Adobe® AIR™ for JavaScript Developers

Pocket Guide

Mike Chambers, Daniel Dura, Dragos Georgita, and Kevin Hoyt

Adobe® AIR™ for JavaScript Developers Pocket Guide
by Mike Chambers, Daniel Dura, Dragos Georgita, and Kevin Hoyt

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Steve Weiss
Copy Editor: Audrey Doyle
Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Illustrator: Robert Romano

Printing History:
April 2008: First Edition

ISBN: 978-0-596-51837-0

[T]

1207246041

Adobe Developer Library

Adobe Developer Library, a copublishing partnership between O'Reilly Media Inc., and Adobe Systems, Inc., is the authoritative resource for developers using Adobe technologies. These comprehensive resources offer learning solutions to help developers create cutting-edge interactive web applications that can reach virtually anyone on any platform.

With top-quality books and innovative online resources covering the latest tools for rich-Internet application development, the *Adobe Developer Library* delivers expert training straight from the source. Topics include ActionScript, Adobe Flex®, Adobe Flash®, and Adobe Acrobat®.

Get the latest news about books, online resources, and more at *http://adobedeveloperlibrary.com*.

Contents

Preface

This book provides a quick introduction to developing applications for Adobe AIR. Adobe AIR is a new cross-platform desktop application runtime created by Adobe. Although Adobe AIR allows both Flash- and HTML-based application development, this book focuses on building applications using HTML and JavaScript.

The book gives an overview of Adobe AIR, shows how to set up your development environment, and discusses new Adobe AIR functionality and APIs. Once you have finished reading this book, you should have a good understanding of what Adobe AIR is as well as how to build HTML and JavaScript applications for it.

Adobe AIR Runtime Naming Conventions

Adobe AIR allows developers to leverage a number of web technologies to deploy web applications to the desktop. Indeed, there are so many technologies that it can be difficult to keep track of them all. Table 1 lists the terms used in the book, and defines each one.

Table 1. AIR runtime naming conventions

Name	Meaning
Adobe AIR	The cross-platform desktop runtime that enables the running of Adobe AIR applications.

Name	Meaning
Adobe AIR application	An application built with Flash, HTML, and/or PDF that runs on top of Adobe AIR.
Adobe Flash	Any content contained within a SWF 9 file format that runs in the Adobe Flash Player or Adobe AIR.
ActionScript	The ECMAScript-based programming language used to program Flash content. Unless otherwise noted, all references to ActionScript in this book refer to ActionScript 3.
HTML	Standard web-based markup language used to create and lay out web pages.
JavaScript	Web-based implementation of ECMAScript used to program content within HTML applications.
PDF	Short for Portable Document Format, a technology that allows for seamless distribution and display of electronic documents.
Adobe Flex Framework	An XML- and ActionScript-based framework designed to make developing Flash-based Rich Internet Applications (RIAs) easy.
Adobe Flex Builder	An Eclipse-based IDE used to build Flash-based RIAs using Flex and ActionScript.

What This Book Covers

This book gives a general overview of Adobe AIR, shows how to set up your development environment to start building applications, provides an overview of the HTML and JavaScript engines within the runtime, and shows how to perform a number of common programming tasks within Adobe AIR.

The following is a partial list of features and functionality included in the Adobe AIR 1.0 release:

- Mac support (OS X 10.4.9 and later; Intel and PPC)
- Windows support (Windows Vista, Windows XP SP2, and Windows 2000 SP4)
- File I/O API
- SQLite embedded database

- All functionality within Flash Player 9, including complete network stack
- Windowing APIs
- Command-line tools (ADL and ADT)
- HTML support within Flash-based content
- Top-level HTML applications
- Flash content within HTML applications
- ActionScript/JavaScript script bridging
- Flex Builder and Flex Framework support for authoring Adobe AIR applications
- Application command-line arguments
- Drag-and-drop support
- Rich clipboard access
- Native menu API
- Full-screen support
- Application update API
- Online/offline detection API
- Encrypted local data stores
- Presence APIs
- File type associations
- Application icons
- PDF support
- Right-click and contextual menu control
- System notifications

We will cover these features in more detail throughout the rest of the book.

Errors and Errata

This book is written against the 1.0 release of Adobe AIR, and was finalized after the runtime was released. Thus, all information in the book should be correct for Adobe AIR 1.0.

However, it is possible that there will be updates to the runtime, or that there are errors within the book. If something in the book does not seem correct, check the online documentation for the latest information.

You can find the latest information and documentation on Adobe AIR at:

http://www.adobe.com/go/air

You should also check the book's errata web site for the latest updates and corrections:

http://www.adobe.com/go/airjavascriptpocketguide

Audience for This Book

We hope this book is for you, but just to be sure, let's discuss some of the assumptions that we made, as well as what types of developers the book targets.

Who This Book Is For

This book is for developers interested in leveraging HTML and JavaScript to build and deploy applications to the desktop via Adobe AIR. If you don't have any experience with developing with HTML and JavaScript, we suggest that you spend some time getting up to speed on these technologies.

What Does This Book Assume?

The book assumes that the reader has at least a basic familiarity with creating HTML-based web applications and content using HTML and JavaScript.

You should be familiar with web technologies such as HTML, JavaScript, Ajax, and CSS, as well as general web development concepts.

Who This Book Is Not For

Although it is possible to create Flash- and Flex-based applications with Adobe AIR, this book does not go into any detail on Flash- and Flex-focused AIR application development. If you are a Flash or Flex developer interested in building AIR applications, this book can provide a good introduction and overview of AIR and its functionality, but you should view the Adobe AIR documentation and articles available from the Adobe AIR web site for a more Flash/Flex-focused discussion.

How This Book Is Organized

This book contains the following chapters, as well as one appendix:

Chapter 1, Introduction to Adobe AIR
: Provides a general overview of what Adobe AIR is and the types of applications it targets

Chapter 2, Getting Started with Adobe AIR Development
: Covers tips on starting your Adobe AIR development, and the steps for creating your first Adobe AIR application from the command line

Chapter 3, Working with JavaScript and HTML Within Adobe AIR
: Gives an overview of the HTML and JavaScript runtime environments within Adobe AIR, and provides an introduction to using JavaScript to access Adobe AIR functionality and APIs

Chapter 4, Adobe AIR Mini-Cookbook
: Provides tips and tricks for accomplishing common tasks within Adobe AIR applications, presented in the O'Reilly Cookbook format

Appendix A
: Lists JavaScript aliases to Adobe AIR APIs

How to Use This Book

You can use this book as an introduction to and overview of Adobe AIR, and as a step-by-step guide to getting started with Adobe AIR application development. Although it may be tempting to jump ahead to specific sections, it is strongly suggested that you at least read the first two chapters, which provide an overview of the runtime and discuss how to set up your development environment for building Adobe AIR applications. This will make it much easier to then jump into the specific areas of runtime functionality that may interest you.

Once you have read the book and understand the basics of how to build an Adobe AIR application with HTML and JavaScript, then you can use the book as a reference, referring to specific sections when you need to know how to tackle a specific problem. In particular, the Cookbook sections should prove useful as you develop your applications.

Finally, this book is just an introduction to Adobe AIR and does not cover all of the features and functionality included within it. It is meant to complement, but not replace, the extensive and in-depth documentation on the runtime provided by Adobe. Make sure to explore the documentation to ensure that you're familiar with all of the APIs and functionality not covered in this book.

Conventions Used in This Book

The following typographical conventions are used in this book:

Plain text
: Indicates menu titles, menu options, menu buttons, and keyboard accelerators (such as Alt and Ctrl).

Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities

`Constant width`
: Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, and the output from commands

`Constant width bold`
: Shows commands or other text that should be typed literally by the user

`Constant width italic`
: Shows text that should be replaced with user-supplied values

License and Code Examples

Support and More Information

Accessing the Book Online

You can always find the latest information about this book, as well as download a free electronic version of it, from the book's web site at:

http://www.adobe.com/go/airjavascriptpocketguide

Online Adobe AIR Resources

Although Adobe AIR is a new technology, there are already a number of resources where you can find more information on Adobe AIR and RIA development.

Official AIR site
: This is the primary web site for information, downloads, and documentation on AIR:

  http://www.adobe.com/go/air

Adobe AIR Developer FAQ
: This is the official Adobe AIR FAQ, answering common questions about AIR development:

  http://www.adobe.com/go/airfaq

Adobe AIR Developer Center
: This Developer Center provides articles, information, and resources on developing applications for Adobe AIR:

  http://www.adobe.com/go/airdevcenter

Developing Adobe AIR applications with JavaScript
: Here's where you'll find the Adobe AIR JavaScript documentation and API reference:

  http://www.adobe.com/go/learn_air_html_jslr

Adobe AIR documentation
: Visit this web site for complete Adobe AIR documentation:

  http://www.adobe.com/go/airdocs

Adobe AIR Developer Center for HTML and Ajax
: This Developer Center provides articles, tutorials, and whitepapers on using HTML and JavaScript to develop applications for Adobe AIR:

  http://www.adobe.com/go/airajaxdevcenter

Adobe AIR Forum
: This is the official Adobe forum for discussing the development of applications for Adobe AIR:

  http://www.adobe.com/go/airforums

Adobe AIR coders mailing list

This is a mailing list for discussing Adobe AIR application development:

http://www.adobe.com/go/airlist

Mike Chambers' weblog

One of the authors of this book, Mike Chambers is a member of the Adobe AIR team who posts frequently on Adobe AIR:

http://www.adobe.com/go/mikechambers

MXNA Adobe AIR Smart Category

The Adobe AIR Smart Category lists any discussions about Adobe AIR within the Adobe online development community:

http://www.adobe.com/go/airmxna

Ajaxian.com (http://ajaxian.com)

This is an Ajax news site with information, tips, tricks, and the latest news on developing with JavaScript and Ajax techniques:

http://www.ajaxian.com

Adobe Flex Developer Center

This Developer Center provides articles, information, and resources on working with the Flex Framework:

http://www.adobe.com/go/flexdevcenter

Flex coders mailing list

This is a popular mailing list for discussing development using the Flex Framework:

http://tech.groups.yahoo.com/group/flexcoders/

Universal Desktop Weblog

This is Ryan Stewart's weblog, which focuses on the latest developments in the world of RIAs:

http://blogs.zdnet.com/Stewart/

About the Authors

Mike Chambers

Mike Chambers has spent the past eight years building applications that target the Flash runtime. During that time, he has worked with numerous technologies, including Flash, Generator, .NET, Central, Flex, and Ajax. He is currently the principal product manager for developer relations for the Flash platform. He has written and spoken extensively on Flash and RIA development and is coauthor of *Adobe Apollo for Flex Developers Pocket Guide, Flash Enabled: Flash Design and Development for Devices*, and *Generator and Flash Demystified*.

Mike received his master's degree in international economics and European studies from the John Hopkins School of Advanced International Studies (SAIS) in 1998.

When he is not programming, Mike can be found playing Halo, trying to recover from his World of Warcraft addiction, working on scale models, or hanging out with his two daughters, Isabel and Aubrey, and his wife Cathy.

Mike maintains a weblog at *http://www.mikechambers.com/*.

Daniel Dura

Currently based in San Francisco, Daniel Dura is a Platform Evangelist at Adobe, focusing on Adobe AIR and Flash. Before joining Macromedia (which merged with Adobe in 2005), Daniel and his brother Josh founded Dura Media LLC, a RIA development company based in Dallas. While at Adobe, he was a member of the Central and Flex teams, as well as a product manager for developer relations.

Daniel has given presentations on Flash, Apollo, and Flex all over the world at user group meetings, conferences, and pretty much anywhere someone has been willing to listen. Outside of

his day job, he enjoys general aviation and is well on his way to earning his Private Pilot license.

Dragos Georgita

Based in Bucharest, Romania, Dragos Georgita is part of the Adobe AIR engineering staff, leading a group that focuses on JavaScript and Ajax support in the runtime. After graduating from the University Politehnica of Bucharest, he worked for a couple of companies and became interested in web technologies. Dragos has worked with both client and server technologies on different platforms and was part of the team that combined the best of the two words in the form of a search-engine-friendly Ajax framework.

Dragos also spent time trying to make the lives of web developers easier by working on automation tools for IDEs such as Adobe Dreamweaver. That period was important in developing his customer-oriented focus and attention to detail.

While part of the Adobe AIR team, Dragos is thrilled to be able to leverage his knowledge into the new breed of RIAs and to look for ways to improve the workflows for Ajax developers developing for Adobe AIR.

Outside of his day job, he enjoys spending time with his family and his 1-year-old daughter, Clara.

Kevin Hoyt

Kevin Hoyt is a Platform Evangelist with Adobe, who likes moving, breaking, blurring, and jumping over the lines of conventional technology. He seeks out every opportunity to congregate with other like-minded developers, and explores ways to escape any lines that form a box. Pushing the envelope of what technology can do, and how people perceive and interact with it, is his passion.

A frequent traveler, Kevin can generally be found deep in code while speaking with customers at conferences, in front of user

groups, or anywhere else they will give him time in front of an audience. The rest of the time he enjoys spending with his family at home in Parker, Colorado, and indulging his photography habit.

This current chapter in Kevin's career started when he accepted a job with Allaire Corporation, circa 2000, with focus on ColdFusion and JRun. Allaire was purchased by Macromedia in 2001, at which point he was able to unleash the latent designer within and help to promote the value of RIAs. Adobe acquired Macromedia in 2005, and Kevin now finds himself helping the company and its customers make sense of Adobe's increasingly large stable of products.

Acknowledgments

The authors would like to thank Mark Nichoson and Alisa Popolizio from Adobe and Steve Weiss and Michele Filshie from O'Reilly for helping to make this book possible in an incredibly short amount of time, and Editor Audrey Doyle. We would also like to thank Adrian Ludwig, Laurel Reitman, Oliver Goldman, Chris Brichford, Lucas Adamski, Rob Dixon, and Jeff Swartz, all from Adobe, for their input and work on the book.

Also, the authors would like to thank everyone on the Adobe AIR team for all of their dedication and hard work in getting a 1.0 runtime out the door.

CHAPTER 1

Introduction to Adobe AIR

Adobe AIR is a cross-platform desktop runtime created by Adobe that allows web developers to use web technologies to build and deploy Rich Internet Applications (RIAs) and web applications to the desktop.

> **NOTE**
>
> During its development cycle, Adobe AIR was referred to in public by its code name of "Apollo".

To better understand what Adobe AIR enables, and which problems it tries to address, it is useful to first take a look at the (relatively short) history of web applications.

A Short History of Web Applications

Over the past couple of years, there has been an accelerating trend of applications moving from the desktop to the web browser. This has been driven by a number of factors, which include:

- Growth of the Internet as a communication medium
- Relative ease of deployment of web applications

- Ability to target multiple operating systems via the browser
- Maturity of higher-level client technologies, such as the browser and the Flash Player runtime

Early web applications were built primarily with HTML and JavaScript, which, for the most part, relied heavily on client/server interactions and page refreshes. This page refresh model was consistent with the document-based metaphor for which the browser was originally designed, but provided a relatively poor user experience when displaying applications.

With the maturation of the Flash Player runtime, however, and more recently with Ajax-type functionality in the browser, it became possible for developers to begin to break away from page-based application flows. Developers began to offer richer application experiences via the browser. In a whitepaper from March 2002, Macromedia coined the term *rich Internet application* to describe these new types of applications in browsers, which "blend content, application logic and communications ... to make the Internet more usable and enjoyable." These applications provided richer, more desktop-like experiences, while still retaining the core cross-platform nature of the Web:

> Internet applications are all about reach. The promise of the web is one of content and applications anywhere, regardless of the platform or device. Rich clients must embrace and support all popular desktop operating systems, as well as the broadest range of emerging device platforms such as smart phones, PDAs, set-top boxes, game consoles, and Internet appliances.

NOTE

You can find the complete whitepaper and more information on RIAs at *http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf*.

The paper goes on to list some features that define RIAs:

- Provide an efficient, high-performance runtime for executing code, content, and communications
- Integrate content, communications, and application interfaces into a common environment
- Provide powerful and extensible object models for interactivity
- Enable rapid application development through components and reuse
- Enable the use of web and data services provided by application servers
- Embrace connected and disconnected clients
- Enable easy deployment on multiple platforms and devices

This movement toward providing richer, more desktop-like application experiences in the browser (enabled by the Flash Player runtime, and more recently by Ajax) has led to an explosion of web applications.

Today, the web has firmly established itself as an application deployment platform that offers benefits to both developers and end-users. Some of these benefits include the ability to:

- Target multiple platforms and operating systems
- Develop with relatively high-level programming and layout languages
- Allow end-users to access their applications and data from virtually any Internet-connected computer
- Easily push application updates to users

The growth of web applications can be seen in both the Web 2.0 trend, which consists almost entirely of web-based applications and APIs, and the adoption of web applications as a core business model by major companies and organizations.

Problems with Delivering Applications via the Browser

As web applications have become more complex, they have begun to push the boundaries of both the capabilities of the browser and the usability of the application. As their popularity grows, these issues become more apparent and important and highlight the fact that there are still a number of significant issues for both developers and end-users when deploying and using applications within the browser.

The web browser was originally designed to deliver and display HTML-based documents. Indeed, the basic design of the browser has not shifted significantly from this purpose. This fundamental conflict between document- and application-focused functionality creates a number of problems when deploying applications via the browser.

Conflicting UI

Applications deployed via the browser have their own user interface, which often conflicts with the user interface of the browser. This application-within-an-application model often results in user interfaces that conflict with and contradict each other. This can lead to user confusion in the best cases, and application failure in the worst cases. The classic example of this is the browser's Back button. The Back button makes sense when browsing documents, but it does not always make sense in the context of an application. Although a number of solutions attempt to solve this problem, they are applied to applications inconsistently, and users may not know whether a specific application supports the Back button or whether it will force their application to unload, causing it to lose its state and data.

Distance from the Desktop

Due in part to the web security model (which restricts access to the user's machine), applications that run in the browser often do not support the types of user interactions with the operating system that people expect from applications. For example, you cannot drag a file into a browser-based application and have the application act on that file. Nor can the web application interact with other applications on the user's computer.

RIAs have tried to improve on this by making richer, more desktop-like interfaces possible in the browser, but they have not been able to overcome the fundamental limitations and separation of the browser from the desktop.

Primarily Online Experience

Because web applications are delivered from a server and do not reside on the user's machine, web applications are primarily an online experience. Although attempts are underway to make offline web-based applications possible, they do not provide a consistent development model and they fail to work across different browsers, or they require users to install additional extensions to the browser. In addition, they often require users to interact with and manage their application and browser in complex and unexpected ways.

Lowest Common Denominator

Finally, as applications become richer and more complex and begin to push the boundaries of JavaScript and DHTML, developers are increasingly faced with differences in browser functionality and API implementations. Although these issues can often be overcome with browser-specific code, they lead to code that a) is more difficult to maintain and scale; and b) takes time away from function-driven development of feature functionality.

Although JavaScript frameworks are a popular way to help address these issues, they can offer only the functionality provided by the browser, and often they resort to the lowest common denominator of features among browsers to ease the development model. The result for JavaScript- or DHTML-based applications is a lowest common denominator user experience and interaction model, as well as increased development, testing, and deployment costs for the developer.

The fact that web applications have flourished despite these drawbacks is a testament to the attractiveness of having a platform with a good development model that has the ability to deliver applications to multiple operating systems. A platform that offered the reach and development model of the browser, while providing the functionality and richness of a desktop application, would provide the best of both worlds. This is what Adobe AIR aims to provide.

Introducing Adobe AIR

So, what is Adobe AIR, and how can it make web application development and deployment better?

Adobe AIR is a cross-operating-system runtime developed by Adobe that allows web developers to leverage their existing web development skills (such as Flash, Flex, HTML, JavaScript, and PDF) to build and deploy RIAs and content to the desktop.

In essence, Adobe AIR provides a platform in between the desktop and the browser, which combines the reach and ease of development of the web model with the functionality and richness of the desktop model.

It is important to step back for a second and point out what Adobe AIR is not. Adobe AIR is not a general desktop runtime meant to compete with lower-level application runtimes. Adobe AIR is coming from the Web to the desktop and is targeted at web developers. Its primary use case is to enable web applications and RIAs to be deployed to the desktop. This is a

very important but subtle distinction, as enabling web applications and RIAs on the desktop is the primary use case driving the Adobe AIR 1.0 feature set.

At its core, Adobe AIR is built on top of web technologies, which allow web developers to develop for and deploy to the desktop using the same technologies and development models that they use today when deploying applications on the Web.

Primary Adobe AIR Technologies

Three primary technologies are included within Adobe AIR, and they fall into two distinct categories: application technologies and document technologies.

Primary Application Technologies

Application technologies are technologies that can be used as the basis of an application within Adobe AIR. Adobe AIR contains two primary application technologies, HTML and Flash, both of which can be used on their own to build and deploy Adobe AIR applications.

HTML/JavaScript

The first core application technology within Adobe AIR is HTML and JavaScript. This includes a full HTML rendering engine, which includes support for:

- HTML
- JavaScript
- CSS
- XHTML
- Document Object Model (DOM)

Yes, you read that right. You don't have to use Flash to build Adobe AIR applications. You can build full-featured applications using just HTML and JavaScript. This usually surprises some developers who expect Adobe AIR to focus only on Flash.

However, at its core, Adobe AIR is a runtime targeted at web developers using web technologies—and what is more of a web technology than HTML and JavaScript?

The HTML engine used within Adobe AIR is the open source WebKit engine. This is the engine behind a number of browsers, including KHTML on KDE and Safari on Mac OS X.

NOTE

You can find more information on the WebKit open source project at *http://www.webkit.org*.

See Chapter 3 for a more in-depth discussion of WebKit within Adobe AIR.

Adobe Flash

The second core application technology that Adobe AIR is built on is Adobe Flash Player. Specifically, Adobe AIR is built on top of Adobe Flash Player 9, which includes the ECMAScript-based ActionScript 3 scripting language, as well as the open source Tamarin virtual machine (which will be used to interpret JavaScript in future versions of Firefox).

NOTE

You can find more information on the open source Tamarin project on the Mozilla website, at *http://www.mozilla.org/projects/tamarin/*.

Not only are all of the existing Flash Player APIs available within Adobe AIR, but some of those APIs have also been expanded and/or enhanced. Some of the functionality that the Flash Player provides to Adobe AIR includes:

- Just-in-time Interpreted ActionScript engine for speedy application performance

- Full networking stack, including HTTP and RTMP, as well as Binary and XML sockets
- Complete vector-based rendering engine and drawing APIs
- Extensive multimedia support including bitmaps, vectors, audio, and video

---

NOTE

Flash Player and ActionScript APIs are available to JavaScript within Adobe AIR applications.

Of course, the Adobe Flex 3 RIA framework is built on top of ActionScript 3, which means that you can take full advantage of all of the features and functionality that Flex offers in order to build Adobe AIR applications.

Primary Document Technologies

Document technologies within Adobe AIR refer to technologies that can be used for the rendering of and interaction with electronic documents.

PDF and HTML are the primary document technologies available within Adobe AIR.

PDF

The Portable Document Format (PDF) is the web standard for delivering and displaying electronic documents on the Web.

PDF functionality requires that Adobe Reader Version 8.1 be installed on the user's computer. If Adobe Reader 8.1 is installed, Adobe AIR applications will be able to take full advantage of all of the features that reader also exposes when running within a web browser.

HTML

HTML was originally designed as a document technology, and today it provides rich and robust control over content and text layout and styling. HTML can be used as a document technology within Adobe AIR—both within an existing HTML application as well as within a Flash-based application.

What Does an Adobe AIR Application Contain?

Now that we know what technologies are available to applications running on top of Adobe AIR (see Figure 1-1), let's look at how those technologies can be combined to build an Adobe AIR application.

Applications can consist of the following combinations of technologies:

- HTML/JavaScript only.
- HTML/JavaScript-based with Flash content.
- Flash only (including Flex).
- Flash-based with HTML content.
- All combinations can leverage PDF content.

Technology integration and script bridging

Because WebKit and Adobe Flash Player are included within the runtime, they are integrated with each other on a very low level. For example, when HTML is included within Flash content, it's actually rendered via the Flash display pipeline, which, among other things, means that anything you can do to a bitmap within the Flash Player (blur, rotate, transform, etc.) you can also do to HTML.

This low-level integration also applies to the script engines within Adobe AIR that run ActionScript and JavaScript. Adobe AIR enables script bridging between the two languages and environments, which makes the following possible:

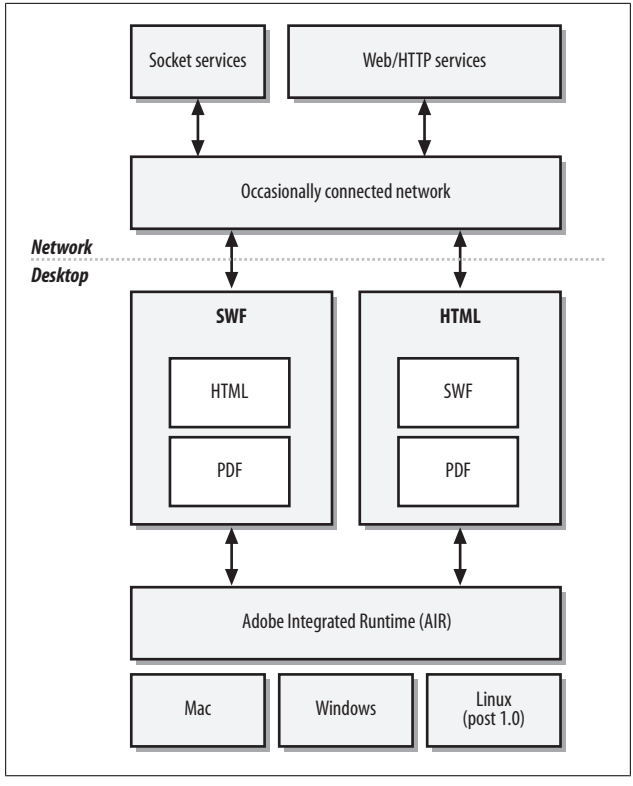

Figure 1-1. Adobe AIR application structure

- JavaScript code can call Adobe AIR, Flash Player, and ActionScript APIs.
- ActionScript code can call JavaScript APIs.
- ActionScript code can directly manipulate the HTML DOM.
- Event registration can occur both ways between JavaScript and ActionScript.

Note that the script bridging is "pass by reference." So, when passing an object instance from JavaScript to ActionScript (or

vice versa), changes to that instance in one environment will affect the instance in the other environment. Among other things, this makes it possible to instantiate and use Flash Player APIs directly from JavaScript, or to register and listen for events.

This low-level script bridging between the two environments makes it very easy for developers to create applications that are a combination of both HTML and Flash.

NOTE

We cover accessing ActionScript and Adobe AIR APIs from JavaScript in more detail in Chapter 3.

The result of all of this is that if you are a web developer using HTML and JavaScript, you already have all of the skills necessary to build an Adobe AIR application.

Adobe AIR Functionality

If Adobe AIR did not provide additional functionality and APIs and simply allowed web applications to run on the desktop, it would not be quite as compelling. Fortunately, Adobe AIR provides a rich set of programming APIs, as well as close integration with the desktop that allows developers to build applications that take advantage of the fact that they're running on the user's desktop.

Adobe AIR APIs

In addition to all of the functionality and APIs already offered by the Flash Player and WebKit engine, Adobe AIR provides additional functionality and APIs.

NOTE

Adobe AIR APIs are available to both ActionScript and JavaScript.

Some of the functionality includes, but is not limited to:

- Complete file I/O API
- Complete native windowing API
- Complete native menuing API
- Online/offline APIs to detect when service connectivity has changed
- Complete control over application chrome
- Local storage/settings APIs
- System notification APIs that tie into OS-specific notification mechanisms (not implemented in beta)
- Application update APIs
- SQLite embedded database

Note that functionality may be implemented directly within the runtime or on the framework layer (in Flex and JavaScript), or by using a combination of both.

Adobe AIR desktop integration

As discussed earlier, applications deployed via the browser cannot always support the same user interactions as desktop applications. This leads to applications that can be cumbersome for the user to interact with, as they do not allow the types of application interactions with which users are familiar.

Because an AIR application is a desktop application, it's able to provide the types of application interactions and experience that users expect from an application. This functionality includes, but is not limited to:

- Appropriate install/uninstall rituals
- Desktop install touch points (such as shortcuts)
- Rich drag-and-drop support:
  - — Between the operating system and AIR applications
  - — Between AIR applications
  - — Between native applications and AIR applications

- Rich clipboard support
- System notifications
- Native icons

Once installed, an AIR application is just another native application, which means that the operating system and users can interact with it in the same way as they do with any other application. For example, things such as OS-level application prefetching and application switching work the same with Adobe AIR applications as they do with native applications.

The goal is that the end-user does not need to know he is running an AIR application to be able to use it. He should be able to interact with the application in the same way that he interacts with any other application running on the desktop.

Security Model

All of this talk of APIs and desktop functionality brings up an important question: what about security? Because Adobe AIR applications have access to local resources, couldn't they theoretically do something harmful?

First, it is important to note that Adobe AIR runs on top of the operating system's security layer. It does not provide any way to get around or subvert this security. This is important, because it means Adobe AIR applications can work only within the permissions given to them by the operating system—and all current and any new security capabilities that the OS implements.

To run an Adobe AIR application, a user must download the application to the desktop, go through an install ritual, and then launch the application. This is an experience very similar to downloading and installing a desktop application. The similarity is not an accident. Adobe AIR applications run in a fundamentally different security content than applications that run within a browser. It is a security context closer to that of a native application than a web application.

To enable safe browsing, the browser security model limits all I/O capabilities of web applications. This includes restricting their ability to work with local resources, limiting what network resources are accessible, and constraining their user interface. The browser only allows applications to connect with data that is associated with (usually, provided by) a server located on a single web domain. In addition, the browser provides a trusted UI for users to understand the origin of the application and control the state of the application. This model is sufficient for applications that are connected to a single service provider and rely on that service for data synchronization and storage.

Some web developers have also stretched the browser security model by integrating data from multiple sources and/or by experimenting with user interfaces that are inconsistent with the browser chrome. Some of these applications require browser plug-ins with capabilities that aren't currently provided by the browsers. Others take advantage of browser features such as user notification or customized security configurations to allow greater or lesser security to applications from specific domains. These mechanisms allow web developers to build more powerful applications, but they also are straining the browser security model.

Rather than trying to extend the web browser so that it can act as both a browser and a flexible application runtime, Adobe AIR provides a flexible runtime for building applications using web technologies. Adobe AIR allows web developers to build applications that incorporate data from multiple sources, provide users with control over where and how their data is stored, and produce user experiences that are not possible within the browser's user interface. Because Adobe AIR applications must be installed on the desktop and require users to specifically trust them, AIR applications can safely exercise these capabilities. Browser-based applications cannot be granted these capabilities if the browser is to continue to fulfill its role as an application for safely browsing any website on the Internet.

The Adobe AIR security model has a number of implications for application developers and users. For application developers, it means that content within an installed AIR application has capabilities that should not be exposed to any untrusted content, including files from the Web. The runtime has a number of features that are designed to reinforce that distinction and to help developers build applications using security best practices.

This also means that users should not install Adobe AIR applications from sources they do not trust. This is very similar to current practices for native desktop applications and for browser plug-ins. Many applications and web content require that a browser plug-in (such as Flash Player or Apple QuickTime) be installed in order to work. The Firefox browser has a very accessible extensibility layer that essentially allows any developer to extend the browser. These applications, plug-ins, and extensions can do potentially harmful things and therefore require that the user trust the source of the content.

Finally, one of the capabilities that will be included in the Adobe AIR 1.0 release is the ability of the runtime to verify the identity of an application's publisher. Users should carefully consider whether they want to trust the publisher of an application, as well as whether they want to install an application that hasn't been signed.

Adobe AIR Development Toolset

One of the reasons web applications have been successful is that they allow developers to easily deploy applications that users can run regardless of which operating system they are on. Whether on Mac, Windows, Linux, Solaris, or cell phones, web applications provide reach.

However, success is based not only on cross-platform deployment, but also on the cross-platform nature of the development environment. This ensures that any developer can develop for —and leverage—the technology. Neither the runtime nor the development tools are tied to a specific OS.

The same is true of Adobe AIR. Not only does Adobe AIR provide the cross-platform reach of web applications, but, just as importantly, Adobe AIR applications can be developed and packaged on virtually any operating system.

Because Adobe AIR applications are built with existing web technologies such as HTML and Flash, you can use the same tools that you use to create browser-based content to create Adobe AIR applications. The Adobe AIR SDK provides two free command-line tools that make it possible to test, debug, and package Adobe AIR applications with virtually any web development and design tool.

ADL	Allows Adobe AIR applications to be run without having to first install them
ADT	Packages Adobe AIR applications into distributable installation packages

Although Adobe has added support to its own web development and design tools for authoring Adobe AIR content (including Adobe Flex Builder, Adobe Flash CS3, and Adobe Dreamweaver), Adobe programs are not required to create applications. Using the Adobe AIR command-line tools, you can create an AIR application with any web development tool. You can use the same web development and design tools that you are already using today.

NOTE

We will cover the development workflow in depth in Chapter 2.

Is Adobe AIR the End of Web Applications in the Browser?

So, by this point, you may be saying to yourself, "Gee, Adobe AIR sure sounds great! Why would anyone ever want to deploy an application to the browser again? Is Adobe AIR the end of web applications within the browser?"

No.

Let's repeat that.

No.

Adobe AIR addresses many of the problems with deploying web applications via the browser. However, there are still advantages to deploying applications via the browser. The fact that there are so many web applications despite the disadvantages discussed earlier is a testament to the advantages of running within the browser. When those advantages outweigh the disadvantages, developers will still deploy their applications via the web browser.

But it's not necessarily an either/or question. Because Adobe AIR applications are built using web technologies, the application that you deploy via the web browser can be quickly turned into an Adobe AIR application. You can have a web-based version that provides the browser-based functionality, and then also have an AIR-based version that takes advantage of running on the desktop. Both versions could leverage the same technologies, languages, and code base. Indeed, some of the most popular early Adobe AIR applications, such as Finetune Desktop and eBay Desktop, complement existing web applications.

NOTE

You can find more information on Finetune Desktop at *http://www.finetune.com/desktop/*.

You can find more information on eBay Desktop at *http://desktop.ebay.com*.

Adobe AIR applications complement web applications. They do not replace them.

CHAPTER 2

Getting Started with Adobe AIR Development

This chapter discusses how to start developing applications for Adobe AIR using HTML and JavaScript. It covers:

- Installing Adobe AIR
- Configuring the Adobe AIR SDK and command-line tools
- Creating your first AIR application
- Testing AIR applications
- Signing, packaging, and deploying AIR applications

Once you have completed this chapter, your environment for developing AIR applications should be configured correctly, and you should have a solid understanding of how to begin to build, test, and deploy Adobe AIR applications.

What Do You Need to Develop Adobe AIR Applications?

You need a number of items to begin developing AIR applications.

Adobe AIR Runtime

The Adobe AIR runtime is required to test application icons and deploy AIR applications. You can download the runtime for free from *http://www.adobe.com/go/getair*.

Adobe AIR SDK

The Adobe AIR SDK contains command-line tools, sample files, and other resources to make it easier to develop AIR applications. In particular, we will be using the command-line tools included in the SDK (ADL and ADT), which will allow us to test, sign, and package our AIR applications from virtually any development environment.

You can download the AIR SDK for free from *http://www.adobe.com/go/getairsdk*.

HTML/JavaScript IDE or Editor

Building AIR applications with HTML and JavaScript requires that you have a way to create the HTML and JavaScript files. You can use any tool that supports creating and editing text files (such as VIM or Notepad), although it's recommended that you use a tool that has richer support for working with HTML and JavaScript files, such as Adobe Dreamweaver, Panic's Coda, or Aptana Studio.

NOTE

You can find more information on Adobe Dreamweaver at *http://www.adobe.com/go/dreamweaver*, Panic's Coda at *http://www.panic.com/coda/*, and Aptana Studio at *http://www.aptana.com*.

Supported Operating System

Although it is possible to develop and package AIR applications on virtually any operating system (including Linux), you can test and deploy AIR applications only on operating systems supported by Adobe AIR.

The supported operating systems are:

- Microsoft Windows 2000 SP4
- Microsoft Windows XP SP2
- Windows Vista Home Premium, Business, Ultimate, or Enterprise
- Mac OS 10.4.910 and later (Intel and PowerPC)

> **NOTE**
>
> H.264 video playback on a Mac requires an Intel processor.

Adobe is also currently working on adding support for Linux.

Uninstalling Prerelease Versions of Adobe AIR

If you have installed prerelease (alpha and/or beta) versions of Adobe AIR, you may need to uninstall them before installing the 1.0 runtime. However, whether this is strictly required can be a little tricky to determine.

Table 2-1 lists the prerelease versions of Adobe AIR and whether they should be uninstalled before installing Adobe AIR 1.0.

Table 2-1. Prerelease versions of Adobe AIR

Version	Uninstall?
Alpha	Uninstall
Beta 1	Will be uninstalled automatically by the Adobe AIR 1.0 installer
Beta 2	No need to uninstall

Version	Uninstall?
Beta 3	No need to uninstall

As Table 2-1 shows, the beta 2 and beta 3 runtimes can run side by side with Adobe AIR 1.0. This allows you to run applications built for beta 2 and beta 3 until those betas expire.

If you do not know whether you have installed the alpha version, it is a good idea to go ahead and uninstall all prerelease versions of Adobe AIR.

Uninstalling on Windows

To uninstall all prerelease versions of Adobe AIR on Windows, follow these steps:

1. In the Windows Start menu, select Settings→Control Panel.
2. Select the Add or Remove Programs control panel.
3. Select the prerelease version of Adobe AIR to uninstall (depending on the version, it may be referred to by its code name of "Apollo").
4. Click the Change/Remove button.

Uninstalling on Mac

Follow these steps to uninstall all prerelease versions of Adobe AIR on the Mac. Depending on the prerelease version(s) of Adobe AIR that you have installed, all steps may not apply.

1. Run the Adobe AIR Uninstaller in the */Users/<User>/Applications* directory (where `<User>` is your system user account name).
2. Run the Adobe AIR Uninstaller in the */Applications* directory.
3. Delete the */Library/Frameworks/Adobe Apollo.framework* directory.

4. Delete the */Library/Receipts/Adobe Apollo.pkg* file.
5. Empty the Trash.

Once you have done this, you are ready to install the 1.0 runtime.

Installing Adobe AIR

Although it is not necessary to have Adobe AIR installed on your computer to develop and test Adobe AIR applications, it is useful to have it to try other AIR applications and to test your final application's deployment and packaging.

Installing the runtime is simple, and requires downloading and running the Adobe AIR Installer.

1. Download the AIR Installer from *http://www.adobe.com/go/getair*.
2. Launch the installer. On a Mac, you must first mount the *.dmg* file, which contains the installer.
3. Follow the installation instructions.

NOTE

It is also possible to install Adobe AIR directly from the runtime via express install. We will cover this in Chapter 4.

As Adobe AIR is simply a runtime and not an application that can be launched, the easiest way to confirm that it is installed correctly is to try installing an AIR application. You can do this by either downloading an AIR application and installing it, or following the instructions later in the chapter to build a simple AIR application.

NOTE

You can download sample AIR applications from Adobe's web site, at *http://www.adobe.com/go/air_samples*.

Uninstalling Adobe AIR

The process for uninstalling Adobe AIR is different depending on your operating system.

Uninstalling on Windows

On Windows, you can uninstall Adobe AIR the same way that you uninstall any other application. Just select Adobe AIR in the Add/Remove Programs section of the Control Panel.

Uninstalling on an Mac

The Adobe AIR installer places an uninstall application on the user's system when it is installed. To uninstall Adobe AIR, launch the uninstaller named *Adobe AIR Uninstaller* which you can find in the */Applications/Utilities* directory.

Setting Up the Adobe AIR SDK and Command-Line Tools

The Adobe AIR SDK beta contains tools, samples, and code that make it easier to develop, test, and deploy applications. In particular, it contains two command-line tools that we will use:

ADL
: You use this tool to launch and test an Adobe AIR application without having to install it first.

ADT
: You use this tool to package and sign an AIR application for distribution.

Installing the Adobe AIR SDK

To ease development, you should place the path to these files within your system's path. This will allow you to execute the tools from anywhere on your system.

The command-line tools are located in the *bin* directory within the SDK.

1. Download the Adobe AIR SDK from *http://www.adobe.com/go/getairsdk*.
2. Open the SDK:

   a) On Windows, uncompress the ZIP archive.

   b) On Mac, mount the *.dmg* file.
3. Copy the contents of the SDK to your system (we will refer to this location as `<SDK_Path>`).

> **NOTE**
>
> To run the command-line tools, you need to copy only the *bin*, *lib*, and *runtime* directories from the SDK.
>
> It's important that the *bin*, *lib*, and *runtime* directories within the SDK maintain their relative paths to each other.

4. At this point, you should have at least the following three directories: `<SDK_Path>/bin`, `<SDK_Path>/lib`, and `<SDK_Path>/runtime`. The ADL and ADT command-line tools are located in the *bin* directory.

Figure 2-1. Placing command-line tools in the system path on Windows

Placing the Command-Line Tools Within the System Path

All that is left to do is to place the *<SDK_Path>/bin* directory into your system path so that you can execute the command-line applications from anywhere on your system.

The instructions for this are different depending on whether you are on a Mac or Windows-based system.

Windows

If you are on a Windows system, follow these steps:

1. Open the System Properties dialog box and click the Advanced tab. You can find this in the System settings in the Control Panel.
2. Click the Environment Variables button.
3. Select the PATH entry and then click the Edit button. Add the path to the *bin* directory to the end of the current variable value, separating it from previous values with a semicolon:

   ```
   ; <SDK_Path>/bin
   ```

   Figure 2-1 Editing PATH variables in Windows.

4. Click OK to close the panels.

To test the installation, open a new Windows Console (Start→Run→Console), and type **adt**.

NOTE

Make sure you open a new Console window in order to ensure the new PATH settings take affect.

You should see output similar to this:

```
usage:
  adt -package SIGNING_OPTIONS <air-file> <app-desc> FILE_ARGS
  adt -prepare <airi-file> <app-desc> FILE_ARGS
  adt -sign SIGNING_OPTIONS <airi-file> <air-file>
  adt -checkstore SIGNING_OPTIONS
  adt -certificate -cn <name> ( -ou <org-unit> )?
( -o <org-name> )? ( -c <country> )?
 <key-type> <pfx-file> <password>
  adt -help

SIGNING_OPTIONS: -storetype <type> ( -keystore <store> )?
( -storepass <pass> )? ( -keypass <pass> )? ( -providerName
 <name> )? ( -tsa <url> )?
FILE_ARGS: <fileOrDir>* (( -C <dir> <fileOrDir>+ ) |
( -e <file> <path> ))* -C dir
```

This means the tools are configured correctly.

If you get an error stating that the file cannot be found, do the following:

- Make sure the *bin*, *lib*, and *runtime* directories are included in the *<SDK_Path>* directory.
- Make sure you included the path to the *<SDK_Path>* directory correctly in the PATH environment variable.
- Make sure you opened a new Console window before running the command.

Mac

There are a number of ways to add the path to the AIR SDK to your system path, depending on which shell you are using and how you specify user environment variables.

The following instructions explain how to modify your `PATH` environment variable if you are using the bash shell:

1. Open the Terminal program (*/Applications/Utilities/Terminal*). Make sure you're in your home directory by typing **`cd`** and pressing Enter.
2. Check to see whether one of two files is present. Enter the command **`ls -la`**.
3. Look for a file named either *.profile* or *.bashrc*.
4. If you have neither file, create the *.profile* file with the command **`touch .profile`**.
5. Open the *.profile* or *.bashrc* file with a text editor.
6. Look for a line that looks similar to this:

   ```
   export PATH=$PATH:/usr/local/bin
   ```

7. Add the path to the *<SDK_Path>/bin* directory to the end of this line. For example, if *<SDK_Path>/bin* is at */airsdk/bin*, the export path should look something like this:

   ```
   export PATH=$PATH:/usr/local/bin;/airsdk/bin
   ```

   Make sure you separate the entries with a colon.
8. If the file is empty, add the following line:

   ```
   export PATH=$PATH:/airsdk/bin
   ```

9. Save and close the file.
10. Run the command **`source .profile`** to load the new settings (or *.bashrc*, if that is the file you edited).
11. Confirm that the new settings have taken effect by typing **`echo $PATH`** and pressing Enter. Make sure the *<SDK_Path>/bin* path is included in the output.
12. To test the installation, open a Terminal window and type **`adt`**.

You should see output similar to this:

```
usage:
  adt -package SIGNING_OPTIONS <air-file> <app-desc>
  FILE_ARGS
  adt -prepare <airi-file> <app-desc> FILE_ARGS
  adt -sign SIGNING_OPTIONS <airi-file> <air-file>
  adt -checkstore SIGNING_OPTIONS
  adt -certificate -cn <name> ( -ou <org-unit> )?
  ( -o <org-name> )? ( -c <country> )?
  <key-type> <pfx-file> <password>
  adt -help

SIGNING_OPTIONS: -storetype <type> ( -keystore
<store> )? ( -storepass <pass> )? ( -keypass <pass> )?
( -providerName <name> )? ( -tsa <url> )? FILE_ARGS:
<fileOrDir>* (( -C <dir> <fileOrDir>+ ) | ( -e <file>
<path> ))* -C
```

This means the tools are configured correctly.

If you get an error stating that the file cannot be found, do the following:

- Make sure the *bin*, *lib*, and *runtime* directories are included in the *<SDK_Path>* directory.
- Make sure you included the path to *<SDK_Path>/bin* correctly in the `PATH` environment variable.
- Make sure you either opened a new Terminal window, or ran *source* on your configuration file.

Creating a Simple AIR Application with HTML and JavaScript

Now that we have installed and configured Adobe AIR and the Adobe AIR SDK, we are ready to build our first AIR application.

We will build a very simple "Hello World" example. Once we have built and tested the application, our development environment will be set up and ready to build more complex and functional AIR applications.

Creating the Application Files

Every Adobe AIR application contains a minimum of two files. The first file is the *root content file*. This is the main HTML or SWF file for the application, and is the file that will be displayed/executed when the application first starts up.

The second file is called the *application descriptor file*, and it is an XML file that provides metadata to Adobe AIR about the application.

Let's create these files for our application:

1. Create a new folder called *AIRHelloWorld*.
2. Inside this folder, create two new files called *AIRHelloWorld.html* and *AIRHelloWorld.xml*.
3. Open each file using your favorite text or HTML editor/IDE.

Understanding application descriptor files

The application descriptor file is an XML file required for each AIR application. It provides general metadata (such as the application name and description) to Adobe AIR, as well as information on how the application should be run. This includes specifying the root application file for the application and the window mode that the initial application window should use.

First, let's look at the entire application descriptor file (*AIRHelloWorld.xml*) for our application, and then we will go into more detail regarding each item within the file.

NOTE

You can find a sample application descriptor file in the AIR SDK in the *templates* folder.

Open *AIRHelloWorld.xml* and type in the following text:

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/1.0">

    <id>com.oreilly.AIRHelloWorld</id>
    <filename>AIRHelloWorld</filename>
    <name>AIR Hello World</name>
    <description>A simple AIR hello world application</
    description> <version>1.0</version>

    <initialWindow>
         <content>AIRHelloWorld.html</content>
         <title>AIR Hello World</title>
         <systemChrome>standard</systemChrome>
         <transparent>false</transparent>
         <visible>true</visible>
         <minimizable>true</minimizable>
         <maximizable>true</maximizable>
         <resizable>true</resizable>
    </initialWindow>
</application>
```

The content should be pretty self-explanatory, but let's go through it line by line to understand what is going on.

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

The namespace specifies the version of Adobe AIR that the application targets—in this case 1.0.

Lets look at the next element:

```
<id>com.oreilly.AIRHelloWorld</id>
```

The `id` element is important; it specifies a unique ID for the application. Adobe AIR uses this ID to differentiate one application from another.

As you can see, it uses the reverse domain format, which you may be familiar with from some programming languages such as Java, ActionScript, and some JavaScript frameworks. You can create your own ID using your domain name and application name.

The next section of elements specify general metadata about the application:

```
<filename>AIRHelloWorld</filename>
<name>AIR Hello World</name>
<description>A simple AIR hello world application</
description> <version>1.0</version>
```

Table 2-2 lists each element and provides a description.

Table 2-2. Application Meta Data Elements

Element	Description
`filename`	The name of the native application executable that will be created.
`name`	The name of the application. This is the name that will be exposed to the operating system and user.
`description`	Optional. A human-readable description of the application that will be presented to the user during the installation process.
`version`	Required. Specifies the version of the applications (such as "1.0", "v1", etc.).

The next element is the `initialWindow` tag, which contains the elements that specify how the application should be run by the runtime:

```
<initialWindow>
     <content>AIRHelloWorld.html</content>
     <title>AIR Hello World</title>
     <systemChrome>standard</systemChrome>
     <transparent>false</transparent>
     <visible>true</visible>
     <minimizable>true</minimizable>
     <maximizable>true</maximizable>
     <resizable>true</resizable>
</initialWindow>
```

The `content` element is required and points to the main root file of the application, which in this case is an HTML file.

NOTE

The application descriptor file and root content file must be in the same folder.

The `initialWindow` element has a number of other elements that specify the initial window parameters and chrome of the application when it is first launched (see Table 2-3).

Table 2-3. initalWindow elements

Element	Description
`title`	The title that will appear in the main application window. Optional.
`systemChrome`	The type of the system chrome that the application should use (`standard` or `none`).
`transparent`	Whether the application background should be transparent. This option is applicable only if `systemChrome` is set to `none`. Optional. Default value is `false`.
`visible`	Whether the application is visible when it is first launched. This is useful if your application needs to perform some complex initialization before displaying the UI to the user. Optional. Default is `false`.
`minimizable`	Whether the application can be minimized. Optional. Default is `true`.
`maximizable`	Whether the application can be maximized. Optional. Default is `true`.
`resizable`	Whether the application can be resized. Optional. Default is `true`.

For our example, we will use the operating system's window chrome.

NOTE

Notice that the default value of the `visible` element is `false`. This means that if you do not explicitly set the element to `true`, your application will have no UI when it launches. In this case, you will have to programmatically set the `visible` property to `true`.

This can be useful if the application needs to perform some initialization or layout when it first launches. You can allow the application to do its layout first, and then only display the UI to the user once the layout is complete.

This is all that is required for the application descriptor file for our application. At this point, we are ready to create the main HTML file for our application.

Creating the root application file

The root application file is the main file for the application that will be loaded when the application is launched. This file can be either a compiled Flash file (SWF) or an HTML file.

For this chapter, we will create a very simple HTML file to ensure that our development environment is configured correctly. We will cover more advanced AIR API usage in Chapter 3 and Chapter 4.

```
<html>
<head>
    <title>AIRHelloWorld</title>

    <script>
        function init()
        {
            runtime.trace("init function called");
        }
    </script>

</head>
<body onload="init()">
    <div align="center">Hello World</div>
```

```
</body>
</html>
```

As you can see, this is a very basic HTML file that displays "Hello World" and calls a JavaScript function once the file has loaded and initialized.

A couple of lines are worth pointing out:

```
<body onload="init()">
```

This line says we are just using the standard `onload` event on the `body` element to get an entry point for JavaScript into our application.

```
<script>
    function init()
    {
        ...
    }
</script>
```

This line says we are using a standard JavaScript function to capture the `onload` event.

Accessing Adobe AIR APIs

Looking at the `init` JavaScript function, you'll see some code you may not be familiar with:

```
runtime.trace("init function called");
```

This is the only AIR-specific code/markup in the entire application. The `runtime` property is a property placed on the `window` object by Adobe AIR and provides an entry point into the Adobe AIR engine and APIs. The `trace` function is a top-level AIR API that takes a string and prints it out to the command line (when the application is launched via the command line).

All access to AIR-specific APIs (including Flash Player APIs) is accomplished from JavaScript via the `runtime` property. We will cover this in more detail throughout the rest of the book.

NOTE

Checking for the existence of the `runtime` property is a simple way to determine whether your HTML and JavaScript application is running within Adobe AIR. To check for the property, run the following code:

```
if(window.runtime)
{
 //running within AIR
}
```

Now that we have created both the application descriptor file and the root HTML application file, we are ready to run and test our application within the runtime.

Testing the Application

Although a number of HTML IDEs (such as Adobe Dreamweaver) have support for launching and testing AIR applications directly from within the IDE, we will focus on launching and testing AIR applications using the ADL command-line tool included within the SDK. This will provide a solid basis for an understanding of what is going on. It also provides the most flexibility in integrating the development process with other IDEs, editors, and workflows.

Using ADL to Launch the Application

The first step in testing the application is to run it as an AIR application to make sure that:

- There are no errors in the application descriptor file
- The application launches
- The HTML renders correctly
- The JavaScript code functions as expected

Although we could package up the entire application and then install it, this would be tedious, and it would make it difficult

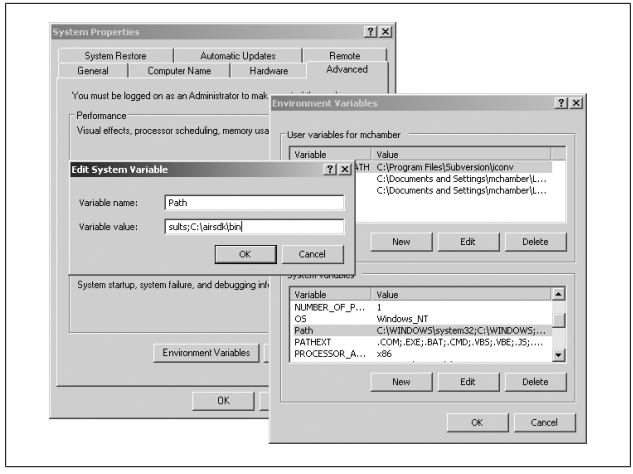

Figure 2-2. AIRHelloWorld application running from ADL on Mac OS X

to quickly iterate on and test new versions. Luckily, the Adobe AIR SDK provides a command-line tool called ADL, which allows you to launch an AIR application without having to install it first.

To test the application:

1. Open a Terminal window (on the Mac) or a Console window (on Windows).
2. Change to the directory that contains the *AIRHello World.html* and *AIRHelloWorld.xml* files.
3. Run ADL with the following command, passing in the name of the application descriptor file:

```
adl AIRHelloWorld.xml
```

This should launch your application within the standard system chrome of your operating system (see Figure 2-2).

If the application does not launch correctly, or if you get an error, do the following:

- Make sure you have configured the SDK correctly so that the ADL tool can be found.
- Make sure you are running the ADL command from the same directory that contains the *AIRHelloWorld.xml* file.
- Make sure your application descriptor file contains well-formed XML.
- Make sure the information in the application descriptor file is correct. Pay particular attention to the application attributes and the `initialWindow` value.
- Make sure the *AIRHelloWorld.html* and *AIRHelloWorld.xml* files are in the same directory.

Now that we have fixed any issues and our application is running correctly, we can explore how to get information from the application at runtime.

Capturing Output from the Application at Runtime

When running applications from the command line via ADL, you can get runtime information and debugging information from the application in a number of ways.

Runtime JavaScript errors

Any runtime errors that arise from JavaScript execution while an AIR application is launched via ADL is running will be output to ADL's standard out.

Let's modify our application to cause it to generate a JavaScript runtime error. Change the contents of *AIRHelloWorld.html* to the following

```
<html>
<head>
    <title>AIRHelloWorld</title>

    <script>
        function init()
        {
            runtime2.trace("init function called");
        }
```

```
    </script>

</head>
<body onload="init()">
    <div align="center">Hello World</div>
</body>
</html>
```

All we did was change the `init` function to try to access a property named `runtime2` that does not exist:

```
runtime2.trace("init function called");
```

Save the file, and run the application from ADL:

```
adl AIRHelloWorld.xml
```

The application should launch, and you should see the following error output from the command line from which you launched the application:

```
ReferenceError: Can't find variable: runtime2
init at app:/AIRHelloWorld.html : 8
init at app:/AIRHelloWorld.html : 8
onload at app:/AIRHelloWorld.html : 13
```

This output provides the error, which in this case is that the variable named `runtime2` cannot be found, as well as the line number on which the error occurred (8) and a stack trace of the call. You can use this information to track down any errors within your application.

There are also times when the application may not be functioning correctly, but is not throwing any errors. In such cases, it is useful to be able to capture information about the state of the application at runtime to track down any issues.

Adobe AIR provides a function to make it possible to send information from the application to standard out at runtime.

runtime.trace

As we touched on earlier in the chapter, Adobe AIR provides a mechanism for sending strings from JavaScript to the command line.

The `trace` function on the `runtime` property takes a string, which will then be output to ADL's standard out. Here is an example of its usage:

```
runtime.trace("This will be sent to standard out");
```

This can be useful for tracking information about the state of the application without having to interrupt the execution of the program.

Any non-string objects passed to `trace()` will have their `toString()` function called. The JavaScript `Object` object provides a default `toString()` implementation, although some classes (such as `Array`) implement more context-sensitive `toString()` functions.

Here is an example of tracing an array that contains various data types:

```
var a = ["a", 1, {foo:"bar"}];
runtime.trace(a);
```

This will result in the following output on the command line from ADL:

```
a,1,[object Object]
```

Of course, you can implement your own `toString()` method on your custom JavaScript classes, or override `toString()` functions on existing classes to provide more class-specific output.

Packaging and Deploying the AIR Application

Now that we understand how to build, test, and debug an AIR application, we are ready to create an AIR file that will allow us to deploy and distribute our application.

What Is an AIR File?

An AIR file is a ZIP-based application distribution package that is used to distribute AIR applications. It contains all of the files necessary to install and run an AIR application, and Adobe AIR

uses it to create and install an AIR application onto the user's system.

The AIR file is created by the ADT command-line tool included in the AIR SDK and is used to distribute the application to other users.

NOTE

Installing an AIR file requires that Adobe AIR already be installed on the user's system.

An AIR file requires a minimum of two files: the application descriptor file and a root application file. However, you can also include other files, icons, directories, and assets that will be bundled with the AIR file and installed alongside your application. These files will then be available to the application at runtime.

In addition, you will also need a certificate to digitally sign your application.

Digitally Signing AIR Files

Adobe AIR requires that all AIR applications be digitally signed. There are two ways to do this.

Signing with a self-signed certificate

Developers can use ADT to digitally sign an AIR file with a self-signed certificate. You self-sign an AIR file by generating a self-signed certificate, and then signing the AIR file with it. Self-signing AIR files provides little security, and no way to verify that the author is who she says she is. When the application is installed, Adobe AIR will warn users that the publisher of the application cannot be verified.

NOTE

You cannot switch among certificate types when updating applications.

AIR files signed with self-signed certificates are meant primarily for development purposes. If you plan to widely distribute your application to the public, you should sign your application with a certificate issued by a respected and well-known Certification authority (CA).

Signing with a CA-issued certificate

ADT also has support for signing applications using a verified certificate from an established CA. This allows Adobe AIR to verify the publisher of the application, and to reflect this information in the installation dialog.

NOTE

You can find more information on signing AIR files, including a list of CAs that publish certificates that work with Adobe AIR, in the Adobe AIR documentation at *http://www.adobe.com/go/learn_air_html*.

With both types of certificates—self-signed and CA-issued—Adobe AIR can verify that the AIR file has not been tampered with.

Because we will not be redistributing the application we are creating, we will be signing our AIR file with a self-signed certificate.

Creating an AIR File Using ADT

Creating a self-signed AIR file requires only two steps:

1. Use ADT to create a self-signed certificate.

2. User ADT to create the AIR file, digitally signed with the self-signed certificate.

Generating a self-signed certificate

Before signing an AIR file with a self-signed certificate, we need to first generate the certificate.

We can use ADT to generate a self-signed certificate with the following command-line options:

```
adt -certificate -cn COMMONNAME KEYTYPE CERTFILE PASSWORD
```

Table 2-4 lists and explains these command-line options.

Table 2-4. ADT Signing Options

Command-line option	Explanation
COMMONNAME	The common name of the new certificate. This is the name associated with the certificate.
KEYTYPE	The type of key to use for the certificate, either 1024-RSA or 2048-RSA.
CERTFILE	The filename in which the certificate will be stored.
PASSWORD	The password for the certificate.

To generate a self-signed certificate, follow these steps:

1. Open a Terminal (Mac OS X) or Console (Windows) window.
2. Change to the directory that contains *AIRHello World.html* and *AIRHelloWorld.xml*.
3. Run the following command:

   ```
   adt -certificate -cn foo 1024-RSA test_cert.p12 mypass
   ```

   For this example, we will give the certificate a common name of "foo" with a password of "mypass".

This generates a self-signed certificate, and stores it in a file named *test_cert.p12*.

NOTE

You can use the same self-signed certificate to sign multiple AIR files.

At this point, you should have a file named *test_cert.p12* in the same directory as your application files. You can now use this file to digitally self-sign your AIR file.

Generating an AIR file

The ADT command-line tool included in the Adobe AIR SDK is used to create AIR files. Its usage format is:

```
adt -package SIGNINGOPTIONS AIRFILENAME FILESTOINCLUDE
```

To create an AIR file that is signed with a self-signed certificate, follow these steps:

1. Open a Terminal (Mac OS X) or Console (Windows) window.
2. Change to the directory that contains *AIRHello World.html* and *AIRHelloWorld.xml*.
3. Run the following command:

```
adt -package -storetype pkcs12 -keystore test_cert.p12
AIRHelloWorld.air AIRHelloWorld.xml AIRHelloWorld.html
```

4. Upon running the command, you should be prompted for the password for the certificate. Enter the password for the certificate, which for this example is **`mypass`**.

NOTE

When signing the AIR file, ADT will attempt to connect to a timeserver on the Internet to timestamp the file. If it cannot connect to the timeserver, you will receive the following error:

```
Could not generate timestamp
```

When developing and self-signing your AIR files, you can get around this error by telling ADT to not timestamp the AIR file; you do this by adding the following option to the signing options on the command line:

```
-tsa none
```

In this case, the entire command would be:

```
adt -package -storetype pkcs12 -keystore test_
cert.p12 -tsa none AIRHelloWorld.air
 AIRHelloWorld.xml AIRHelloWorld.html
```

This should create a file named *AIRHelloWorld.air* in the same directory as your application files. If the file is not created, or if you receive any errors, do the following:

- Make sure you have configured the SDK correctly, and that the ADT tool can be found on your system's path.
- Make sure you are running the ADT command from the same directory that contains the *AIRHelloWorld.xml* file.
- Make sure your application descriptor file contains well-formed XML.
- Make sure the information in the application descriptor file is correct. Pay particular attention to the `application` attributes, and the `content` element.
- Make sure the *AIRHelloWorld.html*, *test_cert.p12*, and *AIRHelloWorld.xml* files are in the same directory.
- Make sure you entered the same password you used when generating the certificate file.

Testing and Installing the AIR File

Now that we have created the AIR file for our application, the only step left is to test the file and make sure it installs correctly.

Testing the AIR file requires trying to install it onto the system, and then launching it:

1. Switch to the directory that contains the AIR file in Windows Explorer (Windows) or the Finder (Mac OS X).
2. Double-click the AIR file.
3. Follow the instructions in the Install dialog box.
4. On the last screen of the Install dialog box, make sure "Start Application after installation" is checked.

The application should launch and run. If it does not launch, or if you receive an error, do the following:

- Make sure you have correctly installed the 1.0 version of Adobe AIR.
- Make sure there were no errors when you created the AIR file via ADT.
- Make sure you have uninstalled any prerelease versions of Adobe AIR.

Once you have confirmed that the application is installed and runs correctly, you can relaunch it by clicking its icon. The default shortcut location varies, depending on your operating system. In Mac, the default shortcut is */Applications*. In Windows, it is Start Menu→Programs→*<APPLICATION NAME>*.

Deploying the AIR File

Now that we have successfully created and packaged our AIR application, it's time to distribute it. We can distribute the AIR file via the Web, or directly via CD-ROM or other distribution mechanisms.

NOTE

You can find extensive information on how to seamlessly deploy Adobe AIR applications on the Web in the Adobe AIR documentation, as well as in the tutorial at *http://www.adobe.com/devnet/air/articles/air_badge_install.html*.

Setting the MIME type

One thing to watch out for when distributing AIR files for download from a web server is that the MIME type is set correctly on the server. If the MIME type is not set correctly, web browsers may treat the AIR file as a ZIP file (and may rename it in the process), or may display the raw bytes of the AIR file in the browser, instead of downloading it to the user's system.

The correct MIME type for an AIR file is:

```
application/vnd.adobe.air-application-installer-package+zip
```

For example, to set the MIME type for the Apache server, you would add the following line to your Apache configuration file:

```
AddType application/vnd.adobe.air-application-installer-
package+zip .air
```

Check the documentation for your web server for specific instructions on how to set the MIME type.

At this point, you have all of the basic knowledge of how to develop, test, and deploy AIR applications. You should now be ready to begin to learn more about the AIR APIs and how to build more full-featured and advanced applications.

CHAPTER 3

Working with JavaScript and HTML Within Adobe AIR

This chapter provides an overview of the HTML and JavaScript environments within Adobe AIR. It discusses:

- The use of the open source WebKit HTML-rendering engine within Adobe AIR
- JavaScript functionality within Adobe AIR
- Security Model
- Working with Adobe AIR, Flash Player and ActionScript APIs from JavaScript
- Troubleshooting AIR Applications written with HTML and JavaScript

Once you have completed this chapter, you should have a solid understanding of the HTML and JavaScript environments within Adobe AIR. You should also understand how to work with AIR and ActionScript APIs within HTML and JavaScript-based applications.

WebKit Within Adobe AIR

Adobe AIR leverages the open source WebKit-rendering engine to add support for rendering HTML content to the runtime.

In addition to HTML rendering, WebKit also provides support for associated web technologies, such as (but not limited to):

- JavaScript
- XMLHttpRequest
- CSS
- XHTML
- W3C DOM Level 2 support

Essentially, Adobe AIR has a full HTML rendering engine, and includes support for all of the same technologies that can be used when developing web applications and content targeting the web browser. Developers can build full-featured AIR applications that leverage these technologies.

NOTE

You can find more information on the WebKit project at: *http://www.webkit.org*.

Why WebKit?

Adobe spent a considerable amount of time researching which HTML engine to use within Adobe AIR and used a number of criteria that ultimately led them to settle on WebKit.

Open project

Adobe knew from the very beginning that it did not want to create and maintain its own HTML rendering engine. Not only would this be an immense amount of work, but it would also make it difficult for developers, who would then have to become familiar with all of the quirks of yet another HTML engine.

WebKit provides Adobe AIR with a full-featured HTML engine that is under continuous development by a robust development community that includes individual developers as well as large companies such as Nokia and Apple. This allows Adobe

to focus on bug fixes and features, and also means that Adobe can actively contribute back to WebKit, while also taking advantage of the contributions made by other members of the WebKit community.

Proven technology that web developers know

As discussed earlier, one of the biggest problems with complex web application development is ensuring that content works consistently across browsers. While something may work perfectly in Firefox on the Mac, it may completely fail in Internet Explorer on Windows. Because of this, testing and debugging browser-based content can be a nightmare for developers.

Adobe wanted to ensure that developers were already familiar with the HTML engine used within Adobe AIR so they did not have to learn all of the quirks and bugs of a new engine. Since Safari (which is built on top of WebKit) is the default browser for Mac OS X (and is also available on Windows), developers should be familiar with developing for WebKit.

Minimum effect on AIR runtime size

The size of Adobe AIR is approximately 11 MB on Windows and approximately 16 MB on MAC. The WebKit code base was well written and organized and has had a minimal impact on the final AIR runtime size.

Proven ability to run on mobile devices

While the first release of Adobe AIR runs only on personal computers, the long-term vision is to extend Adobe AIR from the desktop to cell phones and other devices. WebKit has a proven ability to run on such devices and has been ported to cell phones by both Nokia and Apple.

WebKit version used within Adobe AIR

The version of WebKit included in Adobe AIR 1.0 is based off the WebKit mainline version 523.15.

Some changes were applied to this version in order to support AIR's own rendering pipeline and enforce some security restrictions (please check the "AIR Implementation of Functionality" and the "Security" subchapters to find more details about the differences).

The User Agent reported when running in Adobe AIR in Windows is:

```
Mozilla/5.0 (Windows; U; en) AppleWebKit/420+ (KHTML,
like Gecko) AdobeAIR/1.0
```

and on Mac:

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en) AppleWeb
Kit/420+ (KHTML, like Gecko) AdobeAIR/1.0
```

Overall, developers should expect code running in Safari 3 to also work in Adobe AIR, with the exceptions of the differences discussed in this chapter.

JavaScript within Adobe AIR

Adobe AIR has full support for JavaScript within HTML content. JavaScript 1.5, which corresponds to ECMA-262 is supported.

The JavaScript engine is implemented via WebKit, and works the same as it does within WebKit-based browsers. In addition to having access to the HTML DOM, JavaScript can also access AIR and Flash Player APIs directly via the `window.runtime` property. This will be discussed in more detail later.

NOTE

For an in-depth introduction and discussion of JavaScript, check out *JavaScript: the Definitive Guide*: 5th Edition, published by O'Reilly:

http://www.oreilly.com/catalog/jscript5/

AIR Implementation of Functionality

HTML and JavaScript functionality is consistent with that found in other WebKit-based projects and browsers, such as Apple's Safari browser. When exploring documentation on HTML engine / browser functionality, you can use references to the Safari 3 browser as an indicator of the functionality available within the HTML environment within AIR.

However, because the HTML engine is running within Adobe AIR, and not a browser, there are a few differences that are useful to understand before beginning development with HTML and JavaScript within Adobe AIR.

URI Schemes

Working with Universal Resource Identifiers (URIs) within HTML content in AIR applications is largely the same as working with URIs within the browser. This section gives a quick overview of working with URIs within HTML content in AIR applications, and introduces some new URIs made available by the runtime.

Supported URI schemes

Adobe AIR provides support for the most common URI schemes available within the browser

Table 3-1. Supported URI schemes

Scheme	Description	Example
http://	URI that points to a resource accessed via the standard HTTP protocol.	*http://www.adobe.com*
https://	URI that points to a resource accessed via a protocol encrypted with SSL/TLS.	*https://secure.example.com*
file://	URI that points to a resource on the local or a networked file system.	file:///c:/Test/test.txt
mailto:	URI that opens the default email application.	mailto:*john.doe@example.com*

Unsupported URI schemes

The `feed://` and `data://` URI schemes are not supported by Adobe AIR 1.0, and there is only partial support for the `ftp://` scheme.

Finally, the `javascript:` URI scheme is not supported within applications running within the Adobe AIR application sandbox. Please check the Security model section later is this chapter for more details.

AIR URI Schemes

Adobe AIR also provides a number of additional URIs that makes it easy to reference files and content within specific areas of the users system.

Table 3-2. Adobe AIR URI schemes

URI	Description	Example
app:/	Provides a reference to the root content directory of the application. This should be used when referencing content included within the AIR file.	app:/images
app-storage:/	Provides a reference to an application-specific storage area on the user's system. This area is useful for storing user-specific application settings and content.	app-storage:/settings/pref.xml

> **NOTE**
>
> The AIR-specific URIs take only a single slash, versus two slashes in the other URIs.

Within HTML content, these URI schemes can be used anywhere within HTML and JavaScript content where regular HTTP URIs are used.

Relative URLs

You're not restricted to using just absolute URLs within AIR applications. You can also use relative URLs, but it is important to remember that relative URLs within AIR applications are relative to the application, and not to a server (as they would be when doing traditional browser-based client/server development).

Relative URLs will be relative to the root of the application, and will resolve to the app:/ URI.

For example:

```
<img src="foo/image.png" />
```

will resolve to:

```
<img src="app:/foo/image.png" />
```

You should keep this in mind when moving web and browser-based content and code into an AIR application.

Cookies

Adobe AIR has full support for setting and getting cookies from HTML-based content in remote sandboxes (content loaded from http:// and https:// sources) that is bound to a specific domain. Content loaded from the installed directory of the application (referenced via app:/ scheme) cannot use cookies (the `document.cookie` property).

Cookie support is implemented via the operating system's networking stack. This means that AIR applications can share cookies set by any browser or application that also leverage the operating system stack.

For example, AIR applications can share cookies set through Internet Explorer on Windows, and Safari on Mac, as they both also use the operating system's cookie storage functionality. Firefox implements its own cookie storage and thus cookies set within Firefox cannot be shared with AIR applications

NOTE

In addition to cookies, AIR applications have a number of other APIs that can be used to persist data, including the file API, as well as the embedded database API.

Windowing

Windows

You can create new windows via JavaScript just as you can within the browser.

```
myWindow = window.open("Window.html", "myWindow",
"height=400,width=400");
```

However, the runtime property that provides access to AIR and Flash Player APIs is not automatically available within the new window. In order to make it available, you must explicitly place it within the scope of the new window like so:

```
window.runtime = window.opener.runtime;
```

You can also create native windows using apis provided by AIR. 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 allow you to define the user interface.

```
var initOptions = new runtime.flash.display.NativeWindow
InitOptions();var bounds = new runtime.flash.geom.Rectangle
(10, 10, 600, 400); var myHtml= runtime.flash.html.HTMLLoader.
createRootWindow(true, initOptions, true,bounds);
var urlReq = new runtime.flash.net.URLRequest("http://www.
example.com"); myHtml.load(urlReq);
```

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`.

Dialogs

The `alert`, `confirm` and `prompt` HTML dialogs are also supported within Adobe AIR.

In addition, the file-browse dialog created via:

```
<input type="file" />
```

is also supported in Adobe AIR 1.0.

XMLHttpRequest and Ajax

The `XMLHttpRequest` object, which enables the use of Ajax techniques for sending and loading data, is completely supported within AIR applications.

One advantage to developing Ajax applications within Adobe AIR versus the browser is that because you have a consistent runtime to target across operating systems, you do not have to worry about cross-browser, platform inconsistencies in how the API is implemented.

The primary benefit of this is that you have to write only one version of the code.

Here is a simple example of an `XMLHttpRequest` object call within an AIR application that works regardless of which operating system the application is running on:

```
<script type="text/javascript">
    var xmlhttp;
    function appLoad()
    {
        //replace with URL to resource being loaded
        var url = "http://www.mikechambers.com/blog/";
        xmlhttp = new XMLHttpRequest();
        xmlhttp.open("GET", url,true);

        xmlhttp.onreadystatechange=function(){
            if (xmlhttp.readyState==4)
            {
                runtime.trace(xmlhttp.responseText);
            }
        }
```

```
            xmlhttp.send(null)
        }
    </script>
```

When called, this function uses the `XMLHttpRequest` object to load the specified URL and prints its contents out to the command line. The main thing to note in this example is that because the runtime is known and it is consistent across operating systems, you do not have to detect the existence of, or in the implementation of `XMLHttpRequest` as you would when deploying in the browser.

Both synchronous and asynchronous `XMLHttpRequest` calls are supported, as is loading data across domains.

Canvas object

The canvas element is supported by WebKit and also by AIR. It defines APIs for drawing geometric shapes, but in most respects it behaves like an image.

The result of a drop operation when dragging images can be among other types (e.g.: a file reference) a canvas element; it can be displayed by appending the element to the DOM.

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.

You can use the methods of the `clipboardData` object to read or write clipboard data.

For more details on how to do these operations, please check the cookbook chapter.

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 more details on how to work with drag and drop, please check the cookbook chapter.

Supported plug-ins

Adobe AIR does not support other plug-ins except Acrobat or Adobe Reader 8.1+ for displaying PDF content and Flash Player plug-in for displaying SWF content.

In order to load Flash content into HTML, you use the same embed element syntax that you use when embedding Flash content within a browser page.

```
<embed
    src="content.swf"
    quality="high"
    bgcolor="#FFFFFF"
    width="400"
    height="200"
    name="content"
    swLiveConnect="true"
    align="bottom"
    allowScriptAccess="always"
 type="application/x-shockwave-flash"  pluginspage="http://
www.macromedia.com/go/getflashplayer" />
```

NOTE

You can load both relative and absolute URLs, with relative URLs being relative to the application install directory.

You can use the same techniques that you use in the browser, such as the `ExternalInterace` API, to facilitate Flash to HTML and HTML to Flash communication within HTML content.

For the SWF content, the Flash Player plug-in is built into AIR and doesn't need to use an external plug-in.

Unsupported functionality

The `window.print()` method is not supported within Adobe AIR 1.0. There are other methods available via the exposed AIR APIs that give you access to printing within the runtime, but these might feel different from what is available at the browser level.

In addition, support for Scalable Vector Graphics (SVG) is not included in AIR 1.0.

Security Model

This section discusses a number of differences in the security model implementations when running content within Adobe AIR, versus running it in the browser.

Why a different security model?

Adobe AIR enables Ajax and Flash/Flex developers to use their existing skills to build and deploy desktop applications. Although these applications are built using web technologies, the key thing to keep in mind is that the end result is targeted for running on desktop, and thus the security model for AIR is much closer to that of a desktop application than of a web application.

A desktop application has high privileges compared to a web application as it is installed by the user on a specific machine, implying a degree of trust that is greater than that of arbitrary web content. Unfortunately there are design and implementation patterns common to web applications that can be danger-

ous when combined with the local system access or other AIR APIs available.

Adobe AIR Sandboxes

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

Application sandbox

The application sandbox provides access to all Adobe AIR APIs, including those that provide access to the user's system.

When an application is installed, all files included within an AIR installer file are copied onto the user's computer into an application directory. Developers can reference this directory in code through the `app:/` URI scheme. All files within the application directory tree are assigned to the application sandbox when the application is run.

When running within the application sandbox, some potentially dangerous patterns that are allowed in the browser, are restricted. The things that are disabled revolve around the importing of remote JavaScript content and the dynamic execution/evaluation of JavaScript strings.

> **NOTE**
>
> Please see the Adobe AIR documentation for a complete list of what is and is not allowed within the application sandbox.

Non-application sandbox

The non-application sandbox contains all content that is not loaded directly into the application sandbox. This may include both local and remote content loaded into the application at

runtime. Such content does not have direct access to AIR APIs and obeys the same rules that it would have to obey in the browser when loaded from the same location (for example HTML from a remote domain behaves like it would behave in the browser). Because this content doesn't gain direct access to AIR APIs it can call into any code evaluation technique that works in browser, such as eval, loading remote scripts, etc.

Developing within the Sandboxes

Access to the runtime environment and AIR APIs are only available to HTML and JavaScript running within the application sandbox. At the same time, dynamic evaluation and execution of JavaScript, in its various forms, is largely restricted within the application sandbox. These restrictions are in place whether or not your application actually loads information directly from a remote server.

Developing within the application sandbox

Here's an overview of the differences you may run into when coding for application sandbox.

In general, code evaluation restrictions do not apply while code is initializing prior to the onload event. Strings can be turned into executable code via the use of `eval` or the Function constructor, and attributes are turned into actual event handlers such as:

```
<input type="button" onclick="doClickBtn();"/>
```

NOTE

If code execution is prevented due to sandbox security restrictions, the following JavaScript error will be thrown : "Adobe AIR runtime security violation for JavaScript code in the application security sandbox."

Loading remote JavaScript files. HTML files running in application sandbox (loaded from the installed directory of the appli-

cation) can only import JavaScript files located in the same installed directory via

```
<script src="file.js" type="text/javascript"></script>
```

If you must load remote JavaScript, you must either install the file(s) with the application or load it into the non-application sandbox.

Cross-domain content loading. Code running in the application sandbox can load data from any remote domain via `XMLHttpRequest`. There is no cross domain policy enforced for the code loaded from the application install directory, as there is no domain associated with this code (it uses the AIR specific app:/ scheme to load the files from the installation directory). However, the tradeoff is that the content loaded via XHR can only be used as data; it cannot be transformed into executable code (with one exception explained below).

APIs which allow dynamically generated code to execute are prohibited after the onload event. The table below shows which APIs are restricted when running within the application sandbox:

eval()	Prohibited after onload. After the parsing time, you cannot use `eval()` to transform strings (such as those imported via XHR) into JavaScript code. The exception from this rule is `eval()` used with a string parameter of type JSON — pure JSON strings can be transformed into actual objects, although JSON code that contains call back functions are not supported.
Function constructor	Prohibited after onload.
setTimeout()	Prohibited after onload when using string parameters.
setInterval()	Prohibited after onload when using string parameters.
javascript:URLs	Prohibited.
document.write()	Prohibited after onload.

innerHTML	Code attributes on elements inserted via innerHTML or outerHTML are not transformed into executable code, such as container.innerHTML = '<input type="button" onclick="doClickBtn();"/>';
script.src	Setting the src tag of a script element to load remote data / content is prohibited.
XMLHttpRequest	Synchronous calls outside of the application sandbox prohibited prior to onload. Asynchronous calls initiated in onload always finish after onload, and code evaluation restrictions will be applied to loaded content.

Developing within the non-application sandbox

Everything that is not loaded from the application install directory is loaded into the non-application sandbox.

Code in the non-application sandbox does not have direct access to AIR APIs, and thus does not have the JavaScript code evaluation restrictions applied to it.

Creating a non-application sandbox. When a file loaded from the application sandbox loads a file from a non-application sandbox (either via the `iframe` or `frame` tags, or from opening a new window) the loaded content will be placed within the non-application security sandbox, with no direct access to AIR apis.

Adobe AIR also provides a way to load application content into the non-application sandbox. This is done via the `sandboxRoot` and `documentRoot` properties on the `iframe` and `frame` tags.

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

This example maps content installed in the "sandbox" subdirectory of the application to run in the remote sandbox bound to www.example.com (*http://www.example.com*) domain.

The `sandboxRoot` property specifies the URL to use for determining the sandbox and domain in which to place the `iframe` content. The `file:`, `http:`, or `https:` URL schemes must be used.

The `documentRoot` property specifies the directory from which to load the `iframe` content. The `file:`, `app:`, or `app-storage:` URL schemes must be used.

Differences on code running under non-application sandbox.

XMLHttpRequest
: Code in the non-application sandbox is bound by a same-origin policy and, by default, cannot do cross-domain data loading via XMLHttpRequest from any remote URL.

  However, this behavior can be overwritten by setting the `allowCrossDomainXHR` attribute to true on the `iframe` or frame element loading the `content`.

window.open
: The `window.open()` api only works if it's a direct result of user interaction (such as mouse click or keypress).

Window term
: The title of a window opened by non-application content is prefixed with the application title (to prevent windows opened by remote content from spoofing windows opened by the application).

Scripting between sandboxes

The runtime lets you create an interface called Sandbox Bridge between content in a non-application sandbox and its parent document in the application sandbox. This is a bi-directional serialization API designed to allow domains/sandboxes that otherwise cannot trust each other entirely to interact.

Using the Sandbox Bridge. It is possible for the developer to provide a bridge between application sandboxed content (parent) that loads content (child) into the non-application sandbox. Essentially, the parent can explicitly allow the child to access specific properties and methods that it defines.

The application sandbox content (parent) can set up a property called `parentSandboxBridge` on the `frame` or `iframe` tag, and expose functions by attaching them to this property. These functions can be accessed by the non-application content inside the iframe or frame.

In a similar way, the non-application sandbox—the child—can set up a property called `childSandboxBridge` on its window object and expose functions to the application sandbox.

In order for a script in a child document to access a function attached to the `parentSandboxBridge` property, the bridge must be set up before the script is run. A new AIR-specific event called `dominitialize` is fired when the 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. Here is an example:

parent.html

```
<html>
<head>
<script type="text/javascript">
    var bridgeInterface = {};
    bridgeInterface.testProperty = "Bridge engaged";
    bridgeInterface.testFunction = function() { alert
    ('testFunction') };

    function setupBridge(){
         document.getElementById("sandbox").contentWindow.
         parentSandboxBridge =bridgeInterface;
}
</script>
</head>
<body>
<iframe id="sandbox"
    src="http://www.example.com/airapp/child.html"
    documentRoot="app:/"
    sandboxRoot="http://www.example.com/airapp/"
    ondominitialize="setupBridge ()">
</iframe>
</body>
</html>
```

child.html

```
<html>
<head>
<script type="text/javascript">
alert(window.parentSandboxBridge.testProperty);
</script>
</head>
<body></body>
</html>
```

As a general rule, you should not directly expose AIR apis to content running in the non-application sandbox, but rather should encapsulate the access via a function. This ensures that only specific AIR APIs can be called, and when they are called, are only executed in the manner that you allow.

Table 3-3. Sandboxes capabilities overview

Capability	Application sandbox	Non-application sandbox
Direct access to AIR APIs	Yes	No
Access to application sandbox functions that use AIR APIs via the SandboxBridge	N/A	Yes
Loading remote JavaScript files	No	Yes
Can, by default, execute cross-domain requests (XHR)	Yes	No
Supports transforming strings into executable code after the onload event	No	Yes
Ajax frameworks work by default	No[a]	Yes

[a] Frameworks must add support for Adobe AIR.

For more details on working with the Sandbox Bridge see: *http://www.adobe.com/devnet/air/ajax/quickstart/sandbox_bridge.html*

Using JavaScript Frameworks

Because of the differences of the security models between content running within the browser, and content running within Adobe AIR, most JavaScript frameworks must explicitly add support for Adobe AIR in order for them to running correctly within an Adobe AIR application.

At the time that the book was authored, all major JavaScript frameworks have added (or are adding) support for Adobe AIR.

JavaScript Frameworks and Libraries supporting AIR application sandbox

Dojo Toolkit 1.1.0 Beta
http://www.dojotoolkit.org/air

Ext JS 2.0.2 with Adobe AIR Adapter
http://extjs.com/download

jQuery 1.2.3
http://docs.jquery.com/Downloading_jQuery

YUI 2.5.1
http://developer.yahoo.com/yui/

MooTools 1.11
http://www.mootools.net/download

FCKeditor 2.6.0 Beta
http://www.fckeditor.net/

MochiKit 1.3.1
http://mochikit.com/download.html

Spry Prerelease 1.6.1
http://labs.adobe.com/technologies/spry/

Ajax frameworks and libraries compatible with AIR application sandbox at the time of authoring this book

For a complete and up to date list of frameworks that support Adobe AIR please check the AIR product page *http://*

www.adobe.com/products/air/ or the AIR Developer Center for Ajax: *http://www.adobe.com/devnet/air/ajax/*.

Accessing AIR APIs from JavaScript

In addition to the standard JavaScript and HTML DOM APIs, JavaScript code running within the application sandbox in an AIR application can also take advantage of APIs provided by the runtime, as well as Flash Player APIs and even ActionScript 3 libraries. This greatly extends the capabilities of JavaScript over the APIs available in the browser, and includes functionality such as:

- Playing sounds
- Manipulating images and bitmaps
- Reading and writing files to and from the local file system
- Reading and writing to and from an encrypted local store
- Access to a relational local database
- Creating, controlling and manipulating native windows
- Creating and working with native menus
- Making direct socket connections (both binary and text based)
- Network detection
- Accessing the clipboard
- Dragging data between AIR applications and OS or another application
- File extension registration and running at startup
- Support for dock and tray icons

NOTE

You can find examples of how to leverage these features in the cookbook section.

This section discusses how to leverage AIR and Flash Player APIs from JavaScript, as well as how to load and leverage compiled ActionScript libraries from within JavaScript.

The JavaScript environment and its relationship with AIR

The following diagram shows the relationship between the two environments.

Only a single native window is shown but an AIR application can contain multiple windows. Also, 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 runtime environment through the `runtime`, `nativeWindow`, and `htmlLoader` properties of window.

The `runtime` property provides access to AIR API classes; it allows you to create new AIR objects as well as access static members.

The `nativeWindow` gives you access to the current instance of the `NativeWindow` that controls the current application window.

The `htmlLoader` gives you access to the current instance of the `HTMLLoader` that controls how content is loaded and rendered.

> **NOTE**
>
> Only content that is part of the application sandbox has access to these three 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.

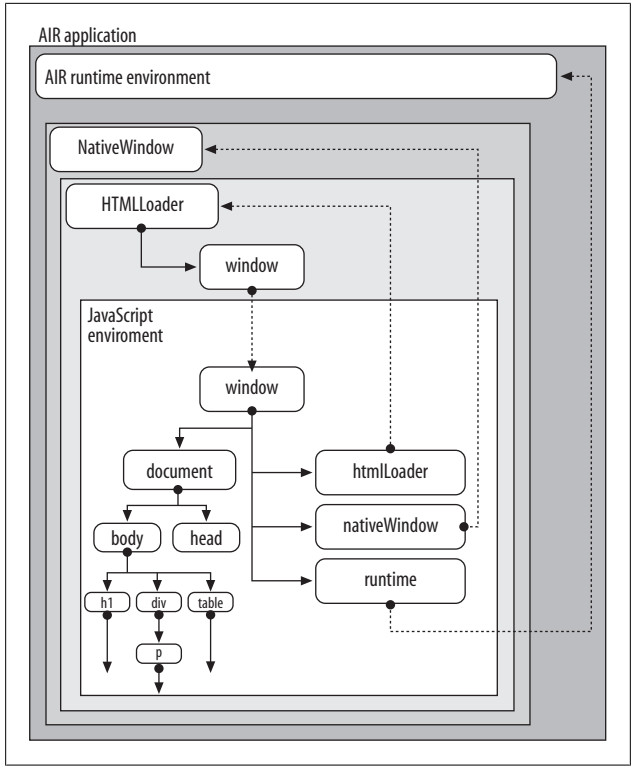

Figure 3-1. JavaScript environment in an AIR Application

There is also another important object instance available in any AIR application that is not shown in the diagram. 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;
```

Accessing AIR and Flash Player APIs

Most AIR and Flash Player APIs are contained within packages (similar to how many Ajax frameworks leverage namespaces and packages). This helps organize the APIs, and also reduces the possibility of naming conflicts. When accessing AIR and Flash Player APIs directly from JavaScript, you must do so via their complete package path and name.

As discussed earlier, all AIR and Flash Player APIs are made available via the `window.runtime` property. The runtime property is at the root of the runtime environment, and all APIs are relative to this root.

For example, to access an API which is not contained within a package, such as `trace` you reference it directly from the runtime property, like so:

```
window.runtime.trace("foo");
```

However, if you want to access an API that is contained within a package, you must prepend the package path to the API. For example, to access the amount of memory currently used by the application, you can call the `totalMemory` Flash Player property that is in the `flash.system.System` class. To call this API from JavaScript:

```
var mem = window.runtime.flash.system.System.totalMemory;
```

This also applies when creating new instances of an API class from within JavaScript:

```
var file = new window.runtime.flash.filesystem.File();
```

This code creates a new `File` instance that can be used to work with the file system.

Here is a complete example that shows how to write a file named *output.txt* to the user's desktop:

```
//call a static property
var desktop = window.runtime.flash.filesystem.File.desktop
```

```
Directory;

//call a function on an instance of a class
var file = desktop.resolvePath("output.txt");

//create a new instance of a class using new
var fileStream = new window.runtime.flash.filesystem.
FileStream();

    //call a function, passing arguments
    fileStream.open(file, window.runtime.flash.filesystem.
                    FileMode.WRITE);

    fileStream.writeUTFBytes("Hello World");
    fileStream.close();
```

Don't worry too much about what the code is doing in this example, but rather focus on how the AIR APIs are called from JavaScript.

This allows you to leverage virtually any AIR or Flash Player API from within JavaScript.

By remembering how to use the package structure to call APIs, you can leverage all AIR, Flash Player and ActionScript APIs even if JavaScript-specific documentation is not provided.

Working with AIR and Flash Player Events

Many of the AIR and Flash Player APIs make extensive use of events. Event handling in ActionScript-based APIs is based on the W3C DOM Level 3 event model. This is similar to the W3C DOM Level 2 event model available within JavaScript, but is very different from the callback mechanism often deployed in JavaScript.

In order to be notified when an event from an AIR or Flash Player API occurs, you must register to listen for it. The best way to understand this is to look at an example. The following example registers for a `NETWORK_CHANGE` event that is broadcast by the `NativeApplication` instance:

```
function onNetworkChange(event)
{
    runtime.trace("Network status changed");
```

```
        }

        function onAppLoad()
        {
window.runtime.flash.desktop.NativeApplication.
nativeApplication.addEventListener(
                    window.runtime.flash.events.Event.
                    NETWORK_CHANGE,
 onNetworkChange);
        }
```

As you can see from the example, you register for events broadcast by a class instance by calling the `addEventListener` function on the class instance. This API requires two arguments.

The first argument is the event name of the event being broadcast. For all AIR and Flash Player APIs, there will be a constant for the event name, which you can find in the documentation.

The second argument is a reference to the function that will handle the event. In this case, the function is named `onNetworkChange`. Looking at the function, you can see that it is passed an argument with information about the event. Again, all AIR and Flash Player APIs will pass an object to the event handler function, which provides information about the event. You can find the exact type of event object passed to the handler, and the information it provides, by referencing the API documentation.

Using AIRAliases.js file

As the previous examples show, being able to leverage AIR and Flash Player APIs from directly within JavaScript can be very powerful. However, because you must reference the APIs via the runtime property and the complete API package path, it can lead to very verbose code.

In order to make it easier to use some of the more common AIR and Flash Player APIs from within JavaScript, Adobe has created a JavaScript include file, named *AIRAliases.js*. This file, which can be found in the frameworks directory of the SDK, provides aliases for commonly used APIs to make them more convenient to use from within JavaScript.

To use the aliases file, copy it from the SDK to your application directory (make sure to also package it in your AIR file). You then include it within your application using the script tag in each HTML document that you want to leverage the aliases in.

For example, let's look at the earlier example that writes a file to the desktop, but uses the JavaScript aliases provided in the *AIRAliases.js* file instead of typing out the complete package paths:

```
<script src="AIRAliases.js" type="text/javascript"></script>

<script type="text/javascript">
    function writeFile()
    {
        var desktop = air.File.desktopDirectory;
        var file = desktop.resolvePath("output.txt");
        var fileStream = new air.FileStream();
            fileStream.open(file, air.FileMode.WRITE);
            fileStream.writeUTFBytes("Hello World");
            fileStream.close();
    }
</script>
```

First, notice that the code is much less verbose. This is because instead of having to reference APIs via `window.runtime` and then the complete package path, we can use the aliases within the include file.

For example, this reduces:

```
var desktop = window.runtime.flash.filesystem.File.
desktopDirectory;
```

to:

```
var desktop = air.File.desktopDirectory;
```

Second, the AIR and Flash Player APIs are placed in a namespace called `air`. If you open the AIRAliases.js file, you can see how the aliases actually work. For example, here is the code that sets up the File API aliases:

```
var air; if (!air) air = {};

// file
air.File = window.runtime.flash.filesystem.File;
```

```
air.FileStream = window.runtime.flash.filesystem.FileStream;
air.FileMode = window.runtime.flash.filesystem.FileMode;
```

To see a complete list of APIs included, open up the *AIRAliases.js* file with a text editor. While not all APIs are included, you can easily add additional APIs by following the existing examples in the file.

Leveraging Compiled ActionScript Libraries

Not only can AIR applications leverage Flash Player APIs directly from JavaScript, they can also access compiled ActionScript 3 libraries from within JavaScript.

In addition to loading external JavaScript files, the HTML script tag within an AIR application also has support for loading compiled ActionScript 3 libraries and providing access to the ActionScript classes included within the file. Once the SWF is loaded, the APIs can be referenced in the same manner as the AIR and Flash Player APIs are referenced via the API package path and API name.

NOTE

This technique works only with ActionScript 3 libraries.

Let's look at an example. Included in the Adobe AIR SDK is a SWF that contains the ActionScript 3 Adobe AIR service connectivity API. While this example won't show how to use that API, it will show how to access those APIs from within JavaScript.

NOTE

An example of how to use the Service Connectivity API is provided in the cookbook section.

In order for the example to work, you must copy the *frame works/servicemonitor.swf* file from the AIR SDK to your application directory.

Any classes and APIs available within the compiled SWF will be made available via the `window.runtime` property. The API we want to reference is in a class named `ServiceMonitor` in the `air.net` package.

Here's the code:

```
<script src="servicemonitor.swf" type="application/x-
shockwave-flash"></script>

<script type="text/javascript">
    function onAppLoad()
    {
        var monitor = new runtime.air.net.ServiceMonitor();
    }
</script>
```

If *AIRAliases.js* is included in the page you can use the short version:

```
var monitor = new air.ServiceMonitor();
```

This is a very simple example that shows how to load compiled ActionScript libraries, and then access them from JavaScript. In this case, we include the *servicemonitor.swf* file via the HTML script tag using the special type "`application/x-shock wave-flash`". This file contains the compiled ActionScript 3 APIs for the `air.net.ServiceMonitor` class.

Using this technique allows you to leverage third party ActionScript APIs from within AIR applications via JavaScript.

NOTE

Most ActionScript 3 libraries are distributed as zip-based SWC files. In order to use the libraries within JavaScript, change the extension of the library from SWC to ZIP, unzip them using a zip program, and then remove the library's SWF file contained within the SWC.

You can then include the SWF within your application in the same manner as demonstrated earlier.

Troubleshooting AIR Applications

The process of developing an AIR application is much the same as that of developing an HTML-based web application.

The same thing is true when it comes to debugging an AIR application. One difference is that the JavaScript error messages go to the console (when running the application with AIR Debug Launcher) and not in a separate window as you might be used to in the browser world.

This section covers some of the new messages you might come across that are introduced by the runtime and also gives you a quick overview of an AIR SDK tool that can be used to make your life easier when working with HTML/JavaScript.

New JavaScript error messages

Security violation for Javascript code

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

Referencing a JavaScript object no longer available

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.

```
// In this example the event listener for an uncaughtScript
Exception event is removed when unload event fires
window.onunload = cleanup;
window.htmlLoader.addEventListener('uncaughtScriptException',
uncaughtScriptException);
function cleanup()
{
    window.htmlLoader.removeEventListener('uncaught
ScriptException',
 uncaughtScriptExceptionHandler);
}
```

Missing event listeners for error events

Exceptions, rather than events, are the primary mechanism for error handling in the runtime classes. However, because exceptions don't work for asynchronous operations, the runtime dispatches an error event object.

If you do not create a listener for the error event (such as when loading files asynchronously), the AIR Debug Launcher presents a dialog box with information about the error.

Most error events are based on the `ErrorEvent` class, and have a property named `text` that is used to store a descriptive error message.

An error event does not cause an application to stop executing. It manifests only as a dialog box when launched via ADL and is not presented when running in an installed application.

AIR Introspector

The Adobe AIR SDK includes a JavaScript based tool called AIR Introspector that makes it easier to develop HTML based

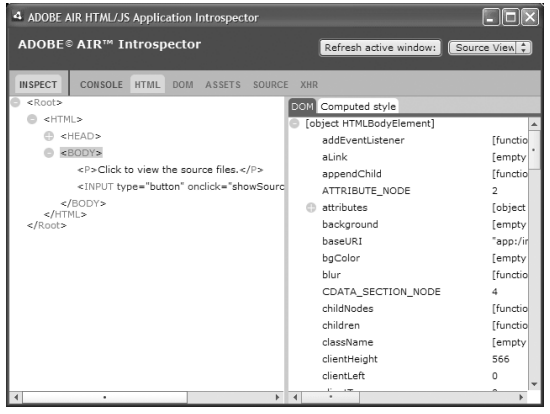

Figure 3-2. AIR Introspector window

applications for Adobe AIR. Tje tool can be used to introspect the content (such as the DOM) of HTML content running within an AIR application.

In order to use the AIR Introspector with your application, copy the *AIRIntrospector.js* file from the *frameworks* directory of the SDK into to your application project directory and then load the file into your application via the script tag:

```
<script src="AIRIntrospector.js" type="text/javascript">
</script>
```

The tool provides a number of features and functionality:

- A tool that allows you to point to a user interface element in the application and see its markup and DOM properties.
- Ability to edit the attributes and text nodes for DOM elements.
- A console for introspecting, and adjusting objects references, properties and code. This includes the ability to execute JavaScript code.
- View of DOM properties and functions.

- Lists of links, CSS styles, images, and JavaScript files loaded into the application.
- Ability to view the initial HTML source and the current markup source for the user interface.
- A viewer for `XMLHttpRequest` objects and their properties, including `responseText` and `responseXML` properties (when available).
- Ability to search for matching text in the source code and files.

At this point, you should have a good understanding of the HTML and JavaScript environments within Adobe AIR, as well as how to leverage AIR, Flash Player and third-party ActionScript 3 APIs directly from JavaScript.

The rest of the book will use this knowledge to show how to accomplish specific tasks from HTML and JavaScript applications running within Adobe AIR.

CHAPTER 4

Adobe AIR Mini-Cookbook

This chapter provides solutions to common tasks when developing Adobe AIR applications. The solutions in this chapter illustrate many concepts used in AIR application development, and provide working HTML and JavaScript code that you can leverage within your application.

NOTE

All examples in this chapter assume that you are using the *AIRAliases.js* file.

Application Deployment

Deploy from a Web Page

Problem

You have finished your application, have signed and packaged it, and want to distribute it via a web page.

Solution

Adobe AIR applications can be easily distributed from a web page using the badge installer included with the SDK.

Discussion

Adobe AIR application files are largely self-contained entities, and are ready for distribution once they are signed and packaged. The resultant file will have an *.air* extension. That application file can be distributed via email, CD-ROM, or other traditional forms; however, installing an *.air* file requires that Adobe AIR is already present on the target machine. Alternatively, a web-page-based "badge installer" can streamline installation by detecting the runtime and installing it if necessary before installing your application.

Though you can customize it in a number of different ways, a sample badge installer is included with the Adobe AIR SDK. The badge takes the form of a small 217×180 area, which is ideal for a blog sidebar or other constrained spaces. The default badge installer runs as a Flash 9.0.115 (Flash Update 3) component in the browser. The Flash source file (FLA) is also included with the SDK for additional customization.

> **NOTE**
>
> You can find the files for the sample badge installer in the *samples/badge* directory of the SDK.

Deploying with the badge installer requires four files: *badge.swf*, *default_badge.html*, *AC_RunActiveContent.js*, and your AIR application.

Even though the badge installer does appear as Flash content on a web page, you do not need to have any Flash knowledge or software such as Adobe Flash CS3. The badge installer was prebuilt with a number of configurable options that you can set from within the containing HTML page. On line 59 of the *default_badge.html* file, you will see the `flashvars` parameter, which is assigned the various initialization properties that are specific to your application. This parameter takes the form of a query string, and has the options outlined in Table 4-1.

Table 4-1. Badge Installer flashvars options

Parameter	Description
`appname`	The name of the application, displayed by the badge installer.
`appurl`	Required. The URL of the Adobe AIR file to be downloaded. You must use an absolute, not a relative, URL.
`airversion`	Required. For the 1.0 version of the runtime, set this to `1.0`.
`imageurl`	Optional. The URL of the image to display in the badge.
`buttoncolor`	Optional. The color of the download button (specified as a hex value, such as FFCC00).
`messagecolor`	Optional. The color of the text message displayed below the button (specified as a hex value, such as FFCC00).

Here is an HTML page that displays the badge installer to install an AIR application, as well as the AIR runtime if necessary:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
<head>

<title>Adobe AIR Application Installer Page</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />

<style type="text/css">
<!--
#AIRDownloadMessageTable {
    width: 217px;
    height: 180px;
    border: 1px solid #999;
    font-family: Verdana, Arial, Helvetica, sans-serif;
    font-size: 14px;
}

#AIRDownloadMessageRuntime {
    font-size: 12px;
    color: #333;
}
-->
</style>

<script language="JavaScript" type="text/javascript">
```

```
<!--
var requiredMajorVersion = 9;
var requiredMinorVersion = 0;
var requiredRevision = 115;
</sc

</head>
<body bgcolor="#ffffff">

<script src="AC_RunActiveContent.js" type="text/javascript">
</script>

<script language="JavaScript" type="text/javascript">
<!--
// Version check based upon the values entered above in
"Globals" var hasRequestedVersion = DetectFlashVer(
requiredMajorVersion, requiredMinorVersion,
requiredRevision );

// Check to see if the version meets the requirements
// for playback
if( hasReqestedVersion )
{

    AC_FL_RunContent(
          'codebase','http://fpdownload.macromedia.com/pub/
          shockwave/cabs/flash/swflash.cab',
         'width','217',
         'height','180',
         'id','badge',
         'align','middle',
         'src','badge',
         'quality','high',
         'bgcolor','#FFFFFF',
         'name','badge',
         'allowscriptaccess','all',
         'pluginspage','http://www.macromedia.com/
          go/getflashplayer',
          'flashvars','appname=My%20Application&appurl=
          myapp.air&airversion=
          1.0&imageurl=test.jpg',
          'movie','badge' ); //end AC code

} else {
    document.write('<table id="AIRDownloadMessageTable"><tr>
    <td>Download <a href="myapp.air">My Application</a>
    now.<br /><br /><span id="AIRDownloadMessageRuntime">
```

```
This
 application requires the <a href="');

    var platform = 'unknown';

    if( typeof( window.navigator.platform ) != undefined )
    {
         platform = window.navigator.platform.toLowerCase();
         if( platform.indexOf( 'win' ) != -1 )
         {
              platform = 'win';
         } else if( platform.indexOf( 'mac' ) != -1 ) {
              platform = 'mac';
         }
    }

    if( platform == 'win' )
    {
    document.write( 'http://airdownload.adobe.com/air/win/
    download/1.0/
AdobeAIRInstaller.exe' );
    } else if( platform == 'mac' ) {
         document.write( 'http://airdownload.adobe.com/air/
         mac/download/1.0/
AdobeAIR.dmg' );
    } else {
         document.write( 'http://www.adobe.com/go/getair/' );
    }

    document.write( '">Adobe&#174; AIR&#8482; runtime</a>.
</span></td></tr></table>' );
}
// -->
</script>

<noscript>
<table id="AIRDownloadMessageTable">
<tr>
    <td>
    Download <a href="myapp.air">My Application</a> now.<br />
<br /><span id="AIRDownloadMessageRuntime">This application
requires Adobe&#174; AIR&#8482; to be installed for
<a href="http://airdownload.adobe.com/air/mac/download/
1.0/AdobeAIR.dmg">Mac OS</a> or <a href="http://
airdownload.adobe.com/air/win/download/1.0/
AdobeAIRInstaller.exe">Windows</a>.</span>
    </td>
```

```
</tr>
</table>
</noscript>

</body>
</html>
```

Application Chrome

Add Custom Controls

Problem

You want to create custom window chrome for your application and you need the user to be able to close and minimize the application.

Solution

Use the `NativeWindow` class within Adobe AIR to integrate, minimize, and close button functionality.

Discussion

Although Adobe AIR allows developers to completely define and customize the application's window chrome, it is important to remember that when doing so, the application is responsible for every type of windowing event that might normally occur. This means the application must connect the various visual elements with their respective operating system events.

When deploying an application on Adobe AIR, the `window` object gets additional properties. Among those properties is `nativeWindow`, which is a reference to the native window that houses the current HTML document. Using the native window reference, the appropriate methods can be called to trigger the operating-system-specific event (or vice versa). In the case of being able to minimize the window, the application can use

`NativeWindow.minimize()`; it can use `NativeWindow.close()` in the case of closing it:

```
window.nativeWindow.minimize();
window.nativeWindow.close();
```

The `NativeWindow.close()` method does not necessarily terminate the application. If only one application window is available, the application will terminate. If multiple windows are available, they will close until only one window remains. Closing the last window terminates the application.

application.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/1.0">

    <id>com.adobe.demo.CustomControls</id>
    <name>Custom Controls</name>
    <version>1.0</version>
    <filename>Custom Controls</filename>
    <description>Adding Custom Window Controls</description>

    <initialWindow>

        <title>Custom Controls</title>
        <content>index.html</content>
        <systemChrome>none</systemChrome>
        <transparent>true</transparent>
        <visible>true</visible>
        <width>206</width>
        <height>206</height>

  </initialWindow>

</application>
```

index.html

```
<html>
<head>

<title>Custom Window Controls</title>

<style>
body {
```

```
   background-image: url( 'custom-chrome.gif' );
   font-family: Verdana, Geneva, Arial, Helvetica, sans-serif;
   font-size: 12px;
}

#closer {
    position: absolute;
    width: 70px;
    left: 68px;
    top: 105px;
}

#minimize {
    position: absolute;
    width: 70px;
    left: 68px;
    top: 75px;
}

textarea {
    position: absolute;
    left: 8px;
    right: 8px;
    bottom: 8px;
    top: 36px;
    border-color: #B3B3B3;
}

#title {
    position: absolute;
    font-weight: bold;
    color: #FFFFFF;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script>
function doClose()
{
    window.nativeWindow.close();
}

function doLoad()
{
    document.getElementById( "minimize" ).addEventListener(
    "click", doMinimize );
```

```
    document.getElementById( "closer" ).addEventListener(
    "click", doClose );
}

function doMinimize()
{
    window.nativeWindow.minimize();
}
</script>

</head>
<body onload="doLoad()">

<input id="minimize" type="button" value="Minimize" />
<input id="closer" type="button" value="Close" />

</body>
</html>
```

Windowing

Create a New Window

Problem

You need to display an additional widow into which additional content can be loaded.

Solution

Basic windows can be generated and maintained in a similar fashion as traditional HTML content using the `window.open()` method.

Discussion

The JavaScript `window.open()` method invokes a new window similar to the way it would when used in the browser. Content that gets loaded into the new window can come from a local file, or URL endpoint. Similar to windows created using JavaScript in the browser, there is finite control over the window

itself. The window properties that can be controlled are `width`, `height`, `scrollbars`, and `resizable`.

```
var login = window.open( 'login.html', 'login', 'width = 300,
height = 200' );
```

A native window is a better choice when additional control over the new window is required. Native windows expose virtually the entire functionality of the operating system, such as control over minimize and maximize functionality, always in front, full screen, and even removal of system chrome altogether. The choice between using `window.open()` and `Native Window` depends largely on the requirements for the window and the overall portability of the JavaScript source code.

NOTE

You also can use the `window.opener` property, which is commonly used in JavaScript to refer from a new window to the parent (creating) window.

```
<html>
<head>

<title>Creating a New Window</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}
</style>

<script type="text/javascript">
function doLoad()
{
    document.getElementById( 'window' ).addEventListener(
    'click', doWindow );
}

function doWindow()
{
```

```
    var login = window.open( 'login.html', null, 'width = 325,
    height = 145' );
}

function doLogin( email, pass )
{
    alert( 'Welcome: ' + email );
}
</script>

</head>
<body onLoad="doLoad();">

<input id="window" type="button" value="Login" />

</body>
</html>
```

Login.html

```
<html>
<head>

<title>Login</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}
</style>

<script>
function doLoad()
{
    document.getElementById( 'signIn' ).addEventListener(
    'click', doSignIn );
}

function doSignIn()
{
    var email = document.getElementById( 'email' ).value;
    var password = document.getElementById( 'password' )
    .value;

    window.opener.doLogin( email, password );
```

```
}
</script>

</head>
<body onLoad="doLoad();">

<table>
    <tr>
         <td>Email:</td>
         <td><input id="email" name="email" /></td>
    </tr>
    <tr>
         <td>Password:</td>
         <td><input id="password" name="password"
         type="password" /></td>
    </tr>
    <tr>
         <td colspan="2" align="right">
              <input id="signIn" type="button"
               value="Sign In" />
         </td>
    </tr>
</table>

</body>
</html>
```

Create a New Native Window

Problem

You need to display an additional window into which content can be loaded, and you need to be able to fine-tune how the new window appears.

Solution

The `HTMLLoader` class represents the root content of an HTML-based Adobe AIR application, and has various methods for creating and loading new native windows that require a high degree of customization and control.

Discussion

Creating and managing native windows with Adobe AIR is highly customizable. As an example, depending on the application requirements, you may want to hide the minimize and maximize buttons. You may also want to control window z-ordering, or force a particular window to always stay on top. The `NativeWindow` and `NativeWindowInitOptions` classes offer control over virtually all of these aspects of a native window. Although you can access the native window directly through the `window.nativeWindow` property, the `HTMLLoader` class provides much of the functionality for creating new native windows.

Calling `HTMLLoader.createRootWindow()` returns a reference to the `HTMLLoader` instance of the newly created window (not the native window itself). The `HTMLLoader.createRootWindow()` method can take up to four arguments controlling initial visibility, initialization options, scroll bars, and window bounds (i.e., the size and position on the screen). The initialization options are passed through an instance of `NativeWindowInitOptions`, which must be created and configured prior to creating the new native window. The `NativeWindowInitOptions` object controls aspects of the window such as whether it is resizable, or even whether it has any system chrome at all. The `NativeWindowInitOptions` constructor takes no arguments:

```
var options = new air.NativeWindowInitOptions();
var login = null;
var bounds = new air.Rectangle(
    ( air.Capabilities.screenResolutionX - 270 ) / 2,
    ( air.Capabilities.screenResolutionY - 150 ) / 2,
    270,
    150 );

options.minimizable = false;
options.maximizable = false;
options.resizable = false;

login = new air.HTMLLoader.createRootWindow( false,
options, true, bounds );
```

All of the arguments for `HTMLLoader.createRootWindow()` have default values which can be further explored in the Adobe AIR documentation. Not all of the options an application may use appear as initialization options. Additional options that may be controlled on an instance of `NativeWindow` itself include the window title, and whether it is always in front.

NOTE

In many cases, it is beneficial to start with an invisible window. This will allow the window to size and position itself, load the desired content, and then lay out and render the application without impacting what is displayed. This technique falls into a broader classification that is often referred to as *perceived performance* and is a very important aspect to consider during development.

Once a reference to the `HTMLLoader` instance of a new native window is obtained, you can load content into it via the `HTMLLoader.load()` method. The `HTMLLoader.load()` method takes a single argument which is a `URLRequest` instance that points to the HTML content to be loaded into the new window:

```
<html>
<head>

<title>Creating a New Native Window</title>

<script src="AIRAliases.js" type="text/javascript"></script>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}
</style>

<script type="text/javascript">
function doLoad()
{
    document.getElementById( 'window' ).addEventListener
```

```
( 'click', doWindow );
}

function doWindow()
{
    var init = new air.NativeWindowInitOptions();
    var bounds = null;
    var win = null;
    var login = air.File.applicationDirectory.resolvePath
    ( 'login.html' );

   bounds = new air.Rectangle( ( air.Capabilities.
   screenResolutionX - 325 ) / 2,
 ( air.Capabilities.screenResolutionY - 145 ) / 2, 325, 145 );

    init.minimizable = false;
    init.maximizable = false;
    init.resizable = false;

    win = air.HTMLLoader.createRootWindow( true, init, false,
    bounds );
    win.load( new air.URLRequest( login.url ) );
}
</script>

</head>
<body onLoad="doLoad();">

<input id="window" type="button" value="Login" />

</body>
</html>
```

Login.html

```
<html>
<head>

<title>Login</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}
</style>
```

```
<script>
function doLoad()
{
    document.getElementById( 'signIn' ).addEventListener
    ( 'click', doSignIn );
}

function doSignIn()
{
    window.nativeWindow.close();
}
</script>

</head>
<body onLoad="doLoad();">

<table>
    <tr>
         <td>Email:</td>
         <td><input id="email" name="email" /></td>
    </tr>
    <tr>
         <td>Password:</td>
         <td><input id="password" name="password" type=
         "password" /></td>
    </tr>
    <tr>
         <td colspan="2" align="right">
              <input id="signIn" type="button"
               value="Sign In" />
         </td>
    </tr>
</table>

</body>
</html>
```

Create Full-Screen Windows

Problem

For the purpose of enabling more viewing space or enabling additional interactions, your application needs to be able to run using the full screen.

Solution

The `HTMLLoader` class provides the functionality to create new native windows, which, when used in conjunction with the `NativeWindowInitOptions` class, can create transparent and full-screen native windows.

Discussion

The difference between using `NativeWindowInitOptions` for full-screen display and using `NativeWindowInitOptions` for custom windows is an additional initialization option and setting the window to fill the screen. To remove any OS-specific windowing chrome, use the `NativeWindowInitOptions.systemChrome` property. The `NativeWindowInitOptions.systemChrome` property should be set to one of the two string constants found in the `NativeWindowSystemChrome` class (see Table 4-2).

Table 4-2. String constants in NativeWindowSystemChrome

String constant	Description
`NativeWindowSystemChrome.STANDARD`	This is the default for `NativeWindow` and reflects the window chrome used on the specific operating system.
`NativeWindowSystemChrome.NONE`	This indicates that no chrome should be present, and requires that the application handle all traditional windowing tasks.

To create a full-screen window without any chrome, set the `NativeWindowInitOptions.systemChrome` property to `NativeWindowSystemChrome.NONE`. Window boundaries that match the full-screen resolution can be passed when calling `HTMLLoader.createRootWindow()`. The boundaries for the window are passed to the `HTMLLoader.createRootWindow()` method as a `Rectangle` object which specifies horizontal and vertical origination, as well as width and height. Depending on the requirements for the application, an alternative approach would be to call `NativeWindow.maximize()` or to set `NativeWindow.bounds` directly when system chrome is set to `NativeWindowSystemChrome.NONE`.

NOTE

If you find yourself confronted with an application that doesn't shut down, but whose visible windows are all closed, you're probably dealing with one of a few different challenges. The first is that you never set a size on a `NativeWindow`. The second is that you never set a `NativeWindow` to `visible`. The most common is that you used `NativeWindowSystemChrome.NONE`, but never added any content.

```
<html>
<head>

<title>Creating a Full Screen Window</title>

<script src="AIRAliases.js" type="text/javascript"></script>

<script type="text/javascript">
function doLoad()
{
    var init = new air.NativeWindowInitOptions();
    var win = null;
    var bounds = new air.Rectangle( 0,
                                    0,
                                    air.Capabilities.
                                    screenResolutionX,
                                    air.Capabilities.
                                    screenResolutionY );

    init.minimizable = false;
    init.maximizable = false;
    init.resizable = false;
    init.systemChrome = air.NativeWindowSystemChrome.NONE;

    win = air.HTMLLoader.createRootWindow( true, init,
    true, bounds );
    win.load( new air.URLRequest( 'http://www.adobe.com/
    go/air' ) );
}
</script>

</head>
<body onLoad="doLoad();">
```

```
</body>
</html>
```

File API

Write Text to a File from a String

Problem

A user has made changes to textual content in the application, which need to be saved to the local disk for offline access.

Solution

You can write text through the `File` and `FileStream` classes that are part of Adobe AIR.

Discussion

Before any reading or writing to disk takes place, a reference to a file or directory must exist in the application. You can establish a file reference in a number of ways, including via programmatic manipulation and user selection. You accomplish both of these by using the `File` class. The `File` class also contains static properties that point to common locations on the operating system. These locations include the desktop directory, user directory, and documents directory:

```
var file =
air.File.applicationStorageDirectory.
resolvePath( 'myFile.txt' );
```

The call to `File.resolvePath()` creates a reference to a file named *myFile.txt* located in the application storage directory. Once a reference has been established, it can be used in file I/O operations. Note that this doesn't actually create the file at this point.

Physically reading and writing to disk is accomplished using the `FileStream` class. The `FileStream` class does not take any arguments in its constructor:

```
var stream = new air.FileStream();
```

With the file reference available and a `FileStream` object instantiated, the process of writing data to disk can take place. The type of data that can be written may be anything from binary, to text, to value objects. You can access all of these by using the respective `FileStream` method for that operation.

The first step in writing a file is to open it using the `FileStream.open()` method. The `FileStream.open()` method takes two arguments. The first argument is the file reference created earlier that will be the destination of the output. The second argument is the type of file access the application will need. In the case of writing data to a file, the `FileMode.WRITE` static property will be most common. A second possibility is `FileMode.APPEND`, depending on the application requirements:

```
stream.open( file, air.FileMode.WRITE );
```

When writing text, an application should use `FileStream.writeMultiByte()` to ensure that data is written with the correct encoding. The `FileStream.writeMultiByte()` method takes two arguments. The first argument is the `String` object (text) that will be written to disk. The second argument is the character set to be used. The most common character set is that which the operating system is using, which is available on the `File` class as `File.systemCharset`:

```
stream.writeMultiByte( document.getElementById
('txtNote' ).value, air.File.systemCharset );
```

Once the text has been written to the file, it is important to close the file to avoid any corruption or blocking of access from other applications. You close a file using the `FileStream.close()` method.

NOTE

XML data is already in textual format and, as such, can be written to disk using this same series of steps. If the application uses the `XMLHttpRequest` object, using the `myXml.responseText` property alone may be adequate for most situations.

```
<html>
<head>

<title>Writing a Text File</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}

#save {
    position: absolute;
    right: 5px;
    bottom: 5px;
}

textarea {
    position: absolute;
    left: 5px;
    right: 5px;
    top: 5px;
    bottom: 32px;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
function doLoad()
{
    document.getElementById( 'save' ).
    addEventListener( 'click', doSave );
}

function doSave()
```

```
{
    var file = air.File.desktopDirectory.
    resolvePath( 'note.txt' );
    var note = document.getElementById( 'note' ).value;
    var stream = new air.FileStream();

    stream.open( file, air.FileMode.WRITE );
    stream.writeMultiByte( note, air.File.systemCharset );
    stream.close();
}
</script>

</head>
<body onLoad="doLoad();">

<textarea id="note"></textarea>
<input id="save" type="button" value="Save" />

</body>
</html>
```

Synchronously Read Text from a File

Problem

You want to read the contents of a small text file into your application.

Solution

Use the `File` and `FileStream` classes provided by Adobe AIR to locate, open, and operate on text files.

Discussion

You can read small files that contain text content using the `FileStream.open()` method. This method opens a file synchronously for reading. Synchronous access requires less code, but also blocks any additional user input until the data has been read. When using asynchronous access, additional user input is not blocked, but event handlers must be registered, which results in more development time.

NOTE

Although it is possible to access XML files as text, the result of this approach is a string of text that can't be readily manipulated. Accessing an XML file for use as a data source is often more easily handled using `XMLHttpRequest` or wrapper functionality offered by most JavaScript libraries.

The steps for synchronously reading a file are almost always the same:

1. Get a `File` reference.
2. Create a `FileStream` object.
3. Open the stream for synchronous access.
4. Call the appropriate `FileStream` read methods.
5. Close the file.

The first step to reading a text file is to get a reference to the resource on disk. You can establish a reference by programmatically designating a path using the appropriate property on the `File` object, such as `File.applicationStorageDirectory`. You will also have to call `File.resolvePath()` when using this approach, as the static `File` class properties always return a directory:

```
var file =
air.File.applicationStorageDirectory.
resolvePath('myFile.txt' );
```

The `FileStream` class has an empty constructor and can be instantiated anywhere in your application. The file reference just established is used during the physical process of opening the file. The mode for which the file is going to be opened must also be specified.

The `FileMode` object serves no purpose other than to provide constants for the types of file access that can be performed. These operations are `FileMode.READ`, `FileMode.WRITE`, `FileMode.UPDATE`, and `FileMode.APPEND`:

```
var stream = new air.FileStream();
stream.open( file, air.FileMode.READ );
```

You can use three `FileStream` methods to read character data from a file. The `FileStream.readUTF()` and `FileStream.readUTFBytes()` methods are specifically tuned for UTF data.

If this is the target format of the data for the application, you should use these methods directly. In the case of other character sets, you can use the `FileStream.readMultiByte()` method to specify the target format. Additional character sets are specified in the form of a string, such as `us-ascii`. There is also a convenience property on the `File` object to use the default system character set, `File.systemCharset`.

You also need to specify the number of bytes to be read in the case of `FileStream.readUTFBytes()` and `FileStream.readMultiByte()`. This sizing will depend largely on the requirements of the application. When reading the entire file is required, you can find the number of bytes available to be read on the `FileStream.bytesAvailable` property:

```
var data = stream.readMultiByte( stream.bytesAvailable,
air.File.systemCharset );
```

Once the contents of a file have been read, it is important to close the file. This operation will allow other applications to access the file:

```
stream.close();
```

Although a demonstrable amount of flexibility has been provided by Adobe AIR, the actual process in its entirety is considerably concise. This brevity is provided when performing synchronous data access operations. Synchronous file access should be reserved for smaller files regardless of reading or writing character or binary data:

```
<html>
<head>

<title>Synchronous File Access</title>

<style type="text/css">
body {
```

```
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}

textarea {
    position: absolute;
    left: 5px;
    right: 5px;
    top: 5px;
    bottom: 5px;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script>
function doLoad()
{
    var data = null;
    var file = new air.File();
    var stream = null;

    file = air.File.applicationDirectory.resolvePath(
     'the-raven.txt' );

    stream = new air.FileStream();
    stream.open( file, air.FileMode.READ );
    data = stream.readMultiByte( stream.bytesAvailable,
    air.File.systemCharset );
    stream.close();

    document.getElementById( 'editor' ).value = data;
}
</script>

</head>
<body onLoad="doLoad();">

<textarea id="editor"></textarea>

</body>
</html>
```

Asynchronously Read Text from a File

Problem

You want to read a large amount of text into your application without impacting the user interface.

Solution

Use the `File` and `FileStream` classes to asynchronously operate on the data; ensuring that the application execution is not blocked while the file is being processed.

Discussion

Files containing a large amount of data should be read using the `FileStream.openAsync()` method. This method opens a file asynchronously for reading or writing and will not block additional user input. Asynchronous file operations achieve this goal by raising events during processing. The result is that event listeners must be created and registered on the `FileStream` object.

The steps for asynchronously reading a file are almost always the same:

1. Get a `File` reference.
2. Create a `FileStream` object.
3. Create event handlers for processing data.
4. Add event listeners for asynchronous operations.
5. Open the stream for asynchronous access.
6. Close the file.

The first step to reading a text file is to get a reference to the resource on disk. You can establish a reference by programmatically designating a path using the appropriate property on the `File` object, such as `File.applicationStorageDirectory`:

```
var file =
air.File.applicationStorageDirectory.
resolvePath('myFile.txt' );
```

A `FileStream` instance is necessary to read or write to the file:

```
stream = new air.FileStream();
```

Before registering event handlers on a `FileStream` object, you must create those handlers. The events that are triggered by file I/O operations using the `FileStream` class will always pass an event object as an argument. The properties on the event object will depend on the type of event being raised. This object can be helpful in determining the target `FileStream` object, how much data is available for reading, how much data is waiting to be written, and more:

```
function doProgress( event )
{
    // Read all the data that is currently available
    var data = stream.readMultiByte( stream.bytesAvailable,
    air.File.systemCharset );

    // Append the most recent content
    document.getElementById( "editor" ).value += data;

    // Close the file after the entire contents
    // have been read
    if( event.bytesLoaded == event.bytesTotal )
    {
        stream.close();
    }
}
```

Registering for events takes place using the `addEventListener()` method:

```
stream.addEventListener( air.ProgressEvent.PROGRESS,
doProgress );
```

You can open a stream for asynchronous access using the `FileStream.openAsync()` method. The `FileStream.openAsync()` method takes two arguments that specify the file being accessed and the type of access being performed.

The `FileMode` object serves no purpose other than to provide constants for the types of file access that can be performed. These operations are `FileMode.READ`, `FileMode.WRITE`, `FileMode.UPDATE`, and `FileMode.APPEND`:

```
stream.openAsync( file, air.FileMode.READ );
```

As soon as the file is opened and new data is available in the stream, the `ProgressEvent.PROGRESS` event is triggered. Depending on the size of the file, as well as machine and network characteristics, not all of the bytes may be read in a single pass. In many cases, additional read operations take place, raising a `ProgressEvent.PROGRESS` event for each iteration.

Once all of the data has been read from the file, an `Event.COMPLETE` event is broadcast.

After the file has been read, it is important to close the file stream to ensure that other applications can access it:

```
stream.close();
```

This example provides a baseline for the various types of asynchronous access an application might choose to perform. In this case, the contents of the file are read and placed into an HTML text area each time more data is available. Asynchronous processing also provides the means for random file access (seek) without interrupting the user interface. An application should always use asynchronous access whenever the size of a file is in question.

```
<html>
<head>

<title>Asynchronous File Access</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}

textarea {
    position: absolute;
    left: 5px;
    right: 5px;
    top: 5px;
    bottom: 5px;
}
```

```
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
var stream = null;

function doLoad()
{
    var file = air.File.applicationDirectory.resolvePath(
     'the-raven.txt' );

    stream = new air.FileStream();
    stream.addEventListener( air.ProgressEvent.PROGRESS,
    doProgress );
    stream.openAsync( file, air.FileMode.READ );
}

function doProgress( event )
{
    var data = stream.readMultiByte( stream.bytesAvailable,
    air.File.systemCharset );

    document.getElementById( 'editor' ).value += data;

    if( event.bytesLoaded == event.bytesTotal )
    {
         stream.close();
    }
}
</script>

</head>
<body onLoad="doLoad();">

<textarea id="editor"></textarea>

</body>
</html>
```

Load Data from an XML File

Problem

You want to read XML data from a local file using common JavaScript techniques, and you want to manipulate the Document Object Model (DOM), not just the character data.

Solution

You can read a local XML document for its data using the `XMLHttpRequest` object, and by using a `File` object reference as the URI endpoint as opposed to a web address.

Discussion

Most JavaScript libraries, and virtually every data-oriented Ajax application, makes use of the `XMLHttpRequest` object to load data. This is a common means to accessing data from the client without refreshing the page, and it is core to Ajax development techniques. Adobe AIR includes support for the `XMLHttpRequest` object, which can be used for data access.

The `XMLHttpRequest.open()` method expects three arguments. The first argument is the HTTP method to be used for the call, which is commonly `GET` or `POST`. The third argument tells the object whether it should make the request asynchronously. The challenge in an Adobe AIR application is the second argument, which tells the object where to get its data:

```
var xml = new XMLHTTPRequest();

xml.open( 'GET', 'myData.xml', true );
```

This URI endpoint generally points to a remote server. This can still happen in an application that is online, but as Adobe AIR applications can also work offline, the endpoint needs to be pointed to a local resource. Rather than pass an endpoint to a remote server, a `File` reference can be provided:

```
var file = air.File.applicationStorageDirectory.resolve
('myData.xml' );
var xml = new XMLHttpRequest();
```

```
xml.onreadystatechange = function()
{
    if( xml.readystate == 4 )
    {
        // Work with data
    }
}

xml.open('GET', file.url, true );
xml.send( null );
```

The key distinction to make for this example is the use of the `File.url` property, which the `XMLHttpRequest` object understands and uses to access the appropriate data. Using this approach results in a traditional DOM that can be used to traverse and manipulate the XML data in the file. Additionally, you can use this approach with common JavaScript libraries.

Given

```
<rolodex>
    <contact>
        <first>Kevin</first>
        <last>Hoyt</last>
    </contact>
    ...
</rolodex>
```

Example

```
<html>
<head>

<title>Reading XML Data (using XMLHttpRequest)</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>
```

```
<script type="text/javascript">
var contacts = air.File.applicationDirectory.resolvePath
( 'rolodex.xml' );

function doLoad()
{
    var xml = new XMLHttpRequest();

    xml.onreadystatechange = function()
    {
         var elem = null;
         var first = null;
         var last = null;
         var rolodex = null;

         if( xml.readyState == 4 )
         {
              rolodex = xml.responseXML.documentElement.
              getElementsByTagName
( 'contact' );

              for( var c = 0; c < rolodex.length; c++ )
              {
                   first = rolodex[c].getElementsByTagName
                   ( 'first' )[0].textContent;
                   last = rolodex[c].getElementsByTagName
                   ( 'last' )[0].textContent;

                   elem = document.createElement( 'div' );
                   elem.innerText = first + " " + last;
                   document.body.appendChild( elem );
              }
         }
    }

    xml.open( 'GET', contacts.url, true );
    xml.send( null );
}
</script>

</head>
<body onLoad="doLoad();">



</body>
</html>
```

Create a Temporary File

Problem

An application needs to store transient information during file processing, and cannot assume that adequate memory exists to store the data in memory.

Solution

Creating temporary files with `File.createTempFile()` is an ideal means to store transient information while relieving the overhead of additional memory.

Discussion

The `File` class contains a static `File.createTempFile()` method that you can use to establish a temporary file. The temporary file is created at a destination determined by the operating system. Temporary files are also automatically given a unique name to avoid collision with other files that may be present:

```
var temp = air.File.createTempFile();
```

Once a temporary file has been created, you can use the `File` and `FileStream` APIs to interact with the file as you would any other file.:

```
var stream = new air.FileStream();

stream.open( temp, air.FileMode.WRITE );
stream.writeMultiByte('Hello', air.File.systemCharset );
stream.close();
```

You can use the `File.moveTo()` and `File.moveToAsync()` methods after the fact, should you decide that it is necessary to keep the temporary file for later reference. Both `move` methods take two arguments. The first argument is a `File` reference to the destination location. The second argument is a `Boolean` value that controls overwriting any existing file. If the second argument is set to `false`, and a collision occurs, the application throws an error:

```
var move = air.File.desktopDirectory.resolve('temp.txt' );

try
{
    temp.moveTo( move, false );
} catch( ioe ) {
    alert('Can\'t move file:\n' + ioe.message );
}
```

The JavaScript `try`/`catch` block will receive an error object of type `IOError`. The `IOError` class has available numerous properties that you can use for further evaluation. The exception in the previous code snippet raises the error message that is generated by Adobe AIR:

```
<html>
<head>

<title>Creating a Temporary File</title>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
function doLoad()
{
    var stream = new air.FileStream();
    var temp = air.File.createTempFile();
    var move = air.File.desktopDirectory.resolvePath
    ( 'temp.txt' );

    stream.open( temp, air.FileMode.WRITE );
    stream.writeMultiByte( 'Hello World!', air.File.
    systemCharset );
    stream.close();

    try
    {
         temp.moveTo( move, false );
    } catch( ioe ) {
         alert( 'Could not move temporary file:\n' +
         ioe.message );
    }

}
</script>
```

```
</head>
<body onLoad="doLoad();">



</body>
</html>
```

Iterate the Contents of a Directory

Problem

The application is required to display information about a directory as part of the user interface.

Solution

Use the `File.browseForDirectory()` method to prompt the user to select a directory, and then use the `File.getDirectoryListing()` method to iterate through the contents of the directory.

Discussion

The `File` class provides numerous properties that you can use to get specific information about files on disk. Also, various methods on the `File` class pertain to getting a directory listing. Although an application can specify a directory programmatically, you can use `File.browseForDirectory()` to prompt the user to select a directory using the native dialog. Once a location on the local disk has been specified, the `File.getDirectoryListing()` method returns an `Array` of `File` objects for the currently referenced directory.

Before prompting the user to select a directory using the native dialog, the application needs to establish and register an event handler for `Event.SELECT`. The `Event.target` property on the raised event object will contain a reference to the `File` object that invoked the browse operation.

The `File.browseForDirectory()` method takes one argument, a `String` representing additional information that will be placed in the dialog box. This `String` is not the title of the di-

alog, as is the case with `File.browseForOpen()`. There is also no need to specify `FileFilter` objects, as the dialog box presented is specific to directories, and no files will be displayed.

After the user has selected a directory, the registered event handler will be called. The file reference, whether using a class or global reference, or `Event.target`, will now contain the path to the selected directory. At this point, `File.getDirectoryListing()` can be called, which returns an `Array` of `File` objects for the selected directory (as represented by the file reference). The `File.getDirectoryListing()` method takes no arguments:

```
var listing = directory.getDirectoryListing();
```

The `File` class can represent both files and directories on the local filesystem. You can use the `File.isDirectory` property to determine whether a specific `File` instance references a file or a directory.

NOTE

See the API documentation for a complete list of data exposed by the File API.

```
<html>
<head>

<title>Selecting a Directory</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
var directory = null;

function doBrowse()
```

```
{
    directory.browseForDirectory( 'Select a directory of
    files:' );
}

function doLoad()
{
    directory = air.File.documentsDirectory;
    directory.addEventListener( air.Event.SELECT, doSelect );

    document.getElementById( 'browse' ).addEventListener
    ( 'click', doBrowse );
}

function doSelect( e )
{
    var files = directory.getDirectoryListing();
    var elem = null;
    var name = null;
    var mod = null;
    var size = null;

    for( var f = 0; f < files.length; f++ )
    {
         name = files[f].name;

         mod = files[f].modificationDate;
         mod = ( mod.month + 1 ) + '/' + mod.date + '/' +
         mod.fullYear;

         size = Math.ceil( files[f].size / 1000 ) + ' KB';

         elem = document.createElement( 'div' );
         elem.innerText = name + ' is ' + size + '
         and was last modified on ' + mod;

         document.body.appendChild( elem );
    }
}
</script>

</head>
<body onLoad="doLoad();">

<input id="browse" type="button" value="Browse" />
```

```
</body>
</html>
```

File Pickers

Browse for a File

Problem

An application needs to prompt the user to select a file to open from the local system using a native dialog.

Solution

The `File` class allows an application to prompt the user to select one or more files of a specific type from the local system.

Discussion

The `File` class provides numerous browse methods that present the native dialog for the specified operation. In the case of browsing for a single file to open, the appropriate method is `File.browseForOpen()`. This method takes a required string argument for the title of the dialog box, and an optional `Array` of `FileFilter` objects.

`FileFilter` objects allow an application to filter the viewable files in the native dialog box. This argument is `null` by default, which allows the user to select any file to which he would normally have access (i.e., not hidden files). An application can provide as many filters as necessary, by placing multiple `FileFilter` objects in an `Array` and passing that `Array` as the second argument to `File.browseForOpen()`.

None of the browse methods on the `File` class is static, and as such, an existing reference to a valid `File` object must first be available. The directory represented by that `File` object reference will be selected by default when the dialog is displayed:

```
var file = air.File.documentsDirectory;
var filters = new Array();
```

```
filters.push( new FileFilter( "Image Files", "*.jpg" ) );
file.browseForOpen( file, filters );
```

When a file selection has been made, Adobe AIR will raise an event in the issuing application. To catch that event, the application must have first registered an event listener. The event that gets raised is `Event.SELECT`, and an event object will be passed to the handler:

```
var file = air.File.documentsDirectory;
var filters = new Array();

filters.push( new air.FileFilter( "Image Files", "*.jpg" ) );

file.addEventListener( air.Event.SELECT, doSelect );
file.browseForOpen( file, filters );

function doSelect( event )
{
    alert( file.nativePath );
}
```

A useful property of the `Event` object that is sent to the handler is the "target" which contains a reference to the originating `File` object. Nothing is returned from the dialog operation to be assigned to a `File` object, as the originating object will now hold a reference to the directory selected by the user. For this purpose, it may be beneficial to have a class or global reference to the `File` object, and even to reuse it:

```
<html>
<head>

<title>Selecting a File</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>
```

```
<script type="text/javascript">
var file = null;

function doBrowse()
{
    var filters = new Array();

    filters.push( new air.FileFilter( 'Image Files',
    '*.jpg' ) );
    file.browseForOpen( 'Select Photo', filters );
}

function doLoad()
{
    file = air.File.documentsDirectory;
    file.addEventListener( air.Event.SELECT, doSelect );

    document.getElementById( 'browse' ).
    addEventListener( 'click', doBrowse );
}

function doSelect( e )
{
    var elem = document.createElement( 'div' );

    elem.innerText = file.nativePath;
    document.body.appendChild( elem );
}
</script>

</head>
<body onLoad="doLoad();">

<input id="browse" type="button" value="Browse" />

</body>
</html>
```

Browse for Multiple Files

Problem

An application needs to prompt the user to select multiple files from the local system using the native dialog.

Solution

Use the `File.browseForOpenMultiple()` method to prompt the user with a dialog box that will allow for multiple file selection.

Discussion

Using the `File` class to open a single file is predominantly the same as using the `File` class to open multiple files. In the case of allowing the user to select multiple files, the appropriate method to use is `File.browseForOpenMultiple()`. The `File.browseForOpenMultiple()` method takes the same two arguments that the `File.browseForOpen()` method takes: a `String` to be used in the title of the dialog, and an `Array` of `FileFilter` objects.

Once the user has selected the files from the dialog, `FileListEvent.SELECT_MULTIPLE` will be broadcast. The event object that is sent to the handler will be of type `FileListEvent`. The `FileListEvent` class contains a `files` property, which will be an `Array` of `File` objects representing the files that the user selected:

```
var file = air.File.documentsDirectory;

file.addEventListener( air.FileListEvent.SELECT_MULTIPLE,
doSelect );

function doSelect( event )
{
    for( var f = 0; f < event.files.length; f++ )
    {
        ...
    }
}
```

Here is the complete code:

```
<html>
<head>

<title>Selecting Multiple Files</title>

<style type="text/css">
body {
```

```
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
var file = null;

function doBrowse()
{
    var filters = new Array();

    filters.push( new air.FileFilter( 'Image Files',
    '*.jpg' ) );
    file.browseForOpenMultiple( 'Select Photos', filters );
}

function doLoad()
{
    file = air.File.documentsDirectory;
    file.addEventListener( air.FileListEvent.SELECT_MULTIPLE,
    doSelect );

    document.getElementById( 'browse' ).addEventListener
    ( 'click', doBrowse );
}

function doSelect( e )
{
    var elem = null;
    var name = null;
    var size = null;

    for( var f = 0; f < e.files.length; f++ )
    {
         name = e.files[f].name;
         size = Math.ceil( e.files[f].size / 1000 );

         elem = document.createElement( 'div' );
         elem.innerText = name + ' (' + size + ' KB)';

         document.body.appendChild( elem );
    }
}
```

```
</script>

</head>
<body onLoad="doLoad();">

<input id="browse" type="button" value="Browse" />

</body>
</html>
```

Browse for a Directory

Problem

Application requirements dictate that you allow users to select a directory in which they will store data.

Solution

Use the `File.browseForDirectory()` method to prompt the user to select a directory.

Discussion

The `File.browseForDirectory()` method creates a native dialog box that allows users to select a directory. The method takes a required `String` argument, which will be used to provide additional information to the user about the purpose of the selected directory.

When a directory selection has been made, Adobe AIR will raise an event in the issuing application. To catch that event, the application must have first registered an event listener. The event that gets raised is `Event.SELECT`, and an event object will be passed to the handler:

```
var file = air.File.applicationStorageDirectory;

file.addEventListener( air.Event.SELECT, doSelect );
file.browseForDirectory( "Where do you want to store your
photos?" );

function doSelect( event )
{
```

```
    alert( file.nativePath );
}
```

Nothing is returned from the dialog operation to be assigned to a `File` object, as the originating object will now hold a reference to the directory selected by the user. For this purpose, it may be beneficial to have a class or global reference to the `File` object, and even to reuse it:

```
<html>
<head>

<title>Selecting a Directory</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
var directory = null;

function doBrowse()
{
    directory.browseForDirectory( 'Select a directory of
    files:' );
}

function doLoad()
{
    directory = air.File.documentsDirectory;
    directory.addEventListener( air.Event.SELECT, doSelect );

    document.getElementById( 'browse' ).addEventListener
    ( 'click', doBrowse );
}

function doSelect( e )
{
    var files = directory.getDirectoryListing();
    var elem = null;
```

```
    var name = null;
    var mod = null;
    var size = null;

    for( var f = 0; f < files.length; f++ )
    {
         name = files[f].name;

         mod = files[f].modificationDate;
         mod = ( mod.month + 1 ) + '/' + mod.date + '/' +
         mod.fullYear;

         size = Math.ceil( files[f].size / 1000 ) + ' KB';

         elem = document.createElement( 'div' );
         elem.innerText = name + ' is ' + size + '
         and was last modified on ' + mod;

         document.body.appendChild( elem );
    }
}
</script>

</head>
<body onLoad="doLoad();">

<input id="browse" type="button" value="Browse" />

</body>
</html>
```

Service and Server Monitoring

Monitor Connectivity to an HTTP Server

Problem

Your application needs to monitor and determine whether a specific HTTP server can be reached.

Solution

Use the `URLMonitor` class to detect network state changes in HTTP/S endpoints.

Discussion

Service monitor classes work through event notification and subsequent polling of the designated endpoint. Service monitoring is not an integrated function of Adobe AIR directly, and needs to be added before it can be used.

The classes for service monitoring are contained in the *service monitor.swf* file, which you can find in the *frameworks* directory of the Adobe AIR SDK. You should copy this file into the application project folder; you can include it through the use of the HTML `SCRIPT` tag. You also need to include the *service monitor.swf* file in the packaged Adobe AIR application. The `SCRIPT` tag used to include service monitoring functionality must come before the *AIRAliases.js* file is declared. You also must specify the content type on the `SCRIPT` tag as `application/x-shockwave-flash`:

```
<script src="servicemonitor.swf"
type="application/x-shockwave-flash"></script>
<script src="AIRAliases.js"
type="text/javascript"></script>
```

The `URLMonitor` class takes a single argument in the constructor, an instance of the `URLRequest` class. The `URLRequest` constructor takes a `String` that represents the URL service endpoint to query. The `URLRequest` class also contains information about how to query the endpoint (i.e., `GET`, `POST`), and any additional data that should be passed to the server:

```
var request = air.URLRequest( 'http://www.adobe.com' ) ;
var monitor = new air.URLMonitor( request );
```

The `URLMonitor` class will raise a `StatusEvent.STATUS` event when the network status changes. Once the event handler has been registered, the `URLMonitor` instance can be told to start watching for network start changes:

```
monitor.addEventListener( air.StatusEvent.STATUS, doStatus );
monitor.start();
```

After a network change has been propagated as an event, you can use the `URLMonitor.available` property on the originating `URLMonitor` instance to check for the presence of a connection. The `URLMonitor.available` property returns a `Boolean` value that reflects the state of the network. As it is necessary to query the originating `URLMonitor` instance for network availability, you should declare the object in a scope that is accessible across the application:

```
<html>
<head>

<title>Connectivity to an HTTP Server</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}
</style>

<script src="servicemonitor.swf"
type="application/x-shockwave-flash"></script>
<script type="text/javascript"
src="AIRAliases.js"></script>

<script type="text/javascript">
var monitor = null;

function doLoad()
{
    var request = new air.URLRequest
    ( 'http://www.adobe.com' );

    monitor = new air.URLMonitor( request );
    monitor.addEventListener( air.StatusEvent.STATUS,
    doStatus );
    monitor.start();
}

function doStatus( e )
{
```

```
    var elem = document.createElement( 'div' );

    elem.innerText = monitor.available;

    document.body.appendChild( elem );
}
</script>

</head>
<body onLoad="doLoad();">



</body>
</html>
```

Monitor Connectivity to a Jabber Server

Problem

A Jabber chat client is required to reflect network presence in the user interface, but the endpoint is a Jabber server on a specific port, and not HTTP/S.

Solution

Use the `SocketMonitor` class to detect network state changes against TCP/IP socket endpoints.

Discussion

The service monitoring features are not built into Adobe AIR directly, and need to be added before they can be used. The *servicemonitor.swf* file, which is included in the Adobe AIR SDK, must be imported as an application resource and included via an HTML `SCRIPT` tag. The content type on the `SCRIPT` tag must be specified, and the `SCRIPT` tag for the service monitor classes must come before the *AIRAliases.js* `SCRIPT` tag.

```
<script src="servicemonitor.swf"
type="application/x-shockwave-flash"></script>
<script src="AIRAliases.js"
type="text/javascript"></script>
```

The SocketMonitor class takes two arguments in the constructor: a String that represents the host endpoint, and a port on which the server is listening:

```
var host = 'im.mydomain.com';
var port = 5220;
var monitor = new air.SocketMonitor( host, port );
```

The SocketMonitor class will raise a StatusEvent.STATUS event when the network status changes. Once the event handler has been registered, calling the SocketMonitor.start() method will start watching the network for changes:

```
monitor.addEventListener( air.StatusEvent.STATUS, doStatus );
monitor.start();
```

After a network change has been propagated as an event, you can use the SocketMonitor.available property on the originating SocketMonitor instance to check for the presence of a connection. The SocketMonitor.available property returns a Boolean value that reflects the state of the network. As a best practice, you should declare the SocketMonitor object in a scope that is accessible across the application and is referenced directly during event handling:

```
<html>
<head>

<title>Connectivity to a Jabber Server</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}
</style>

<script src="servicemonitor.swf"
type="application/x-shockwave-flash"></script>
<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
var monitor = null;

function doLoad()
```

```
{
    monitor = new air.SocketMonitor
    ( 'im.mydomain.com', 1234 );
    monitor.addEventListener
    ( air.StatusEvent.STATUS, doStatus );
    monitor.start();
}

function doStatus( e )
{
    var elem = document.createElement( 'div' );

    elem.innerText = monitor.available;

    document.body.appendChild( elem );
}
</script>

</head>
<body onLoad="doLoad();">

</body>
</html>
```

Online/Offline

Cache Assets for Offline Use

Problem

You want to load an asset from a URL and store it for use when the application is offline.

Solution

Use the File I/O API to save the requested asset to the application's store and read that file on subsequent requests.

Discussion

In this example, we will load an XML file that is at a known URL. Once the data has been loaded, it will be saved to the

local disk, and on subsequent requests for the document it will be loaded from the local disk instead of from the remote location.

First, we will use the `XMLHttpRequest` object to load the XML data from the remote location. The `XMLHttpRequest.open()` method takes three arguments. The first argument is the method of the HTTP request that is being made. The second argument is the URI of the location of the data being loaded. The third argument is a `Boolean` that specifies whether the operation will be asynchronous.

Once we have specified these arguments in the `open` method, we will call the `send` method. The `send` method takes a single argument that contains the content that is to be sent with the request. In our case, we won't send any data with the request:

```
var xml = new XMLHttpRequest();
xml.open( "GET", "http://www.foo.com/data.xml", true );
xml.send( null );
```

Because we are loading the data asynchronously, we need to create a handler for the response which is called once the data has loaded from the server. This handler will be added before the `send` method is called. Within this handler, we will save the data that is located in the `responseText` property of the `XMLHttpRequest` instance to a known location on the local filesystem for retrieval in subsequent requests. We cover reading and writing text to the local system elsewhere in the book, and therefore we won't cover it in detail here:

```
xml.onreadystatechange = function()
{
    if( xml.readyState == 4 ) // the request is complete
    {
        // write the data to the local system
       var file =
       air.File.applicationStorageDirectory.resolvePath
       ("data.xml");
        var fileStream = new air.FileStream();
        fileStream.open( file, air.FileMode.WRITE );
        fileStream.writeMultiByte( xml.responseText ,
                                   air.File.systemCharset );
        fileStream.close();
```

```
        }
    }
```

Before each request of the data, we will need to check whether the *data.xml* file exists. If it exists, we do not need to load the file using the `XMLHttpRequest` object and can use the File API to load it from the disk. This allows us to load the data even if the user is not currently online:

```
var data = null;
var file = air.File.applicationStorageDirectory.resolvePath
("data.xml");

if( file.exists )
{
    var fileStream = new air.FileStream();
    fileStream.open( file, air.FileMode.READ );
    data =
    fileStream.readMultiByte( fileStream.bytesAvailable,
                              air.File.systemCharset );
    fileStream.close();
}
else
{
    // read the data via XMLHttpRequest and write that
    // data to the file system
}
```

Here is the complete example:

```
<html>
<head>
    <title>Caching Assets for Offline Use</title>
    <script src="AIRAliases.js"></script>
    <script>

        var file =
        air.File.applicationStorageDirectory.resolvePath
        ("data.xml");

        function onLoad()
        {
            if( file.exists )
            {
                var fileStream = new air.FileStream();
                fileStream.open( file, air.FileMode.READ );
                document.getElementById( "dataText" ).value =
```

```
                    fileStream.readMultiByte(
                        fileStream.bytesAvailable,
                        air.File.systemCharset );
                fileStream.close();
            }
            else
            {
                var xml = new XMLHttpRequest();
                xml.open( "GET",
                          "http://www.foo.com/data.xml",
                          true );

                xml.onreadystatechange = function()
                {
                    if( xml.readyState == 4 ) // the request
is complete
                    {
                        var file = air.File.
applicationStorageDirectory. resolvePath("data.xml");
                        var fileStream = new air.FileStream();
                        fileStream.open( file, air.
FileMode.WRITE );
                        fileStream.writeMultiByte( xml.
responseText , air.File. systemCharset );
                        fileStream.close();

                        document.getElementById(
"dataText" ).value = xml. responseText;
                    }
                }

                xml.send( null );
            }
        }
    </script>
</head>
<body onload="onLoad()">
    <textarea id="dataText"></textarea>
</body>
</html>
```

Drag-and-Drop

Use Drag—and-Drop from HTML

Problem

You want to allow users to drag files, images, text, and other data types into and out of HTML-based AIR applications.

Solution

By using Adobe AIR's Drag and Drop implementation in JavaScript, developers can react to drag-and-drop operations that occur on HTML DOM objects.

NOTE

Adobe AIR's support for drag-and-drop within HTML content is based on the WebKit implementation. You can find more information on this at *http://developer.apple.com/documentation/AppleApplications/Conceptual/SafariJSProgTopics/Tasks/DragAndDrop.html*.

Discussion

One of the benefits of developing for the desktop is providing users with a more integrated experience when interacting with multiple applications. One of the most frequently used user gestures is to drag-and-drop files, data, and other elements between applications and the desktop and between the applications themselves.

This example will demonstrate how you can accept text being dragged items into your application as well as support dragging elements out. It will also show you how to modify the drag effect to demonstrate for the user what type of drag operations he can perform with the element he is dragging as well as the ability to modify the drag image.

Two flows are important to consider when using drag-and-drop operations in HTML. First, we will examine the flow for HTML elements that are drag-enabled:

1. The element specifies that it is available for drag operations.
2. The user selects the element and starts to drag it.
3. The element receives an `ondragstart` event and sets the data which will be transferred, as well as specifies which drag operations are supported. It can also specify a custom drag image at this time.
4. The element receives `ondrag` events while it is being dragged.
5. The user drops the element being dragged and receives an `ondragend` event.

The typical flow for HTML elements that want to receive drop operations is as follows:

1. The user drags an item over the element listening for drop events.
2. The element receives an `ondragenter` event and specifies which drop operations are allowed.
3. The element receives `ondragover` operations continuously as the item is being dragged over.
4. The user drops the item and the receiving element receives an `ondrop` event.
5. Alternatively, if the user moves the dragged item outside the boundaries of the listening element, it will receive an `ondragleave` event.

Linked text and highlighted text elements are drag-enabled by default. To disable this functionality, use the `-khtml-user-drag:none` style. Conversely, to enable other HTML elements to be drag-enabled, use the `-khtml-user-drag:element` style.

To manipulate the data that is being transferred as part of the drag operation, listen for the `ondragstar` operation and use the `dataTransfer` object that is attached to the event object. The

dataTransfer object has two data modification methods: getData and setData. The setData method takes two parameters: the MIME type and the string of data that conforms to that type. You can call the setData method multiple times, and it allows you to store multiple data types. For example, if you wanted to specify a text/plain type and a text/uri- list type, you would do the following:

```
function dropStartListener( event )
{
    event.dataTransfer.setData( "text/plain", "Adobe" );
    event.dataTransfer.setData( "text/uri-list",
                                "http://www.adobe.com" );
}
```

If setData is called for a MIME type that already exists on the element being dragged, that data will be overwritten. Retrieving data from an element that is being dragged can occur only within an ondrop event handler. The getData method takes a MIME type as its only parameter and returns the value of the MIME type if it exists on the element being dragged. For example:

```
function dropListener( event )
{
    alert( event.dataTransfer.getData( "text/plain" ) );
// Adobe
}
```

AIR supports the following MIME types:

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"

When a user is dragging data from one application to another, or from one location in your application to another, you may want to indicate to the user which operations (copy, link, or move) are available. By using the effectAllowed and dropEffect properties of the dataTransfer object, you can specify

which operations are allowed. You can see the list of available values for these properties by reading the WebKit documentation referenced earlier.

The `effectAllowed` property tells the system what operations the source element supports. The `dropEffect` property specifies the single operation that the current target receiving the drag event supports. The operating system then uses this information regarding which effects the source and destination targets support, and allows the user to make that choice. Generally, the user chooses by using the system's standard keyboard modifiers.

To modify the drag image that is displayed to the user as she is dragging the item, use the `setDragImage` method of the `dataTransfer` object. This method takes three arguments. The first argument is a JavaScript `Image` object which references the image that will appear to the user. The second and third arguments are the respective X and Y offsets that will modify the position of that image relative to the cursor's X and Y positions on-screen.

Assume that we had the following HTML element in our document. Notice that we explicitly specify that this element is draggable using the `-khtml-user-drag:element` style:

```
<div id="box" style="-khtml-user-drag:element"
      ondragstart="onBoxDragStart
(event)"></div>
```

We can then change the image by listening for the `ondragstart` event and modify the image using the `setDragImage` method:

```
// First create a reference to our drag image in the
// main document scope.
var dragImage = new Image();
dragImage.src = "app:/images/dragImage.png";

// This method gets called when a drag starts
// on our 'box' element.
function onBoxDragStart( event )
{
    // Set the data we would like to be transferred.
```

```
            event.dataTransfer.setData("text/plain",
            "This is a red box!");

            // Modify the drag image to use the reference
            // we created above.
            event.dataTransfer.setDragImage( dragImage, 0, 0 );
        }
```

Here is the full example:

```
    <html>
    <head>
        <title>HTML Drag Test</title>
        <script src="AIRAliases.js" />
        <script>

        // DROP EVENTS

        function onDragEnter(event)
        {
            air.trace("onDragEnter");
            event.dataTransfer.dropEffect = "copy";
            event.preventDefault();
        }

        function onDrop(event)
        {
          air.trace("onDrop");
          air.trace( event.dataTransfer.getData("text/plain") );
          air.trace( event.dataTransfer.getData("text/uri-list") );
        }

        function onDragOver(event)
        {
            event.preventDefault();
        }

        // DRAG EVENTS

        function onDragStart(event)
        {
            air.trace("onDragStart");
            event.dataTransfer.setData("text/plain",
               "This is the URL I am dragging" );
            // We overwrite the default URL specified in the
            // anchor tag with a different URL. When the data
            // is dropped, this is the URL that will be
```

```
        //  transferred.
        event.dataTransfer.setData("text/uri-list",
                                   "http://www.foo.com" );
        event.dataTransfer.effectAllowed = "all";
    }

    function onDragEnd( event )
    {
        air.trace("onDragEnd");
    }


    var dragImage = new Image();
    dragImage.src = "app:/images/dragImage.png";

    function onBoxDragStart( event )
    {
         event.dataTransfer.setData("text/plain", "This is a
red box!");
         event.dataTransfer.setDragImage( dragImage, 0, 0 );
    }

    </script>
</head>
<body>
    <div style="margin: 0px auto; width: 80%;
    background-color: white; border: solid black;">
        <div style="background-color: lightblue;
        border-bottom: solid black; padding: 3px;
        font-family: sans-serif; font-weight: bold;"
             ondragenter="onDragEnter(event)"
             ondragover="onDragOver(event)"
             ondrop="onDrop(event)">
            Drop Here
        </div>
        <p>
            <span id="content" ondragstart="onDragStart(event)"
                               ondragend="onDragEnd(event)">
           <a href="http://www.adobe.com">Drag Me
           (text/uri-list)</a>
            </span>
        </p>
        <p>
         <div id="box" style="-khtml-user-drag:element"
      ondragstart="
onBoxDragStart(event)"></div>
        </p>
```

```
    </div>
</body>
</html>
```

Embedded Database

Adobe AIR includes an embedded SQLite database that AIR applications can leverage. SQLite is a compact open source database that supports ACID transactions, requires zero configuration, implements most of SQL92, and supports strings and BLOBs up to 2 GB in size. All database information is stored in a single file on disk, which you can freely share between machines, even if they have different byte orders.

NOTE

You can find more information about SQLite on the project web site, at *http://www.sqlite.org*.

Adobe AIR supports both synchronous and asynchronous database transactions. A synchronous transaction will block additional user interface interaction until the transaction has been completed, but can be substantially less effort to code. An asynchronous approach will allow additional interaction with the user interface while the transaction is processing, though it may require a substantial amount of code for event handlers. All of the following examples showcase an asynchronous approach.

Connect to a Database

Problem

You need to connect to a local database prior to working with the schema or altering data.

Solution

You can create and connect to a database using the single `SQLConnection.open()` method.

Discussion

SQLite stores all database information in a single file on disk. This means that before an application can access a database, it must first have a reference to the file. A single application might choose to access any number of database files. Databases are managed through the `SQLConnection` data type.

You can obtain a reference to the database file through the `File.resolvePath()` method, which takes a single argument: the name of the file that will be referenced. Files that do not yet exist can have a reference, and the `File.exists` property returns a `Boolean` to determine that file's presence on disk:

```
var db = new air.SQLConnection();
var file =
air.File.applicationStorageDirectory.
resolvePath( 'mycrm.db' );
```

The extension to the database file is not specific and can be named as necessary for the application.

To operate using asynchronous database transactions, an application must first create and register a handler for the events in which it is interested. In the case of establishing a connection to a database, the `SQLEvent.OPEN` event will be monitored. Among various other properties, you can use the `SQLEvent.type` property to determine the status of the database.

```
db.addEventListener( air.SQLEvent.OPEN, doDbOpen );

function doDbOpen( event )
{
    alert('Connected' );
}
```

The `SQLConnection.open()` method can take a number of different arguments. The most common arguments are the file reference to the database, and a `String` value indicating the

mode in which the database should be opened. The default value of `SQLMode.CREATE` will create the database if it does not exist, and then will establish a connection to the database.

```
<html>
<head>

<title>Connecting to a Database</title>

<script type="text/javascript" src="AIRAliases.js"></script>

<script>
var db = new air.SQLConnection();

function doDbOpen( event )
{
    alert( 'You are now connected to the database.' );
}

function doLoad()
{
    var file = air.File.applicationDirectory.resolvePath(
 'crm.db' );

    db.addEventListener( air.SQLEvent.OPEN, doDbOpen );
    db.open( file, air.SQLMode.READ );
}
</script>

</head>
<body onLoad="doLoad();">


</body>
</html>
```

Create Database Tables

Problem

An application has a specific schema it needs to provide for data storage.

Solution

You can create a database schema using the `SQLStatement` class, using SQL92 grammar.

Discussion

Once a database file has been created and a connection to the database has been established, the next likely step will be to create any required schema. You can do this using SQL92 in conjunction with the `SQLStatement` class. The `SQLStatement` class executes commands against a specified database.

Using an asynchronous approach, the best place to check for any required schema—or to create it—is in the handler for the `SQLEvent.OPEN` event. At this point, the application can be assured a connection against which statements can be executed. Along the same lines, event handlers must also be registered on the `SQLStatement` instance:

```
var stmt = new air.SQLStatement();

stmt.addEventListener( air.SQLErrorEvent.ERROR, doStmtError );
stmt.addEventListener( air.SQLEvent.RESULT, doStmtResult );
```

When applied to a `SQLStatement` object, the `SQLErrorEvent.ERROR` event is called when an error has occurred while executing a `SQLStatement.next()` or `SQLStatement.execute()` method. Conversely, the `SQLEvent.RESULT` event is called when results are returned from the database. This usually indicates a successful execution:

```
function doStmtError( event )
{
    alert( 'There has been a problem executing
    the statement.' );
}

function doStmtResult( event )
{
    alert( 'The database table has
    been created.' );
}
```

To execute a SQL statement, a `SQLConnection` instance against which to execute must be established. You can assign a `SQLConnection` instance to the `SQLStatement.sqlConnection` property. The `SQLStatement.text` property is then assigned any SQL that needs to be executed. Finally, the `SQLStatement.execute()` method is called:

```
stmt.sqlConnection = db;
stmt.text = 'CREATE TABLE IF NOT EXISTS contact ( ' +
    'id INTEGER PRIMARY KEY AUTOINCREMENT, ' +
    'first TEXT, ' +
    'last TEXT )';
stmt.execute();
```

In this case, a `CREATE TABLE` statement has been applied to the database. Additional types of SQL statements, such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, are executed in the same manner. The `SQLStatement.execute()` method can take two optional arguments: the number of rows to prefetch, and a responder object to handle events.

The prefetch option defaults to `-1`, which indicates that all rows should be returned. The responder object can be a custom object designed to handle any status or result events that take place during execution. The default responder is `null` in this case, as event handlers have been registered with the `SQLStatement` object directly:

```
<html>
<head>

<title>Creating Database Tables</title>

<script type="text/javascript" src="AIRAliases.js"></script>

<script>
var db = null;
var stmt = null

function doDbOpen( event )
{
    stmt = new air.SQLStatement();
    stmt.addEventListener( air.SQLErrorEvent.ERROR,
doStmtError );
```

```
    stmt.addEventListener( air.SQLEvent.RESULT, doStmtResult );

    stmt.sqlConnection = db;
    stmt.text = 'CREATE TABLE IF NOT EXISTS contact ( ' +
                        'id INTEGER PRIMARY KEY
AUTOINCREMENT, ' +
                        'first TEXT, ' +
                        'last TEXT )';

    stmt.execute();
}

function doLoad()
{
    var file = air.File.applicationDirectory.resolvePath(
 'crm.db' );

    db = new air.SQLConnection();
    db.addEventListener( air.SQLEvent.OPEN, doDbOpen );
    db.open( file, air.SQLMode.CREATE );
}

function doStmtResult( event )
{
    alert( 'The database table has been created.' );
}

function doStmtError( event )
{
    alert( 'There has been a problem executing
    a statement:\n' + event.error.message );
}
</script>

</head>
<body onLoad="doLoad();">



</body>
</html>
```

Store Data in a Database

Problem

An application needs to store user-provided data in a relational database on disk.

Solution

SQL92 `INSERT` statements can be created and executed using the `SQLStatement` class.

Discussion

Given a valid database file with the appropriate schema created, SQL92 statements can be executed using the `SQLStatement` object. The same `SQLStatement` object can be reused to execute multiple statements. When reusing the same `SQLStatement` object, it is important to differentiate what type of statement has just been executed. You can listen for the different actions in various ways.

```
function doSave()
{
    var first = document.getElementById( 'txtFirst' ).value;
    var last = document.getElementById( 'txtLast' ).value;

    stmt.text = 'INSERT INTO contact VALUES ( ' +
        'NULL, ' +
        '\'' + first + '\', ' +
        '\'' + last + '\' ) ';
    stmt.execute();
}
```

One approach is to assign different event handlers for the different statements that will be executed. (Do not forget to remove the old handlers.) Another approach is to specify different responder objects that have been created with the specific statement in mind. The approach used in this example is a basic state machine that tracks what type of statement has just been executed:

```
var NONE = - 1;
var CREATE_SCHEMA = 0;
```

```
var INSERT_DATA = 1;

var state = NONE;

var stmt = new air.SQLStatement();

// Other database creation and configuration

function doSave()
{
    var first = document.getElementById( 'txtFirst' ).value;
    var last = document.getElementById( 'txtLast' ).value;

    stmt.text = 'INSERT INTO contact VALUES ( ' +
        'NULL, ' +
    '\' + first + '\', ' +
        '\'' + last + '\' )';

    // Track state
    state = INSERT_DATA;
    stmt.execute();
}
```

After successfully executing a database statement, the `SQLResultEvent.RESULT` event will be triggered. If an error occurs, the `SQLStatusEvent.STATUS` event will be raised. By tracking the state, the method designed to handle the result can determine the appropriate action(s) to take. In the case of inserting new data, this may be user notification and updating of the user interface:

```
<html>
<head>

<title>Storing Data in a Database</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>
```

```
<script type="text/javascript">
var db = null;
var stmt = null

var NONE = -1;
var CREATE_SCHEMA = 0;
var INSERT_DATA = 1;

var state = NONE;

function doDbOpen( event )
{
    stmt = new air.SQLStatement();
    stmt.addEventListener( air.SQLErrorEvent.ERROR,
doStmtError );
    stmt.addEventListener( air.SQLEvent.RESULT,
    doStmtResult );

    stmt.sqlConnection = db;
    stmt.text = 'CREATE TABLE IF NOT EXISTS contact ( ' +
                        'id INTEGER PRIMARY KEY
AUTOINCREMENT, ' +
                        'first TEXT, ' +
                        'last TEXT )';

    state = CREATE_SCHEMA;
    stmt.execute();
}

function doLoad()
{
    var file = air.File.applicationDirectory.resolvePath(
 'crm.db' );

    db = new air.SQLConnection();
    db.addEventListener( air.SQLEvent.OPEN, doDbOpen );
    db.open( file, air.SQLMode.CREATE );

    document.getElementById( 'btnSave' ).addEventListener(
 'click', doSave );
}

function doSave()
{
    var first = document.getElementById( 'txtFirst' ).value;
    var last = document.getElementById( 'txtLast' ).value;
```

```
    stmt.text = 'INSERT INTO contact VALUES ( ' +
                        'NULL, ' +
                        '\'' + first + '\', ' +
                        '\'' + last + '\' )';

    state = INSERT_DATA;
    stmt.execute();
}

function doStmtResult( event )
{
    switch( state )
    {
         case CREATE_SCHEMA:
              alert( 'The database table has been created.' );
              state = NONE;

              break;

         case INSERT_DATA:
              document.getElementById( 'txtFirst' ).value = '';
              document.getElementById( 'txtLast' ).value = '';

              alert( 'A new record has been stored.' );
    }
}

function doStmtError( event )
{
    alert( 'There has been a problem executing a
    statement:\n' + event.error.message );
}
</script>

</head>
<body onLoad="doLoad();">

<div>
    First name: <input id="txtFirst" type="text" />
</div>
<div>
    Last name: <input id="txtLast" type="text" />
</div>
<div>
    <input id="btnSave" type="button" value="Save" />
</div>
```

```
</body>
</html>
```

Access Database Data

Problem

You need to generate a tabular display of data from the embedded database.

Solution

Database data can be queried using SQL92 and the `SQLStatement` class.

Discussion

You can run traditional `SELECT` statements using a `SQLStatement` object that has been referenced against an existing database. A successful execution of the `SELECT` statement invokes the registered `SQLResultEvent.RESULT` event handler. That event handler will get a `SQLResultEvent` object which contains the result data:

```
function doStmtResult( event )
{
    var elem = null;
    var results = stmt.getResult();

    if( results.data != null )
    {
         for( var c = 0; c < results.data.length; c++ )
         {
              elem = document.createElement( 'div' );
              elem.innerText = results.data[c].first + ' '
+ results.data[c].last;

              document.body.appendChild( elem );
         }
    }
}
```

NOTE

This snippet forgoes much of the state management, event registration, and database connectivity covered in other sections. Please review that content, or the example at the end of this section, for complete coverage of the topic.

To get any result data, `SQLStatement.getResult()` is called, which returns a `SQLResult` object. The `SQLResult.data` property is an `Array` of the results, if any. `SQLResult.data` Array will contain `Object` instances whose properties match the names of the columns used in the query. This `Array` can be used to iterate over the results of a query.

If the database table that is being queried has no data, or the statement did not return any data, the `SQLResult.data` property will be `null`:

```
<html>
<head>

<title>Accessing Data in a Database</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #0B333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
var db = null;
var stmt = null

var NONE = -1;
var CREATE_SCHEMA = 0;
var SELECT_DATA = 1;

var state = NONE;
```

```
function doDbOpen( event )
{
    stmt = new air.SQLStatement();
    stmt.addEventListener( air.SQLErrorEvent.ERROR,
doStmtError );
    stmt.addEventListener( air.SQLEvent.RESULT, doStmtResult );

    stmt.sqlConnection = db;
    stmt.text = 'CREATE TABLE IF NOT EXISTS contact ( ' +
                        'id INTEGER PRIMARY KEY
AUTOINCREMENT, ' +
                        'first TEXT, ' +
                        'last TEXT )';

    state = CREATE_SCHEMA;
    stmt.execute();
}

function doLoad()
{
    var file = air.File.applicationDirectory.resolvePath(
 'crm.db' );

    db = new air.SQLConnection();
    db.addEventListener( air.SQLEvent.OPEN, doDbOpen );
    db.open( file, air.SQLMode.CREATE );
}

function doStmtResult( event )
{
    var elem = null;
    var result = null;

    switch( state )
    {
         case CREATE_SCHEMA:
              stmt.text = 'SELECT * FROM contact';

              state = SELECT_DATA;
              stmt.execute();

              break;

         case SELECT_DATA:
              result = stmt.getResult();

              if( result.data != null )
```

```
                {
                     for( var c = 0; c < result.data.length;
                     c++ )
                     {
                        elem = document.createElement( 'div' );
                        elem.innerText = result.data[c].first +
                        ' ' + result.data[c].last;

                          document.body.appendChild( elem );
                     }
                }

                state = NONE;
                break;

          default:
                state = NONE;
             break;
    }
}

function doStmtError( event )
{
    alert( 'There has been a problem executing
    a statement:\n' + event.error.message );
}
</script>

</head>
<body onLoad="doLoad();">



</body>
</html>
```

Command-Line Arguments

Capture Command-Line Arguments

Problem

You need to capture command-line arguments sent to your application—either at application startup or while the application is running.

Solution

Register for the `InvokeEvent`, and capture command-line arguments passed into your application.

Discussion

Whenever an application is started, or an application is called from the command line while it is running, an `InvokeEvent` will be broadcast. The event handler for this is passed information about the event, including any arguments passed to the application on the command line.

You should register for the `InvokeEvent` during your application's initialization phase, to ensure that the event is captured when the application is initially launched.

You can register for the event from the `NativeApplication` singleton, like so:

```
function init()
{
air.NativeApplication.nativeApplication.addEventListener
(air.InvokeEvent.INVOKE,onInvoke);
}
```

This registers the `onInvoke` function as a handler for `InvokeEvent`. The handler is passed an instance of the `InvokeEvent` object, which contains a property named `arguments` which is an `Array` of `Strings` of any arguments passed to the application:

```
function onInvoke(event)
{
```

```
        air.trace("onInvoke : " + event.arguments);
    }
```

When testing your application via ADL, you can pass in command-line arguments by using the `--` argument. For example:

```
adl InvokeExample.xml -- foo "bim bam"
```

This would pass in two arguments to the application: `foo` and `bim bam`.

The complete example follows; it listens for the `InvokeEvent`, and prints out to the included `textarea` HTML control, as well as the command line via `air.trace()`:

```
<html>
<head>

    <script src="AIRAliases.js" />
    <script type="text/javascript">

      function onInvoke(event)
      {
        air.trace("onInvoke : " + event.arguments);

         var field = document.getElementById("outputField");
         field.value += "Invoke : " + event.arguments + "\n";
        }

        function init()
        {
air.NativeApplication.nativeApplication.addEventListener(air.
InvokeEvent.INVOKE,onInvoke);
        }

    </script>

</head>

<body onload="init()">

    <textarea rows="8" cols="40" id="outputField">
    </textarea>

</body>
</html>
```

Networking

Communicate on a Socket

Problem

You would like to communicate with a server using a protocol that is not directly supported by Adobe AIR (e.g., communicate with an FTP server).

Solution

Use the `Socket` class in the AIR API to send binary or text data to the server and register for events that will alert you to incoming data from the server.

Discussion

When communicating using protocols other than those directly supported by Adobe AIR, you may need to use the Socket API. The Socket API is an asynchronous API that lets you send data to a persistent socket endpoint and receive data from it in real time. You do not need to create a new `Socket` instance for each set of data sent to the same endpoint. The connection can be kept alive for the entire conversation between your client and the service to which you're connecting. This is the typical flow when using the Socket API:

1. Create a connection to the endpoint.
2. Listen for notification of connection success or failure.
3. Queue data that will be sent to the endpoint.
4. Send the data to the endpoint.
5. Listen for data incoming from the endpoint.
6. Repeat steps 3 through 5.
7. Close the connection.

The first step is to create a connection to the socket endpoint that consists of a host and a port number. For example, to

connect to an endpoint the host might be foo.com (*http://foo.com*) and the port number might be 5555. Create the instance of the `Socket` class and connect to the endpoint using that information. At this time, we will also set up our listeners to listen for the different events that the Socket can dispatch:

```
var socket = new air.Socket();
socket.addEventListener( air.Event.CONNECT, onSocketOpen );
socket.addEventListener( air.ProgressEvent.SOCKET_DATA,
onSocketData );
socket.connect( 'foo.com', 5555 );
```

We will also need to create the functions to handle the events for which we subscribed. The first event is the `air.Event.CONNECT` event. This event will tell us when the socket has been initiated and when communication with the service behind the endpoint is possible. In this example, we are sending the bytes of a UTF-8 encoded string to the service:

```
function onSocketOpen( event )
{
    // This queues up the binary representation of the
    // string 'Bob' in UTF-8 format to be sent to the
    // endpoint.
    socket.writeUTFBytes( "Bob" );

    // Send the actual bytes to the server and clear
    // the stream. We then wait for data to be sent
    // back to us.
    socket.flush();
}
```

The `air.ProgressEvent.SOCKET_DATA` event is dispatched whenever data is received. The service we are connecting to uses a simple protocol: we send a UTF-8 encoded string and it returns a UTF-8 encoded string. This makes accessing the data sent back to us very simple. To access this data, we measure the total number of bytes of data available on the Socket and read that many bytes as a UTF-8 encoded string using the `readUTFBytes()` method of the `Socket` class.

```
function onSocketData( event )
{
    var data =
      socket.readUTFBytes( socket.bytesAvailable );
```

```
        air.trace( data ); // Hello Bob
    }
```

In our example, the protocol of communication was just a single string. In some cases, depending on the service with which you're communicating, you may need to send and receive other data types. The `Socket` class provides methods for reading and writing many data types, such as `int`s, `Boolean`s, `float`s, and more. For example, if we were talking with a fictional service that required us to send a `Boolean` followed by an `int`, our `onSocketOpen` function in the preceding example could look like this:

```
function onSocketOpen( event )
{
    // First send the boolean
    socket.writeBoolean( true );
    // Now send an int
    socket.writeInt( 10 );

    // Now we send the bytes to the service and
    // clear the buffer.
    socket.flush();
}
```

This example provides a baseline of functionality that can be expanded upon to speak to many different protocols. The only current limitation is that there is not currently an SSL Socket implementation in AIR. For secure communication you will be limited to HTTPS:

```
<html>
<head>

<title>Communicating on a Socket</title>
<script type="text/javascript" src="AIRAliases.js">
</script>

<script>
var socket = null;

function init()
{
   socket = new air.Socket();
```

```
    // Create our listeners which tell us when the Socket
    // is open and when we receive data from our service.
     socket.addEventListener( air.Event.CONNECT,
     onSocketOpen );
     socket.addEventListener( air.ProgressEvent.SOCKET_DATA,
                              onSocketData );

     // Connect to our service, which is located at
     // host foo.com using port 5555.
     socket.connect( 'foo.com', 5555 );
}

function onSocketOpen( event )
{
     // This queues up the binary representation of the
     // string 'Bob' in UTF-8 format to be sent to the
     // endpoint.
     socket.writeUTFBytes( "Bob" );

     // Send the actual bytes to the server and clear
     // the stream. We then wait for data to be sent
     // back to us.
     socket.flush();
}

function onSocketData( event )
{
     var data = socket.readUTFBytes( socket.bytesAvailable );
     air.trace( data ); // Hello Bob
}
</script>

</head>
<body onload="init()">
</body>
</html>
```

Upload a File in the Background

Problem

The application user has created numerous files offline, and you now want to send those to the server without blocking the user from doing any additional work.

Solution

The `File` class in Adobe AIR provides an `upload()` method that is designed specifically for this purpose, without having to create and manage HTML forms.

Discussion

The `File.upload()` method can upload files via HTTP/S to a server for additional processing. The upload takes places just like a traditional multipart file upload from an HTML form, but without the need to manipulate forms on the client. The upload process also takes place asynchronously in the background, allowing the application to continue processing without interruption.

> **NOTE**
>
> The implementation of the receiving server is beyond the scope of this example. Numerous technologies, and tutorials for these technologies, elegantly handle file upload. You're encouraged to investigate your options.

The primary events that are useful are `ProgressEvent.PROGRESS` and `Event.COMPLETE`. These events handle notifying the application of upload progress, and when an individual upload is complete, respectively:

```
var file =
new air.File.documentsDirectory.
resolvePath( 'myImage.jpg' );

file.addEventListener( air.ProgressEvent.PROGRESS,
doProgress );
file.addEventListener( air.Event.COMPLETE,
doComplete );
```

`ProgressEvent` contains various properties that can help in reflecting upload progress in the user interface. The most notable of these properties are `ProgressEvent.bytesLoaded` and `ProgressEvent.bytesTotal`, which show how much of the file has been

uploaded and the total size of the file. `Event.COMPLETE` is broadcast once the upload is complete.

To start the upload, you first need a valid `File` object that points to a resource on disk.

Once a valid file reference is established, developers will want to call the `File.upload()` method. The `File.upload()` method can take three arguments, the first of which is a `URLRequest` object that contains information about where the file should be sent. The `URLRequest` object can also contain additional data to be passed to the receiving server. This additional data manifests itself as HTML form fields might during a traditional multipart file upload:

```
var request = new air.URLRequest(
'http://www.mydomain.com/upload' );
file.upload( request, 'image', false );
```

The second argument provided to the `File.upload()` method call is the name of the form field that contains the file data.

The third argument is a `Boolean` value that tells the upload process whether it should try a test before sending the actual file. The test upload will `POST` approximately 10 KB of data to the endpoint to see if the endpoint responds. If the service monitoring capabilities of Adobe AIR are not being used, this is a good way to check for potential failure of the process.

NOTE

More than one great web application has been caught by this subtlety. If the server is expecting the file data outright, a test upload will almost assuredly cause an error. If you intend to use the test facility, be sure that your server code is prepared to handle the scenario.

```
function doProgress( event )
{
    var pct = Math.ceil( ( event.bytesLoaded / event.
bytesTotal ) * 100 );
    document.getElementById( 'progress' ).innerText =
```

```
        pct + "%";
    }
```

The `Event.COMPLETE` event is relatively straightforward in that it signals the completion of the upload process. This is a good place to perform any filesystem maintenance that the application might otherwise need to accomplish. An example would be removing the just-uploaded file from the local disk to free up space. Another task that might be accomplished in the `Event.COMPLETE` handler is to start the upload of subsequent files:

```
<html>
<head>

<title>Uploading a File in the Background</title>

<style type="text/css">
body {
    font-family: Verdana, Helvetica, Arial, sans-serif;
    font-size: 11px;
    color: #OB333C;
}
</style>

<script type="text/javascript" src="AIRAliases.js"></script>

<script type="text/javascript">
var UPLOAD_URL = 'http://www.ketnerlake.com/work/watcher/
upload.cfm';

var file = null;

function doComplete( e )
{
    document.getElementById( 'progress' ).style.visibility =
'hidden';
    document.getElementById( 'progress' ).innerText =
'Uploading... 0%';

    document.getElementById( 'upload' ).disabled = null;
}

function doLoad()
{
    file = air.File.desktopDirectory;
```

```
    file.addEventListener( air.Event.SELECT, doSelect );
    file.addEventListener( air.ProgressEvent.
    PROGRESS, doProgress );
    file.addEventListener( air.Event.
    COMPLETE, doComplete );

    document.getElementById( 'upload' ).
    addEventListener( 'click', doUpload );
}

function doProgress( e )
{
    var loaded = e.bytesLoaded;
    var total = e.bytesTotal;
    var pct = Math.ceil( ( loaded / total ) * 100 );

    document.getElementById( 'progress' ).innerText =
'Uploading... ' +
 pct.toString() + '%';
}

function doSelect( e )
{
    var request = new air.URLRequest( UPLOAD_URL );

    request.contentType = 'multipart/form-data';
    request.method = air.URLRequestMethod.POST;

    document.getElementById( 'upload' ).disabled = 'disabled';
    document.getElementById( 'progress' ).style.visibility =
'visible';

    file.upload( request, 'image', false );
}

function doUpload()
{
    file.browseForOpen( 'Select File' );
}
</script>

</head>
<body onLoad="doLoad();">

<input id="upload" type="button" value="Upload" />
<div id="progress" style="visibility: hidden">Uploading...
0%</div>
```

```
</body>
</html>
```

Sound

Play a Sound

Problem

You need to play a sound in your application.

Solution

Use the Sound API within AIR to play an MP3 file.

Discussion

AIR includes complete support for accessing Flash Player APIs from JavaScript. This includes the `Sound` class that can be used to play local or remote MP3 files.

Playing a sound is simple, and requires two main steps:

1. Create a `URLRequest` instance that references the local or remote sound.
2. Pass the `URLRequest` to the `Sound` instance, and play it.

Here is the relevant code snippet:

```
var soundPath =
  new air.URLRequest("app-resource:/sound.mp3");
var s = new air.Sound();
    s.load(soundPath);
    s.play();
```

First, we create a `URLRequest` that points to the location of the MP3 file we will play. In this case, we use an `app-resource` URI that references the *sound.mp3* file contained in the application install directory. You can also use any valid URI, including both file and HTTP URIs:

```
var soundPath =
  new air.URLRequest("app:/sound.mp3");
```

We then create an instance of the `Sound` class, pass the reference to the MP3 path, and then call `play`:

```
var s = new air.Sound();
    s.load(soundPath);
    s.play();
```

Here is the complete example with a Play button:

```
<html>
<head>

    <script src="AIRAliases.js" />
    <script type="text/javascript">

        function playSound()
        {
            var soundPath =
             new air.URLRequest("app:/sound.mp3");
            var s = new air.Sound();
                s.load(soundPath);
                s.play();
        }
    </script>

</head>

<body>
    <input type="button" value="Play" onClick="playSound()">
</body>
</html>
```

At this point, you should have a solid understanding of Adobe AIR, how to build AIR applications, and how to work with AIR APIs. Make sure to check the resources listed in the Preface to learn more advanced Adobe AIR development techniques.

APPENDIX

AIR JavaScript Aliases

Table A-1 through Table A-15 show the JavaScript aliases created in *AIRAliases.js* and the AIR and Flash Player APIs to which they correspond.

> **NOTE**
>
> All non-aliased ActionScript APIs are accessed through the `window.runtime` property in JavaScript.

Table A-1. Top-level aliases

Alias	ActionScript API
`air.trace`	`trace`
`air.navigateToURL`	`flash.net.navigateToURL`
`air.sendToURL`	`flash.net.sendToURL`

Table A-2. File aliases

Alias	ActionScript API
`air.File`	`flash.filesystem.File`
`air.FileStream`	`flash.filesystem.FileStream`
`air.FileMode`	`flash.filesystem.FileMode`

Table A-3. Event aliases

Alias	ActionScript API
air.AsyncErrorEvent	flash.eventsAsyncErrorEvent
air.BrowserInvokeEvent	flash.events.BrowserInvokeEvent
air.DataEvent	flash.events.DataEvent
air.DRMAuthenticateEvent	flash.events.DRMAuthenticateEvent
air.DRMStatusEvent	flash.events.DRMStatusEvent
air.Event	flash.events.Event
air.EventDispatcher	flash.events.EventDispatcher
air.FileListEvent	flash.events.FileListEvent
air.HTTPStatusEvent	flash.events.HTTPStatusEvent
air.IOErrorEvent	flash.events.IOErrorEvent
air.InvokeEvent	flash.events.InvokeEvent
air.NetStatusEvent	flash.events.NetStatusEvent
air.OutputProgressEvent	flash.events.OutputProgressEvent
air.ProgressEvent	flash.events.ProgressEvent
air.SecurityErrorEvent	flash.events.SecurityErrorEvent
air.StatusEvent	flash.events.StatusEvent
air.TimerEvent	flash.events.TimerEvent
air.ActivityEvent	flash.events.ActivityEvent

Table A-4. Native window aliases

Alias	ActionScript API
air.NativeWindow	air.NativeWindow = flash.display.NativeWindow

Alias	ActionScript API
air.NativeWindowDisplayState	flash.display.NativeWindowDisplayState
air.NativeWindowInitOptions	flash.display.NativeWindowInitOptions
air.NativeWindowSystemChrome	flash.display.NativeWindowSystemChrome
air.NativeWindowResize	flash.display.NativeWindowResize
air.NativeWindowType	flash.display.NativeWindowType
air.NativeWindowBoundsEvent	flash.events.NativeWindowBoundsEvent
air.NativeWindowDisplayStateEvent	flash.events.NativeWindowDisplayStateEvent

Table A-5. Geometry aliases

Alias	ActionScript API
air.Point	flash.geom.Point
air.Rectangle	flash.geom.Rectangle
air.Matrix	flash.geom.Matrix

Table A-6. Network aliases

Alias	ActionScript API
air.FileFilter	flash.net.FileFilter
air.LocalConnection	flash.net.LocalConnection
air.NetConnection	flash.net.NetConnection
air.URLLoader	flash.net.URLLoader
air.URLLoaderDataFormat	flash.net.URLLoaderDataFormat
air.URLRequest	flash.net.URLRequest
air.URLRequestDefaults	flash.net.URLRequestDefaults

Alias	ActionScript API
air.URLRequestHeader	flash.net.URLRequestHeader
air.URLRequestMethod	flash.net.URLRequestMethod
air.URLStream	flash.net.URLStream
air.URLVariables	flash.net.URLVariables
air.Socket	air.Socket = flash.net.Socket
air.XMLSocket	flash.net.XMLSocket
air.Responder	flash.net.Responder
air.ObjectEncoding	flash.net.ObjectEncoding

Table A-7. System aliases

Alias	ActionScript API
air.Capabilities	flash.system.Capabilities
air.System	flash.system.System
air.Security	flash.system.Security
air.Updater	flash.desktop.Updater

Table A-8. Desktop aliases

Alias	ActionScript API
air.Clipboard	flash.desktop.Clipboard
air.ClipboardFormats	flash.desktop.ClipboardFormats
air.ClipboardTransferMode	flash.desktop.ClipboardTransferMode
air.NativeDragManager	flash.desktop.NativeDragManager
air.NativeDragOptions	flash.desktop.NativeDragOptions
air.NativeDragActions	flash.desktop.NativeDragActions
air.Icon	flash.desktop.Icon

Alias	ActionScript API
air.DockIcon	flash.desktop.DockIcon
air.InteractiveIcon	flash.desktop.InteractiveIcon
air.NotificationType	flash.desktop.NotificationType
air.SystemTrayIcon	flash.desktop.SystemTrayIcon
air.NativeApplication	flash.desktop.NativeApplication

Table A-9. Display aliases

Alias	ActionScript API
air.NativeMenu	flash.display.NativeMenu
air.NativeMenuItem	flash.display.NativeMenuItem
air.Screen	flash.display.Screen
air.Loader	flash.display.Loader
air.Bitmap	flash.display.Bitmap
air.BitmapData	flash.display.BitmapData

Table A-10. UI aliases

Alias	ActionScript API
air.Keyboard	flash.ui.Keyboard
air.KeyLocation	flash.ui.KeyLocation
air.Mouse	flash.ui.Mouse

Table A-11. Utility aliases

Alias	ActionScript API
air.ByteArray	flash.utils.ByteArray
air.CompressionAlgorithm	flash.utils.CompressionAlgorithm
air.Endian	flash.utils.Endian
air.Timer	flash.utils.Timer

Alias	ActionScript API
air.XMLSignatureValidator	flash.security.XMLSignatureValidator

Table A-12. HTML aliases

Alias	ActionScript API
air.HTMLLoader	flash.html.HTMLLoader
air.HTMLPDFCapability	flash.html.HTMLPDFCapability

Table A-13. Media aliases

Alias	ActionScript API
air.ID3Info	flash.media.ID3Info
air.Sound	flash.media.Sound
air.SoundChannel	flash.media.SoundChannel
air.SoundLoaderContext	flash.media.SoundLoaderContext
air.SoundMixer	flash.media.SoundMixer
air.SoundTransform	flash.media.SoundTransform
air.Microphone	flash.media.Microphone
air.Video	flash.media.Video
air.Camera	flash.media.Camera

Table A-14. Data aliases

Alias	ActionScript API
air.EncryptedLocalStore	flash.data.EncryptedLocalStore
air.SQLCollationType	flash.data.SQLCollationType
air.SQLColumnNameStyle	flash.data.SQLColumnNameStyle
air.SQLColumnSchema	flash.data.SQLColumnSchema
air.SQLConnection	flash.data.SQLConnection
air.SQLError	flash.errors.SQLError

Alias	ActionScript API
air.SQLErrorEvent	flash.events.SQLErrorEvent
air.SQLErrorOperation	flash.errors.SQLErrorOperation
air.SQLEvent	flash.events.SQLEvent
air.SQLIndexSchema	flash.data.SQLIndexSchema
air.SQLMode	flash.data.SQLMode
air.SQLResult	flash.data.SQLResult
air.SQLSchema	flash.data.SQLSchema
air.SQLSchemaResult	flash.data.SQLSchemaResult
air.SQLStatement	flash.data.SQLStatement
air.SQLTableSchema	flash.data.SQLTableSchema
air.SQLTransactionLockType	flash.data.SQLTransactionLockType
air.SQLTriggerSchema	flash.data.SQLTriggerSchema
air.SQLUpdateEvent	flash.events.SQLUpdateEvent
air.SQLViewSchema	flash.data.SQLViewSchema

Table A-15. Service Monitor aliases

Alias	ActionScript API
air.ServiceMonitor	air.net.ServiceMonitor
air.SocketMonitor	air.net.SocketMonitor
air.URLMonitor	air.net.URLMonitor

Index

Symbols

A

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

B

C

D

E

F

G

H

I

J

K

R

S

T

U

V

W

X

Y

Z