file:File = File.applicationStorageDirectory.resolvePath("My App v2.air");
    var fileStream:FileStream = new FileStream();
    fileStream.open(file, FileMode.WRITE);
    fileStream.writeBytes(fileData, 0, fileData.length);
    fileStream.close();
    trace("The AIR file is written.");
}
```

For more information, see "Workflow for reading and writing files" on page 161.

Checking to see if an application is running for the first time

Once you have updated an application you may want to provide the user with a "getting started" or "welcome" message. Upon launching, the application checks to see if it is running for the first time, so that it can determine whether to display the message.

One way to do this is to save a file to the application store directory upon initializing the application. Every time the application starts up, it should check for the existence of that file. If the file does not exist, then the application is running for the first time for the current user. If the file exists, the application has already run at least once. If the file exists and contains a version number older than the current version number, then you know the user is running the new version for the first time.

Here is a Flex example:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    title="Sample Version Checker Application"
    applicationComplete="init()">
    <mx:Script>
        <![CDATA[
            import flash.filesystem.*;
            public var file:File;
            public var currentVersion:String = "1.2";
            public function init():void {
                file = File.applicationStorageDirectory;
                file = file.resolvePath("Preferences/version.txt");
                trace(file.nativePath);
                if(file.exists) {
                    checkVersion();
                } else {
                    firstRun();
                }
            }
            private function checkVersion():void {
                var stream:FileStream = new FileStream();
                stream.open(file, FileMode.READ);
                var prevVersion:String = stream.readUTFBytes(stream.bytesAvailable);
                stream.close();
                if (prevVersion != currentVersion) {
                    log.text = "You have updated to version " + currentVersion + ".\n";
                } else {
                    saveFile();
                }
                log.text += "Welcome to the application.";
            }
            private function firstRun():void {
                log.text = "Thank you for installing the application. \n"
                    + "This is the first time you have run it.";
                saveFile();
            }
            private function saveFile():void {
                var stream:FileStream = new FileStream();
                stream.open(file, FileMode.WRITE);
                stream.writeUTFBytes(currentVersion);
                stream.close();
            }
        ]]>
    </mx:Script>
    <mx:TextArea id="log" width="100%" height="100%" />
</mx:WindowedApplication>
```

If your application saves data locally (such as, in the application storage directory), you may want to check for any previously saved data (from previous versions) upon first run.

Index