Getting Started with the Alpha Release of Apollo

Apollo *for* Adobe® Flex™ Developers

Pocket Guide

Adobe Developer Library

Mike Chambers, Robert L. Dixon, & Jeff Swartz

O'REILLY®

Adobe Apollo® for Flex™: Pocket Guide
by Mike Chambers, Robert L. Dixon, and Jeff Swartz

Printed in the United States of America.

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 (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Steve Weiss
Production Editor: Philip Dangler
Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:
March 2007: First Edition.

ISBN-10: 0-596-51391-7
ISBN-13: 978-0-596-51391-7
[C]

CHAPTER 4

Using the File System API

Apollo provides a file I/O API that lets applications read and write files and directories on the user's computer. The file I/O API includes the following functionality:

- Create and delete files and directories
- Copy and move files and directories
- List the contents of directories
- Get system information on files and directories
- Read and write binary files
- Read and write text files
- Serialize and deserialize ActionScript objects

The low-level functionality for working with the file system is accessed via ActionScript. The Flex framework for Apollo includes components for working with files and directories, but these are graphical components for navigating the file system and selecting files and directories. They do not provide direct access to the more fundamental file I/O operations.

In addition to the information in this chapter, see the examples presented in "Working with the File System" in Chapter 5. Those examples illustrate many of the concepts described in this chapter, and they provide working MXML code that you can test, using Flex Builder or the Apollo SDK.

Security Model

Apollo will eventually provide a complete security model for managing access to local resources, such as the file system. However, this security model has not been implemented in the Apollo Alpha 1 build.

It is important to remember that Apollo applications are installed to and run from the user's computer. Apollo applications have a different security context and security model than those of web browsers. Because of this, the same rules that apply to downloading and running other applications also apply to downloading and running Apollo applications. Users should download and install applications only from trusted sources.

Accessing Files and Directories

Apollo applications can run on multiple platforms, including Windows and Mac OS. The Apollo file API uses platform-neutral code syntax so you don't have to write any OS-specific code.

For example, the way you represent a path to a file differs between Mac OS and Windows:

- A typical file path on Mac OS is */Users/joe/Documents/test.txt*
- A typical file path on Windows is *C:\Documents and Settings\joe\My Documents\test.txt*

However, you can use exactly the same Apollo components, classes, methods, and properties to access files in either operating system.

An ActionScript `File` object is a pointer to a file or directory. The `File` class includes the static property `documentsDirectory`, which contains a `File` object that points to the user's documents directory. This is the *My Documents*

directory on Windows, and it is the *Documents* subdirectory of the user directory on Mac OS, as illustrated in the following code:

```
trace(File.documentsDirectory.nativePath)
     // On Windows:
     //       C:\Documents and Settings\joe\MyDocuments
     // On Mac OS:  /Users/joe/Documents
```

Once you point a `File` object to a directory, you can use the `resolve( )` method to modify it to point to a file or subdirectory within that directory (or within a subdirectory). For example, the following code creates an *Apollo Test* subdirectory of the user's documents directory:

```
var newDir:File = File.documentsDirectory;
newDir = newDir.resolve("ApolloTest");
newDir.createDirectory( );
```

A `File` object can point to either a file or a directory. Also, a `File` object may point to a file or directory that does not exist, as in the previous example. This lets you point a `File` object to a directory location that you wish to create.

File Class Properties for Accessing Common Directory Locations

The `File` class includes the following static properties, which point to commonly used directory locations:

Property	Description
`File.appStorageDirectory`	Each installed Apollo application is given a unique *application storage directory*. This is a good place to store files that the application may want to maintain but that the user will probably need not see. This may include log files, cache files, and preferences files.
`File.appResourceDirectory`	The application's install directory.
`File.currentDirectory`	This is the directory from which the file was launched. You may use this property to resolve the file path of any command-line parameters that were passed to the application.
`File.desktopDirectory`	This is the user's desktop directory.

Property	Description
`File.documentsDirectory`	This is the *My Documents* directory on Windows, and the *Documents* subdirectory of the user directory on Mac OS.
`File.userDirectory`	This is the user's home directory. For example, on Mac OS, it is the *Users/username* directory, and on Windows it is typically *c:\\Document and Settings\username*.

The url and nativePath Properties of a File Object

The `url` property of a `File` object returns the location of a file or folder as a platform-independent string that begins with a URL scheme, such as `file`, as in the following:

```
var directory:File = File.userDirectory;
trace(directory.url)
  // on Windows: file:///C:/Documents%20and%20Settings
  // on Mac OS: file:///Users
```

whereas the `nativePath` property of a `File` object returns a string that is unique to Windows or Mac OS. For example, you can use this code to point to a specific file on a Windows computer:

```
var file:File = new File( );
file.nativePath = "c:/ApolloTest/surprise.txt";
```

However, it is generally better to start with one of the static properties listed in the table in the previous section (such as the `File.appStorageDirectory`)—that point to known directories on the operating system—and then use the `resolve( )` method to create to a relative path based on that directory, as in this code:

```
var logFile:File = File.appStorageDirectory;
logFile = logFile.resolve("log.txt");
```

Use the application store directory to store files that you want your application to be able to access in the future but that the end user may not need to know about. For instance, this is a good place to store preferences files.

URI Schemes

A URI scheme is specified at the beginning of a URL, such as `file` in the following example:

```
file:///c:/ApolloTest/test.txt
```

In addition to the `file` URI scheme, Apollo supports the new URI schemes `app-storage` and `app-resource`.

app-storage

Identifies the application storage directory, as shown in the following example:

```
var logFile:File = File.appStorageDirectory;
logFile = logFile.resolve("log.txt");
trace(logFile.url); // app-storage:/log.txt
```

app-resource

Identifies this application's installation folder, as in the following:

```
var installDir:File = new File( );
installDir.url = "app-resource:/";
installDir = installDir.resolve("HelloWorld-app.xml");
trace(installDir.url); // app-resource:/HelloWorld-app.xml
```

file

The `url` property of other `File` object returns a standard `file` URL scheme:

```
var file:File = File.documentsDirectory;
file = file.resolve("ApolloTest/test.txt");
trace(file.url);
  // On Windows:
  // file:///C:/Documents%20and %20Settings/ ... /test.txt
  // On Mac OS:
  // file:///Users/userName/Documents/ ... /test.txt
```

Asynchronous and Synchronous Versions of Methods

Some of the methods of the `File` class (such as `File.copyFile( )` and `File.copyFileAsync( )`) and of the `FileStream` class have both synchronous and asynchronous versions.

The synchronous methods don't relinquish control until the file operation is complete. The asynchronous methods run in the background, allowing other ActionScript processes to take place at the same time. When the asynchronous file operation finishes, an event is dispatched to notify listeners that it is done.

Here's an example of copying a file using the synchronous `copyTo( )` method:

```
var file1:File = File.documentsDirectory.
resolve("ApolloTest/test.txt");
var file2:File = File.documentsDirectory.
resolve("ApolloTest/copy of test.txt");
file1.copyTo(file2);
trace("Not output until the file is copied.");
```

Here's an example of copying a file using the asynchronous `copyToAsync( )` method:

```
var file1:File = File.documentsDirectory.
resolve("ApolloTest/test.txt");
var file2:File = File.documentsDirectory.
resolve("ApolloTest/copy of test.txt");
file1.copyToAsync(file2);

file1.addEventListener(Event.COMPLETE, completeHandler);
trace("This line executes before the complete event.");
trace("So does this line.");

private function completeHandler(event:Event):void {
  trace("Done.");
}
```

The following table lists the asynchronous methods of the File class (all of which have synchronous counterparts) and the events that can fire after the method is called:

Asynchronous File method	Events
copyToAsync( )	complete, ioError
deleteDirectoryAsync( )	complete, ioError
deleteFileAsync( )	complete, ioError
listDirectoryAsync( )	directoryListing, ioError
moveToAsync( )	complete, ioError
moveToTrashAsync( )	complete, ioError

When you open a file, use either the open( ) or openAsync( ) method of the FileStream object. The first opens the file for synchronous operations, and the second opens the file for asynchronous operations. For more information, see "The open( ) and openAsync( ) Methods" later in this chapter.

Use asynchronous methods whenever you want to make sure that other essential ActionScript-driven processes—such as progress bar animation—continue while the file operations take place. For example, you could use the open( ) (synchronous) method of a FileStream object if you are going to write a small file (1 MB or less) and use the openAsync( ) method when writing larger files, or when the file size is unknown.

For more information on asynchronous methods in general, see the "Handling Events" chapter in *Programming ActionScript 3.0*, which is available at:

http://livedocs.macromedia.com/flex/2/docs/Part5_ProgAS.html

Reading Directory Contents

The File.listDirectory( ) method returns an array listing of File objects that represent the files and directories contained within the specified directory. For example, the following code lists the contents of the desktop directory:

```
var directory:File = File.desktopDirectory;
var contents:Array = directory.listDirectory( );
for (var i:uint = 0; i < contents.length; i++) {
    if (contents[i].isDirectory) {
        trace(contents[i].name);
    } else {
        trace(contents[i].name,
        contents[i].size,
        "bytes");
    }
}
```

The `File.listDirectory( )` method returns only the root level files and directories in a directory. It does not recursively search through subdirectories for their contents. You can, of course, write code to traverse subdirectories, though if you do so, you might want to use the `File.listDirectoryAsync( )` method so that other ActionScript-driven processes can continue while the directory listings are being compiled.

Also see "Getting a Directory Listing" in Chapter 5.

Getting File Information

The `File` class includes a number of properties that contain information about a file or directory.

Property	Description
`exists`	States whether the file or directory exists. This is worth checking, for example, before you attempt to read, write, copy, or move a file.
`isDirectory`	States whether the `File` object points to a directory (`true`) or a file (`false`). You will want to check this before attempting directory-specific operations (such as the `listDirectory( )` method) or attempting file-specific operations (such as reading a file).
`isHidden`	States whether the file or directory is hidden.
`nativePath`	Notes the operating system-specific path to the file or directory.
`parent`	Notes the parent directory of the File instance.
`url`	Notes the operating system-independent path to the file or directory.

The File class also inherits the following useful properties from the FileReference class:

Property	Description
creationDate	The date the file or folder was created.
modificationDate	The date when the file was last modified.
name	The file or folder name.
size	The size of the file, in bytes.

Copying and Moving Files and Directories

The File.copyTo( ) and File.moveTo( ) methods copy or move a file or directory to a specified new location. For example, the following code copies the *test.txt* file in the *Apollo Test* subdirectory of the user's documents directory to the *User Data* subdirectory of the application storage directory:

```
var file1:File = File.documentsDirectory.resolve("Apollo
Test/test.txt");
var destination:File = File.appStorageDirectory.
resolve("User Data");
destination.createDirectory( );
var file2:File = destination.resolve("test.txt");
file1.copyTo(file2);
```

Note the call to the File.createDirectory( ) method, which ensures that the destination directory exists.

The following code moves the *Apollo Test 1* subdirectory of the user's documents directory to the *Apollo Test 2* subdirectory (effectively renaming the directory):

```
var dir1:File = File.documentsDirectory;
dir1 = dir1.resolve("Apollo Test 1");
var dir2:File = File.documentsDirectory;
dir2 = dir2.resolve("Apollo Test 2");
```

You might want to use the asynchronous versions of these methods, `File.copyToAsync( )` and `File.moveToAsync( )`, if the copy or move operation could take a long time.

Each of these methods includes a `clobber` parameter, which you can set to `true` to have the operation overwrite existing files. By default, this parameter is set to `false`.

Creating Files and Directories

The `File.createTempFile( )` and `File.createTempDirectory( )` static methods of the `File` class let you create a temporary file or directory. Apollo ensures that the temporary file or directory created by these methods is new and unique. For example, the following code creates a temporary file:

```
var bufferStorage:File = File.createTempFile( );
```

Temporary files and directories are not automatically deleted when you close an Apollo application. You will generally want to delete temporary files and directories before closing the application. See the next section, "Deleting Files and Directories," for more details.

The `File.createDirectory( )` method lets you create a directory in the location specified by the `File` object:

```
var directory = File.documentsDirectory;
directory = directory.resolve("ApolloTest");
```

When you open a `FileStream` object with write capabilities then directories are created automatically, if needed. For more information about `FileStream` objects, see "Reading and Writing Files" later in this chapter.

Deleting Files and Directories

The `File.deleteFile( )` method permanently deletes a file, and the `File.deleteDirectory( )` method permanently deletes

a directory. The `File.moveToTrash( )` method lets you move a file or directory to the system trash.

Each of these methods also has an asynchronous counterpart.

Reading and Writing Files

The `FileStream` class provides methods that let your application read and write files.

Here's the general process for reading and writing to a file:

1. Set up a `File` object that points to the file you want to read or write.

   For details, see "Accessing Files and Directories," earlier in this chapter.
2. Instantiate a `FileStream` object—for example:

   ```
   var stream:FileStream = new FileStream( );.
   ```
3. Call the `FileStream.open( )` or `FileStream.openAsync( )` method, passing in the `File` object as the `file` parameter and passing an appropriate file mode as the `fileMode` parameter. For example:

   ```
   stream.open(file, FileMode.READ);
   ```

   For more information, see "File Open Modes" later in this chapter.
4. If you called the `FileStream.openAsync( )` method, set up the appropriate event listener functions.

   For more information, see "The open( ) and openAsync( ) Methods," next.
5. Call the appropriate read and write method for your data.

   For more information, see "Read and Write Methods" later in this chapter
6. Close the file, using the `FileStream.close( )` method. For example:

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

Steps 3, 4, and 5 are described in more detail the sections that follow. First, here is a sample of code for reading UTF-8 text from a file synchronously:

```
var file:File = File.appStorageDirectory;
file = file.resolve("settings.xml");
var stream:FileStream = new FileStream( );
stream.open(file, FileMode.READ);
var data:String = stream.readUTFBytes(stream.
bytesAvailable);
stream.close( );
```

Here is some code that reads the same data asynchronously:

```
var file:File = File.appStorageDirectory;
file = file.resolve("settings.xml");
var stream:FileStream = new FileStream( );
stream.openAsync(file, FileMode.READ);
stream.addEventListener(Event.COMPLETE, readData);
var data:String;

private function readData(event:Event):void {
  data = stream.readUTFBytes(stream.bytesAvailable);
  stream.close( );
}
```

The open( ) and openAsync( ) Methods

Your application needs to open a file before it can read from or write to the file.

When you open a file with the `FileStream.openAsync( )` method, the file is opened for asynchronous operations, and you've registered event listeners to monitor progress.

The `FileStream.open( )` method opens the file for synchronous operations. If your application opens the file using this synchronous method, all subsequent calls to methods that read or write to the file will be done synchronously as well. In the following example, each of the calls to `stream.open( )`, `stream.writeUTFBytes( )`, and `stream.close( )` will complete before the next call is made.

```
var newFile:File = File.documentsDirectory;
file = file.resolve("ApolloTest/test.txt");
```

```
var stream:FileStream = new FileStream( )
stream.open(file, FileMode.WRITE);
stream.writeUTFBytes("This is some sample text.");
stream.close( );
```

The advantage of opening a file for synchronous operations is that you can write less code to complete a task. The disadvantage is that execution of other ActionScript code can be delayed if the file operations take a while. As a result, if you are working with large files or opening files that are shared on slow networks, you should consider using the `FileStream.openAsync( )` method instead.

When you use the `openAsync( )` method, the following processes are all handled asynchronously:

Closing the file
: The `FileStream` object dispatches a `close` event when the file is closed.

Reading data into the read buffer
: The `FileStream` object dispatches `progress` events as data is read, and it dispatches a `complete` event once all the data is read. However, once data is read, calling a read method (such as `readBytes( )`) to read data is a synchronous process.

I/O errors
: The `FileStream` object dispatches an `ioError` event upon encountering an error. This may occur for a number of reasons, such as attempting to open a file that doesn't exist or attempting to write to a file that is locked. However, some errors, such as attempting to read from a file that has not been opened, throw exceptions (rather than dispatch `ioError` events) because the Apollo runtime can detect the error condition instantly.

Before calling the `FileStream.openAsync( )` method, your application should set up event listener functions to handle those events in which it is interested.

The following example opens a file in asynchronous read mode. After the file has been opened, the complete event will be dispatched (unless there is an error, in which case the `ioError` event will be dispatched instead). The `completeHandler()` method then calls the `FileStream.readBytes()` method, which starts reading data from the file as an array of bytes, in asynchronous mode. When all the bytes have been read from the file, the `complete` event will be dispatched:

```
var file:File = File.documentsDirectory.
resolve("ApolloTest/test.txt");
var stream:FileStream = new FileStream();

stream.addEventListener(ProgressEvent.PROGRESS,
progressHandler);
stream.addEventListener(Event.COMPLETE, completeHandler);
stream.addEventListener(IOErrorEvent.IO_ERROR, ioErrorHandler);
stream.addEventListener(Event.CLOSE, closeHandler);

stream.openAsync(file, FileMode.READ);

var data:ByteArray = new ByteArray();

private function progressHandler(event:ProgressEvent):void {
    trace(stream.bytesAvailable, "bytes read.");
}
private function completeHandler(event: Event):void {
    data = stream.readBytes(stream.bytesAvailable);
    stream.close();
}
private function ioErrorHandler(event:IOErrorEvent):void {
    trace("An I/O error was encountered.");
}
private function closeHandler(event: Event):void {
    trace("File closed.");
}
```

File Open Modes

The `FileStream.open()` method and `FileStream.openAsync()` method both accept two parameters: the `file` parameter corresponding to the file that you want to open, and the

fileMode parameter, which is a string defining the capabilities of the FileStream object. The possible values for the fileMode parameter are defined as constants in the FileMode class.

For example, the following code opens the file synchronously for write operations, but not for read operations:

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

Here are the FileMode constants and their meanings:

FileMode constant	Definition
FileMode.APPEND	The file is opened in write-only mode, with all written data appended to the end of the file. Upon opening, any nonexistent file is created.
FileMode.READ	The file is opened in read-only mode. The file must exist (missing files are not created).
FileMode.UPDATE	The file is opened in read/write mode, and data can be written to any position in the file or appended to the end. Upon opening, any nonexistent file is created.
FileMode.WRITE	The file is opened in write-only mode. If the file does not exist, it will be created. If the file does exist, it will be overwritten.

Read and Write Methods

The FileStream class includes a number of read and write methods, each corresponding to the format of the data being read or written. For example, you can use the readUTFBytes( ) and writeUTFBytes( ) methods to read or write an array of bytes, whereas the readByte( ) and writeByte( ) methods read or write a single byte at a time. All in all, there are 26 read and write methods. For details on each, see the description of these methods in the *ActionScript 3.0 Language Reference*, which is distributed with Apollo Alpha 1.

Even though reading and writing text data may seem trivial, you should consider the encoding of the text in the file. The readUTFBytes( ) and writeUTFBytes( ) methods provide

means to read and write UTF-8–encoded text. The `readMultiByte()` and `writeMultiByte()` methods let you specify a different character encoding for the file data. There are other factors to consider as well. For example, a UTF file may start with a UTF byte order mark (BOM) character, which defines the UTF encoding and the byte order (or "endianness") of the data.

For more information, see the "Data formats, and choosing the read and write methods to use" section of the *Apollo Developer's Guide* (*http://www.adobe.com/go/apollodocs*).

More Information

For examples of reading and writing files, see the following sections in Chapter 5:

- "Writing a Text File from a String"
- "Reading a Text File into a String"
- "Encoding Bitmap Data into PNG or JPEG Format and Writing It to the File System"
- "Serializing and De-Serializing ActionScript Objects to the File System"