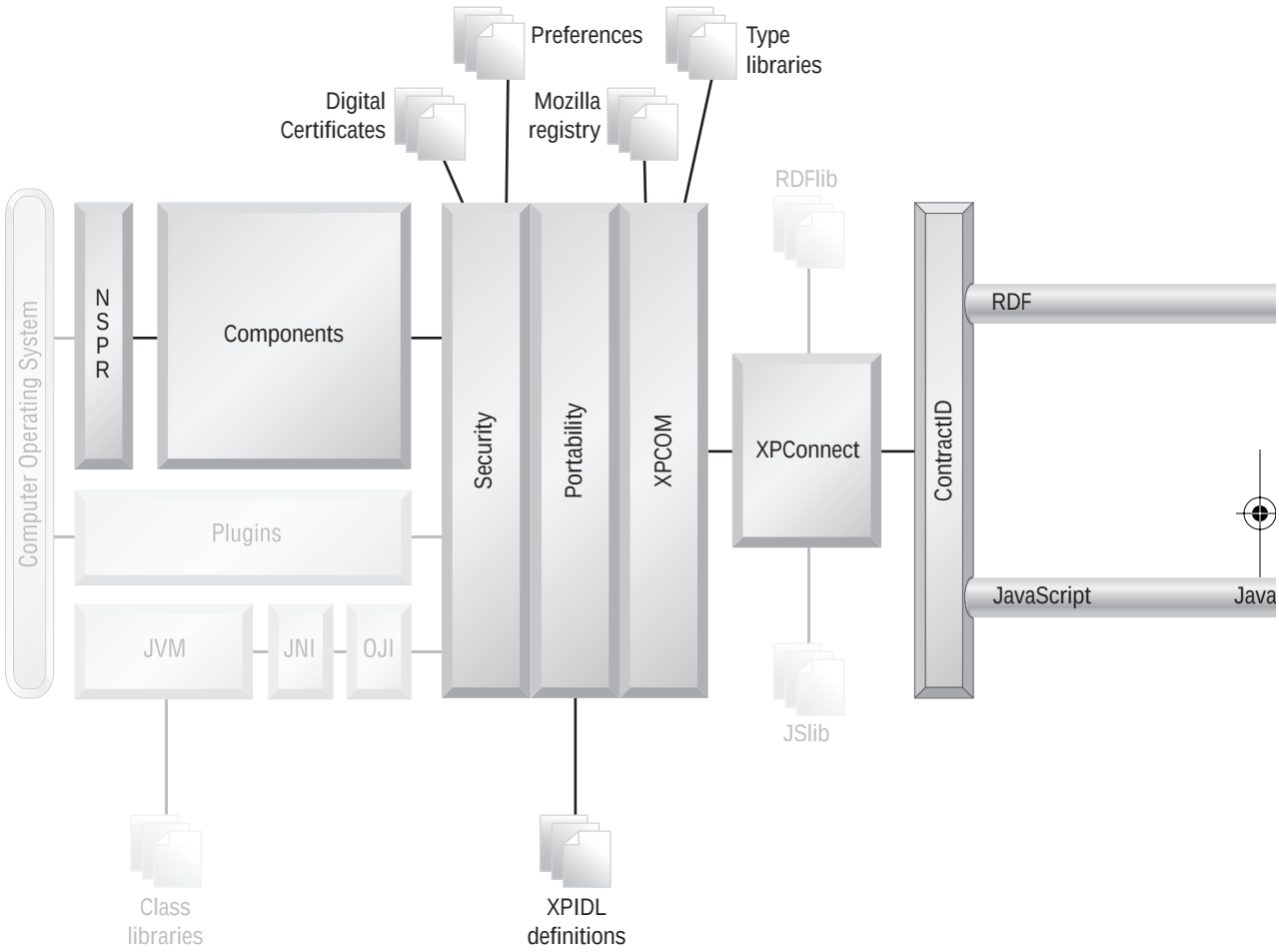
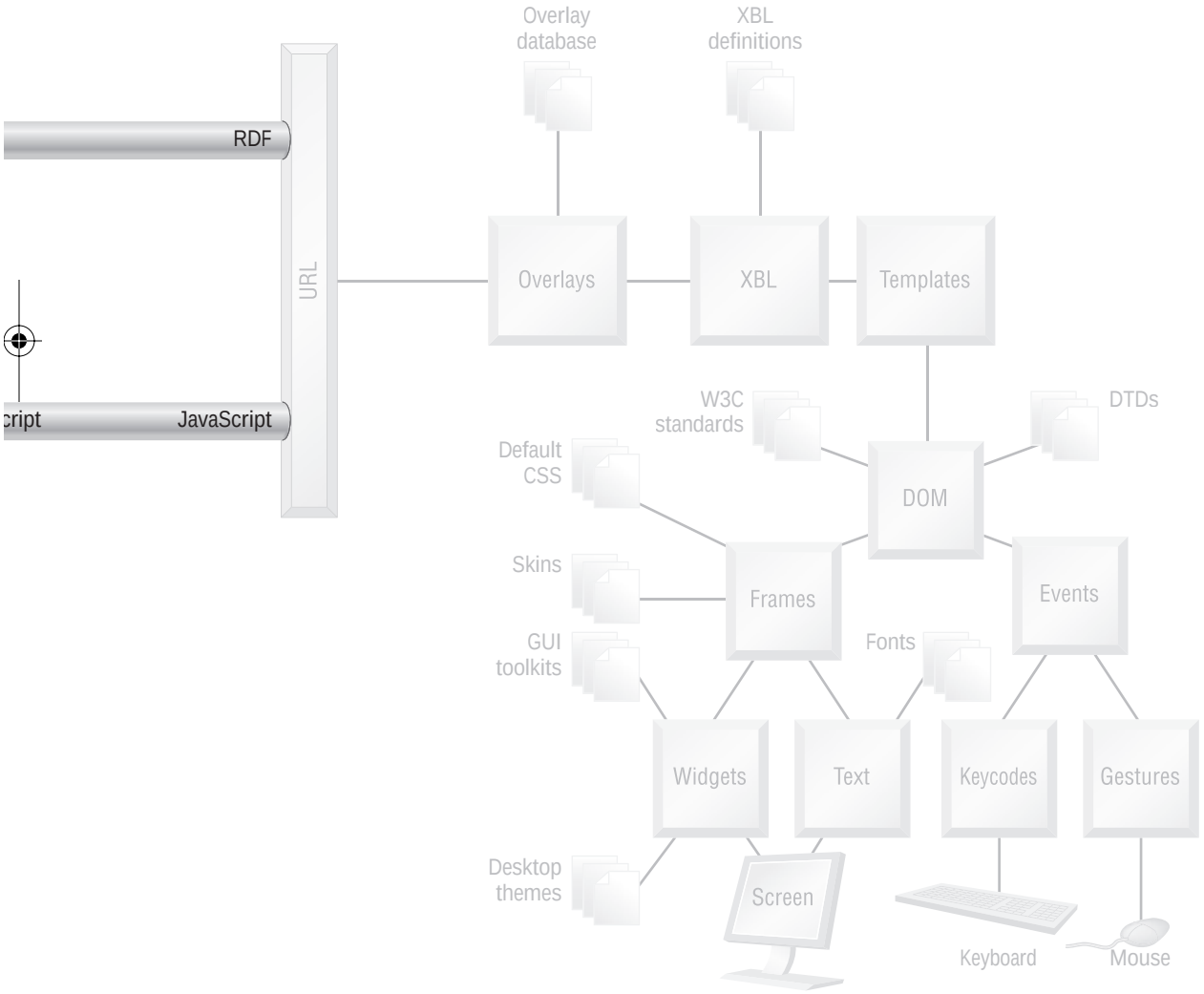

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 16

XPCOM Objects



The Mozilla Platform is a base for building software applications. It contains an object library of over a thousand objects. Many of these objects have nothing to do with GUIs. This chapter explains which of those objects solves which common programming problems.

A thousand example scripts is too big a goal for a single chapter—the best that can be done here is to provide some pointers and guidance. In addition to this discussion, you'll need to read the XPIDL interface definitions for the objects discussed. Many script fragments are presented here to complement those interfaces. All solutions presented use JavaScript.

Mozilla's object library consists mostly of XPCOM components. Without XPCOM components, an application programmer is trapped within an XML document, whether it be HTML or XUL. Inside such a document, URLs, HTTP, SOAP, and WSDL are the only mechanisms out. By adding XPCOM components, everything opens up dramatically. Components add support for networking, databases, files, and processes—all the meat and drink of traditional software applications. XPCOM components are available on all platforms where Mozilla runs and are entirely portable with only a very few exceptions.

Mozilla's set of XPCOM components are just like any 3GL or OO library. Just as C++ and Java have a stream concept and stream objects, so too does Mozilla. Just as C, Perl, and numerous other languages have a file concept, so too does Mozilla. Mozilla objects are therefore standard programmer fare—almost.

The reason that these objects are *almost* standard fare is because Mozilla is still version 1. This newness has affected the set of components that are available. Rather than having a very broad range of low-level objects, Mozilla consists of some low-level objects, some mid-level objects, and some very application-specific high-level objects. The platform was first designed to build a Web browser application suite, and so objects exist to support that goal at all levels of abstraction. The platform has not had the extensive, general-purpose design that the Java class libraries have received. On the other hand, a thousand objects is not a trivial total; it approaches the size of Perl's extensive library of modules.

A second nonstandard aspect of Mozilla's objects is that many objects are network-centric. Navigator, Messenger, and Chat applications are all Internet clients that interact with servers, and this is reflected in the object library. Some simple operations, like loading a file, are complex because the file might be anywhere in the world. This complexity affects the interfaces used. Those interfaces are also constrained by Mozilla's security system.

If an application is not a Web browser, email client, or XML display system, then Mozilla's objects still have plenty to offer, but some of the highly specialized objects available will be of little use. On the other hand, if the application contains browser-like functionality, then specialized objects speed the development process greatly.

As shown in the NPA diagram for this chapter, the back half of Mozilla is the home of XPCOM components. The XPCOM and XPConnect technologies rely on various external files; a registry (a simple database like the Microsoft Windows registry) and type libraries (component descriptions) are foremost. Preferences and certificates are two aspects of security also stored external to components. From a programmer's point of view, the most interesting part of the XPCOM system are the externally stored XPIDL files, which contain readable descriptions of all the XPCOM interfaces.

This chapter begins with some of the key concepts of the components' environment. It then moves on to address common programming tasks and Mozilla's solutions. General programming tasks come first: more application-specific tasks come later. Finally, a look is taken at the platform itself and its security systems. The "Hands On" session in this chapter has an extensive series of examples that show how to script objects associated with RDF.

16.1 Concepts and Terms

Mozilla's collection of XPCOM components is large and a world of its own. To write scripts effectively, pick up some of the established jargon used at the scripting level.

16.1.1 Reading Others' Code: Naming Conventions

Mozilla is a huge piece of software, and regular exploration of features is commonplace. Most Mozilla code and documentation have a particular style that is worth becoming accustomed to. That style appears in the following sources:

- The XPIDL definition files that define the existing XPCOM interfaces. These interfaces are help files that you can't do without.
- The numerous at-work examples that can be found in the chrome of a built Mozilla application.
- Less importantly, the C/C++ source code of the platform. This code is more challenging and of less immediate value, but it is also the ultimate authority.

Of these information sources, the XPIDL files are the main item that you need to survive. See the introduction of this book for a URL for those files.

The Mozilla coding style uses naming conventions as usage hints. These hints are especially important in JavaScript code. JavaScript syntax provides only weak signatures for objects and methods compared with compiled languages like Java or C++.

Here follows some examples of naming hints used by Mozilla code.

Mozilla uses prefix characters to hint at the nature of an XPCOM interface. The most common prefix is `nsI`, meaning "netscape Interface." This pre-

fix is used to identify an interface that is available for application programmer use. Other prefixes like `imgI`, `inI`, `jsdI`, and `mozI` (image, inspector, JavaScript debugger, and Mozilla, respectively) serve the same purpose, but they are intended to be application- or technology-specific. They are still available for general use. The plain prefix `ns` (without a trailing `I`) is used for objects that are not intended for application programmer use. Most occurrences of plain-`ns` prefixes are hidden inside the platform.

Second, the ECMAScript standard allows a host object to report its type as a string. In Mozilla, this string is calculated by stripping a prefix from the current XPCOM interface name for that host object. The characters stripped are `nsIDOM`, so an object presenting the `nsIDOMHTMLDocument` interface will report its JavaScript object type as `HTMLDocument`.

Third, capitalization is also used for interface attributes and methods:

- Constants in DOM-based interfaces are written in `ALL_UPPERCASE_STYLE`.
- Constants in non-DOM interfaces typically use `initCapStyle`.
- Variables and methods are written in `initCapStyle`.
- RDF interfaces are an exception, they use `InitCapStyle` for method names—the first letter is capitalized.

Either way, XPIDL and JavaScript cases always match. Inside the C/C++ of the platform, method names are translated from `initCap` to `InitCap`, and interface names sometimes appear with the XPIDL capitalization and sometimes appear in `ALL_CAPS`.

Mozilla makes frequent use of a single-character prefix for interface attributes and method arguments. This notation is common in XBL bindings, platform C/C++ code, XPIDL interfaces, and ordinary JavaScript. JavaScript's use of these prefixes is somewhat patchy. These single-character prefixes are described in Table 16.1.

Table 16.1 Mozilla single-character name prefixes

Prefix	Frequency of appearance	Meaning
a	Common	aVar is a temporary variable or a function or method argument. It stands for a value or object of ordinary significance, one that is usually subjected to processing. aFile is an example.
e	Occasional	eVar is a value, usually a constant that is one item of an enumeration. eTuesday is an example.
g	Occasional	gVar is a global variable; either global to the current window (the JavaScript global object) or entirely global in the case of C/C++. gMenuControllers is an example.

Table 16.1 Mozilla single-character name prefixes (Continued)

Prefix	Frequency of appearance	Meaning
k	Occasional	kVar is a key value—one of a set of values that some specific variable can take on. It is similar to the e prefix, except that key values are often bitmasks or strings and so don't number sequentially from one. kMimeType is an example.
m	Common	mVar is a member (property or attribute) of an object. It is usually used to store private data. mLength is an example.
n	Occasional	nVar usually holds a total of some kind. It is an ordinary variable.
s,i,b,f,r,p	Rare	These prefixes are used inside the platform's C/C++ to mean a string, integer, Boolean, floating-point number, short integer, and pointer type, respectively. They are not usually seen in JavaScript code, which sometimes uses f to mean file or folder instead.

One further prefix used in the Mozilla Platform is *PR*. PR stands for Portable Runtime, and NSPR stands for Netscape Portable Runtime. NSPR is a library and coding standard designed to overcome the portability problems of C, C++, and operating systems. The Mozilla Platform uses a set of data types that are guaranteed portable, and these types have PR prefixes. This notation is occasionally exposed to XPCOM and to the application programmer.

16.1.2 Modular Programming

The Mozilla Platform breaks technology up into pieces a number of different ways. Almost every technical term for *part* or *piece* that software engineering offers is used by Mozilla. Correct usage of these terms is given here:

- **binding**. A binding is an interface written down in a particular programming language. In Mozilla, a binding is either an ECMAScript interface stated in the W3C DOM standards or an XBL binding.
- **class**. The only classes in Mozilla are the component classes of XPCOM. Each class is used to create zero or more objects. JavaScript 2.0 (ECMAScript 1.4) will have class definitions of a different type, but JavaScript 1.5 does not have these.
- **component**. A component is a thing with unique identity within the XPCOM system. It is a class with a CID (a component identifier) and a matching ContractID (like `@mozilla.org/test;1`), or it is an interface with an IID (an interface identifier). Because interfaces can't be used

without a concrete implementation but classes can, classes are thought of as the only real components.

- **interface**. An interface is a set of access points to an object. XPCOM interfaces are the only interfaces in Mozilla. Any object that supplies those access points is said to implement that interface. Each XPCOM object and XBL binding implements zero or more XPCOM interfaces; JavaScript objects may also implement XPCOM interfaces.
- **library**. The Mozilla Platform has a number of dynamically linked libraries, but those libraries are of no particular interest to the application programmer. General-purpose JavaScript scripts may informally be called libraries. Type libraries are data files that define XPCOM interfaces. These are created when the platform is compiled and automatically used by XPConnect.
- **module**. XPCOM components in the Mozilla Platform are grouped into modules, but this is only obvious to developers of the platform. Modules have no meaning to application programmers, unless a whole new XPCOM module is being created.
- **object**. Mozilla contains XPCOM objects and JavaScript objects. An XPCOM object is of a given class type and implements one or more interfaces. A JavaScript object may be anything from a simple data structure to a complex host object. JavaScript host objects are either XPCOM objects or Java objects.
- **package**. A group of related files installed in Mozilla's chrome is called a package. A package has a name that matches a file system directory name.
- **prototype**. A JavaScript object used as the basis for the construction of a new JavaScript object is called a prototype.

16.1.3 Foreign-Type Systems

The XPCOM system gives JavaScript access to other programming environments. Those other environments have their own basic types. Either those foreign types must be automatically converted to JavaScript types (and back again) or those foreign types must provide some kind of usable interface.

There are five foreign-type systems in Mozilla that are accessible from JavaScript:

- **The fundamental platform types implemented by NSPR**. These are the portable C/C++ types out of which the platform is built out.
- **RDF data types**. These are the types Mozilla can identify in RDF documents.
- **XML schema data types**. Mozilla can intelligently parse files that have this format and can identify the standard primitive types contained inside.

- **XML RPC XDR**. The RPC-over-XML network protocol is supported by Mozilla, which includes underlying XDR network-independent data types.
- **Java**. A Java JVM can be run as a Mozilla platform plugin, which gives access to Java-typed objects.

Of these type systems, automatic conversion to and from JavaScript works only for Java. The other four type systems are available from the following specific XPCOM interfaces:

- The NSPR types are all represented by interfaces that derive from `nsISupportsPrimitive`, like `nsISupportsPRInt32`.
- RDF data types have interfaces `nsIRDFLiteral`, `nsIRDFDate`, and `nsIRDFInt`, all based on `nsIRDFNode`.
- XML schema data types are named as constants in `nsISchemaSimpleType` and `nsISchemaBuiltinType` interfaces.
- The XDR encoding of XML RPC uses the NSPR types. The `nsIXmlRpcClient` interface has a factory method that can create NSPR-typed objects.

In addition to these foreign type systems, the XPCOM architecture and the components specified using that architecture are a type system of their own, one based on object types (classes and interfaces) rather than simple data types. This set of classes and interfaces is the main set of type symbols that application programmers use when building nontrivial scripts.

16.2 General-Purpose Scripting

This topic explains how to solve generic programming tasks using Mozilla.

16.2.1 Command-Line Arguments

When the Mozilla Platform is started up via the command line, it remembers the command-line arguments passed in. On Microsoft Windows, the platform does not remember the command-line arguments used to start up further windows that are managed by the same running instance.

To access the command-line arguments, use this component and interface:

```
@mozilla.org/appShell/commandLineService;1 nsICmdLineService
```

The `nsICmdLineService` exposes `argc`, the count of arguments, but not `argv`, the list of strings making up the arguments. The `argc` count is the number of argument-value pairs, not the number of whitespace-separated strings, which is the traditional UNIX/Windows definition. Because `argv` is not sup-

plied, you must guess what parameters were supplied, using the `getCmdLineValue()` method. A typical call to this method is

```
var url = cls.getCmdLineValue("-chrome");
```

If the argument passed in wasn't supplied at invocation time, then `null` is returned.

This interface also contains a factory method `getHandlerForParam()`, which returns an object with the `nsICmdLineHandler` interface. This returned object is a read-only record that contains configuration information for a handler, such as default values. Each command-line handler that exists adds to the command-line options available to the platform. Such handlers can be created in JavaScript if necessary.

It is not possible to retrieve the original, raw command-line string using JavaScript.

16.2.2 Data Structures and Algorithms

The JavaScript language provides basic arrays and objects, which are enough for most simple needs. Separate from JavaScript, the Mozilla Platform provides a huge data model that is an implementation of the W3C DOM standards. This data model has obvious uses in representing HTML, XUL, MathML, SVG, and XML documents generally. It is covered under the topic "Web Scripting" as well as being described in Chapter 5, Scripting.

Apart from the DOM, Mozilla provides very few data structure interfaces. What exists is used mostly to expose platform-internal data structures to the scripting environment. They are not really intended to be the basis for new data structures, although there is nothing wrong with reusing these interfaces when developing application objects.

XPCOM collection objects that can be used alone are listed in Table 16.2. These collections are not recommended for most uses, but are worth noting.

JavaScript is flexible to start with, and these collections do not add much by themselves. These objects are complemented by a number of cursor or iterator interfaces. There are two varieties of cursor in XPCOM. The simpler variety consists of enumerators:

```
nsIEnumerator nsIBidirectionalEnumerator
```

Enumerators are read-only cursors that step through a given collection. The collection must be static, and only one enumerator at a time is allowed. These interfaces return the `nsISupports` interface for each object they enumerate.

The more complex variety of cursor is called an iterator. Iterators support tasks like: stepping through dynamically updated collections, inserting items into collections, and simultaneous use of iterators on a given collection. These iterators, all with interfaces named

```
nsI{something}Iterator
```

Table 16.2 XPCOM collection objects

Interface	Implemented in	Description
nsIArray	@mozilla.org/array;1	An XPCOM version of a read-only JavaScript array
nsIMutableArray	@mozilla.org/array;1	Adds methods that allow modification of an nsIArray's contents
nsICollection	@mozilla.org/supports-array;1	Adds a simple collection interface to a serializable stream; not generally useful from JavaScript; use any of the other interfaces
nsIDictionary	@mozilla.org/dictionary;1	A simple collection that holds key-to-value pairs (a map), implemented in JavaScript; in Microsoft Windows a dictionary is equivalent to a map; potentially useful
nsIProperties	@mozilla.org/properties;1	A simple collection that holds key-to-value pairs, implemented in C/C++ (a map); in Java a properties collection is equivalent to a map

are rarely needed by themselves but may be useful as a design guide if complex application data structures are required. They are occasionally produced by other XPCOM interfaces.

The DOM 2 Traversal and Ranges standard supplies a powerful iterator. In Mozilla it is named `nsIDOMNodeIterator`. It is occasionally useful when working with DOM data structures.

The Mozilla Platform does not provide much in the area of general-purpose algorithms. JavaScript provides regular expressions and the `Array.sort()` method, but there is little beyond that. The sort functionality used in templated XUL cannot be applied elsewhere.

16.2.3 Databases

The Mozilla Platform has support for databases, but that support is only slowly emerging. Only the most trivial support is available in the default builds of the Mozilla application suite; other support must be sought out and set up before it is available for use. Platform support for databases falls into five groups: flat files, relational databases, application-specific databases, fact stores, and caches.

Table 16.3 describes the flat file databases inside Mozilla.

The last two items deserve some explanation. `dbm` is an old version of the standard Berkeley `db` package. It is used to create several security files stored in the user's profile. This package is not directly exposed to XPCOM and cannot be used from JavaScript.

Table 16.3 General-purpose flat file databases in Mozilla

File format	Application support	Covered under what topic?
Raw files	Read/Write	"Files and Folders," "Data Transfer"
DTD documents	Read Only	Chapter 3, Static Content
Properties files	Read Only	Chapter 5, "Scripting"—see stringbundles example
Preferences	Read / Deferred Write	"Preferences"
XML Documents	Read or Write	"Web Scripting"
RDF Documents	Read / Flushed Write	"Data Sources"
Mozilla Registry	Read / Write	Chapter 17, Deployment
dbm	Unavailable	See text
Mdb	Unavailable	See text

`Mdb`, the "message database," is a custom-designed, single-file database invented for Mozilla's use. It supports the concepts of cursors, tables, rows, cells, and schema information. It supports both relational data and more general attribute value lists, as well as references between rows and tables so that a row can exist in several tables at once. It is not multiuser or multicursor and has no transactions and no recovery process. It is a general-purpose file format for self-referential data and has a low-level format equivalent to the basic structure of RDF. It has an XPCOM interface, but no XPIDL definition, and therefore no matching type library. No type library means it is not available from JavaScript.

On the relational database side, the default version of Mozilla has no direct access to relational databases. Access can be added, though. Table 16.4 lists the options both available and forecast during the release of version 1.4.

Finally, the platform has some application-specific flat file formats. These file formats are all read and written indirectly using high-level interfaces. Their location is fixed inside the current user profile. Table 16.5 lists these databases.

The cookies and bookmarks files are written entirely each time a change is made. The address book file is partially written each time a change is made.

"`Mork`" is a simple flat file database built using the `Mdb` technology, which was described in the discussion of Table 16.3. It provides a set of specifically formatted data structures suitable for storing address book cards and for status information about the user's current email and newsgroup windows. Like `Mdb`, `Mork` is not accessible from JavaScript. You can see `Mork` database content by viewing email and news control files with a text editor. Those files have `.mdb` suffixes.

Table 16.4 RDBMS add-ons for Mozilla

Database	Installation method	Cross platform?	Notes
MySQL	Downloadable XPInstall package	Yes	See *mysqlxpcom.mozdev.org*
MySQL	Recompile Mozilla 1.5+	Linux/ UNIX	See *www.mozilla.org/projects/sql*
PostgresSQL	Recompile Mozilla 1.3+	Linux/ UNIX	See *www.mozilla.org/projects/sql*
Protozilla	Downloadable XPInstall package	Linux/ UNIX	Adds the ability to extend Mozilla's network protocol support to other databases. See *http://protozilla.mozdev.org*
Web-based	Make a Web server available	Yes	Standard Web-based solution for database access; use HTTP GET and POST requests to manage a database client behind the server

Table 16.5 Application-specific flat file databases in Mozilla

File format	Notes
Cookies file	
Bookmarks file	
Address book	Uses Mork
News server control file	Uses Mork
Newsgroup hostinfo	Standard format newgroup control file
Email server control file	Uses Mork
Email folder	Standard UNIX mail(1) format mail folder
Mork	See following discussion in text

The `Mork/Mdb` file format is a disk-based format. The somewhat equivalent memory-based format is an RDF fact store, the best example being the `in-memory-datasource` data source.

The other database-like features in the Mozilla Platform are caches: the Web cache, which caches documents of remote origin, and the fastload (or XUL) cache, which holds chrome files that make up displayed windows.

16.2.4 Environment

The environment of the currently running process can be retrieved one variable at a time using this component and interface:

```
@mozilla.org/process/util;1 interface nsIProcess
```

The `nsIProcess` interface has a method `getEnvironment()` that returns the value for a supplied variable name. Supplied variable names are converted from Unicode to 8-bit extended ASCII, and values returned are Unicode values converted from the 8-bit extended ASCII value retrieved.

The operating system type or version can be detected without use of the environment. Just examine the `window.navigator.userAgent` property.

Versions of Mozilla built with debug support require that the environment variable `MOZILLA_FIVE_HOME` be set to the directory that the Mozilla binaries are installed in.

There is no environment variable that specifies the location of the current user's profile (or any profile). Setting such a variable by hand requires guesswork or foreknowledge since profile names are encrypted. The profile directory can be found using a well-known alias. See "File System Directory."

A number of debug environment variables are available when the Mozilla build is compiled with `--debug-enabled`. To understand their use, the source code for the platform needs to be examined closely. Their output is often too cryptic for general use.

16.2.5 Files and Folders

This topic describes how to locate and specify files and folders on the computer on which the platform is installed. The term *folder* is used for a file system directory because in Mozilla, the term *directory* means directory service.

Handling files in Mozilla is made complex by portability constraints and Web standards. The XPCOM objects used to represent files must be portable across all operating systems (or at least UNIX, Microsoft Windows, and Macintosh), and the idea of a file or folder must somehow interoperate with the idea of a URL.

The need for portability affects the names used for files and folders. The Mozilla Platform does not have a concept of full path name because operating system paths are expressed in different syntaxes and because volume-oriented operating systems like MacOS don't have paths at all. There are a number of volume-oriented operating systems still. Even the pathless file name part of a file identifier can be problematic. Mozilla's response to these constraints is to keep explicit use of paths and file names at arm's length, except in cases where portability is not critical.

This arm's length approach is implemented with the `nsIFile` interface. An object with that interface is frequently used in scripts, but it is rarely specified or inspected directly. Whatever naming information is available inside the `nsIFile` object stays there. Only in nonportable cases is the naming infor-

mation manipulated directly. That means the `nsIFile` object is only rarely created directly with its standard XPCOM pair:

```
@mozilla.org/file/local;1 nsIFile
```

Instead, objects with this interface are produced indirectly, using methods belonging to other interfaces. Files that are stored locally are represented by objects with a specialized `nsIFile` interface. The XPCOM pair responsible for that specialized interface is

```
@mozilla.org/file/local;1 nsILocalFile
```

Both interfaces apply to folders as well as files. There is also an older interface for files. It is deprecated (unfashionable) and shouldn't be used at all:

```
@mozilla.org/filespec;1 nsIFileSpec
```

The application programmer therefore relies on that object being manufactured, initialized, and returned by a method of some other interface. Overall, there are many ways to bring such an object into existence:

1. A directory service that retains a list of well-known files and folders can be consulted if the nature of the needed file or folder is known in advance.
2. Users can be prompted to identify the file or folder. Whatever they pick is the file or folder required.
3. The file or folder can be deduced from a URL. This does not automatically work; the context of the URL and file are both important.
4. The file or folder can be deduced from another file or folder object, if that other object is somehow related. Finding the parent directory of a given file requires only `nsIFile`'s parent property; finding the contents of a folder requires only `nsIFile`'s `directoryEntries` property.
5. If portability is not an issue, then the file's full or partial UNIX or Windows path name can be specified as a JavaScript string and an object initialized with that string.
6. Finally, if an `nsIFile` object is used to create a stream, channel, or other XPCOM object, then that other object can usually reveal the originating `nsIFile` object at any later point.

Examples of these dot-points are given in "File System Directory."

The `nsIFile` interface addresses portability issues, but the issue of integration with Web URLs remains. URLs are maintained in objects with the `nsIURL` interface. See "Web Scripting" for details on that interface. Files and URLs can be converted to each other using this XPCOM pair:

```
@mozilla.org/network/protocol;1?name=file nsIFileProtocolHandler
```

This interface provides `newFileURI()` and `getFileFromURLSpec()` methods that do the required conversion. `newFileURI()` is also available on the `nsIIOService` interface.

The nsIURL and nsIFile interfaces also allow a URI or file specification to be dissected into its component parts. Those parts can be read or updated. Simple string operations may be sufficient to turn the naming information inside an nsIFile object into an nsIURL object and vice versa. Files and URLs can therefore also be connected by application-specific string manipulation code.

16.2.5.1 Using the File System Directory The file system directory is described in "Platform Configuration," where some further examples are given. This piece of code gets by the shortest route a folder suitable for holding temporary files. Listing 16.1 illustrates.

Listing 16.1 Directory specification of an nsILocalFile object.

```
var Cc = Components.classes;
var Ci = Components.interfaces;
var dp = Cc["@mozilla.org/file/directory_service;1"];
    dp = dp.createInstance(Ci.nsIDirectoryServiceProvider);

var folder = dp.getFile("TmpD", {});
```

This approach reveals only those files and folders that already have a nominated purpose for the Mozilla Platform. The code revolves around the special folder alias TmpD. It must be looked up in a table. Those tables are compiled in "File System Directory."

16.2.5.2 Using the End User To create an nsILocalFile object, one solution is to ask the user as Listing 16.2 shows.

Listing 16.2 User-specification of an nsILocalFile object.

```
var file;
var CcFP = Components.classes["@mozilla.org/filepicker;1"];
var CiFP = Components.interfaces.nsIFilePicker;
var fp = CcFP.createInstance(CiFP);

// use whatever nsIFilePicker options are suitable
fp.init(window, "File to Read", Picker.modeOpen);

if ( fp.show() != fp.returnCancel )
  file = fp.file;
```

The nsIFilePicker object manufactures an nsILocalFile object in response to the user's selection.

If the object created holds a folder name, it can be converted to a more specific file or folder using the appendRelativePath() method, which accepts relative path names that do not include "..". The appended path can, with care, be hand-constructed without the loss of portability. Fortu-

nately, all of Microsoft Windows, UNIX, and Macintosh support the forward slash (/) character as a path separator for subpaths, even though a Microsoft Windows DOS Box does not. Be aware that some Windows users (and some Linux users in rare cases) may not have Long File Name (LFN) support, which limits file names to "8.3" or 14-character names. The `nsIFile` interface supports a number of attributes and methods that assist with portable path construction.

Finally, if the application does not need to be portable, or if separate implementation code for each supported platform are permitted, then an `nsILocalFile` object can be initialized directly from a string using the `initWithPath()` method. Backslash characters used in Microsoft Windows paths should be escaped (\\) or forward slashes used instead. Separate implementation code can be as simple as a series of `if` statements that test the current platform.

16.2.5.3 Using a Literal or a URL

If the application does not need to be portable, or if separate implementation code for each supported platform is maintained, then an `nsILocalFile` object can be initialized directly from a string. A suitable piece of code is simply as shown in Listing 16.3.

Listing 16.3 Literal specification of an `nsILocalFile` object.

```
var file;
var CcLF = Components.classes["@mozilla.org/local/file;1"];
var CiLF = Components.interfaces.nsILocalFile;
var file = CcLF.createInstance(CiLF);

file.initWithPath("C:\\WINDOWS\NOTEPAD.EXE");
```

The literal "`C:`" could be replaced with a portable file system root folder using the DrvD alias and the file system directory. Recall that Microsoft Windows supports both forward slashes and backslashes as path separators, so those too could be made more portable.

Alternately, the location of a local file might be contained inside a URL. The portable way to achieve conversion is with this code:

```
var conv = Cc["@mozilla.org/network/protocol;1?name=file"];
conv = conv.createInstance(Ci.nsIFileProtocolHandler);

var url = ... // Some nsIURL that already exists

var file = conv.getFileFromURLSpec(url);
```

That URL will either be prefixed with `file:`, or it will refer to a virtual directory behind a local Web server. In either case, the URI scheme may be chopped off and replaced with a different file system root. For example,

```
file.initWithPath(myURL.filePath.replace(/\|/,":"));
```

Here, `myURL` is an `nsIURL` object. The `replace()` regular expression operation changes a URL fragment like "`C|/test`" to "`C:/test`". Beware that under Microsoft Windows, network-mapped paths (UNC paths) like `\\saturn\tmp` require a prefix of five forward slashes (`http://///saturn/tmp`) when expressed as URLs, and that convention is not finalized at this time.

16.2.5.4 Working on Files and Folders

After a file is located and represented internally by an object, it's usual to read, write, or manipulate that file.

File descriptors, file pointers, and file handles are not available in the Mozilla JavaScript environment. That means that file-descriptor-based pipes cannot be created. The `nsIPipe` interface creates an application-level pipe, not a traditional UNIX pipe. The platform cannot be used to create named pipes (or symbolic links), but such pipes can be read from and written to if they already exist. In short, direct low-level file access is not the way to go.

Instead of file handles, Mozilla uses objects. Instead of manipulating a single object, a script must manipulate at least two objects. One object identifies the file or folder that is to be used—an `nsIFile` or `nsILocalFile` object. This object is a file *name* specifier. The other object is a stream that data are read from and written to. This object is a file *content* specifier. Both objects must be created, and this must be done in such a way that they are connected to each other. "Data Transfer" in this chapter discusses streams and other content specifiers extensively. Streams are similar to Java or C++ streams, but there is no operator overloading at work.

16.2.5.5 Working on ZIP and JAR Archives

This XPCOM pair uses `nsIFile` concepts and provides access to the contents of locally stored `.zip` and `.jar` files:

```
@mozilla.org/libjar/zip-reader;1 nsIZipReader
```

Zip files can also be created with this interface.

The stream converters noted in "Stream Content Conversion" can be used to work with a stream of compressed characters in their raw form.

16.2.6 Interrupts and Signals

There is no way to catch or send operating system signals from JavaScript. To catch signals, an XPCOM component must be written in either Java or C/C++.

The `nsIThread` XPCOM interface is used to manage a piece of code that can be interrupted. This will work only if the code to be interrupted is not JavaScript. The JavaScript interpreter in the Mozilla Platform is single-threaded and can't interrupt itself. This means that thread-based interrupts are not usable for entirely scripted applications.

The event-oriented technologies described in Chapter 6, Events, and the command system of Chapter 9, Commands, are the main alternatives to interrupts.

16.2.7 Network Protocols

Mozilla provides support for well-known network application protocols like FTP. Mozilla assumes the underlying transport layer will be TCP/IP. Other protocols, such as RS232, X.25, or TP4 must be overlayed with TCP/IP before they are usable. Mozilla supports the following low-level protocols:

- **TCP/IP v4** and **v6**. v6 is not enabled in the default build and must be compiled in with `--enable-ipv6`. The Configuration windows in Classic Mozilla do not yet support IPv6.
- **DNS**. The platform supports multithreaded (parallel) DNS look-ups.
- **FTP**. Mozilla supports FTP, but the Download Manager does not support FTP resume as of version 1.4.
- **RPC**. Mozilla provides support for RPC over XML only, not RPC over NDR/XDR. The latter approach is the traditional method of doing RPC.
- **SSL (Secure Sockets Layer)** and **SOCKS**. Mozilla supports SSL versions 2 and 3, and SOCKS 4.0 and 5.0. SSL is used for Secure SMTP (SMIME) and Secure HTTP (HTTPS) protocol support.

Protocols are used indirectly in Mozilla. Resources are identified by URL and the access method that prefixes the URL (like *http:*) determines the protocol used. The protocol is then exploited automatically by the platform. In general, a channel object accepts a URL, and everything "just works" from then on. Nevertheless, individual protocols do exist as separate objects and can be created as instances of this XPCOM pair:

```
@mozilla.org/network/protocol;1?name={x} nsIProtocolHandler
```

In this component name, `{x}` should be replaced with a value like `ftp` or `http`. A dump of the `window.Components.classes` array reveals all the protocols (actually URL schemes) for which Mozilla has components.

Mozilla can be configured at the IP port level using preferences. Individual ports can be enabled or disabled. Enabling such ports has no effect unless they are enabled in the operating system as well. Enabled ports can create security holes at the application level and are not recommended unless suitable firewalls are in place. See the preferences displayed by the URL `about:config` that start with `network` for an extensive list of configuration options.

Application programmers also have access to sockets. Operating systems represent a TCP/IP connection with a socket library that maps the TCP/IP connection to a file descriptor. Mozilla wraps the connection and descriptor details up into an object. Such a socket object is the lowest level network data structure available in the default build of the platform.

Finally, the Optimoz project, documented at *www.mozdev.org*, can be used to extend the network support of Mozilla. Using Optimoz, new protocols that are written in JavaScript alone can be added. The requirements for such

protocols follow: They must be built on top of a TCP/IP socket; they must be tolerant of small time delays; they must implement the `nsIProtocolHandler` interface; and the JavaScript code must register a full XPCOM component implementation for the created handler.

We now turn to specific low-level network-oriented tasks. For application-level communications, see "Data Transfer" and "Web Scripting."

16.2.7.1 How to Find an IP Address

To find the IP address for a given domain name, use this XPCOM pair:

```
@mozilla.org/network/dns-service;1 interface nsIDNSService
```

An object so created returns the IP address for a given domain name, or the current host, as a string like "`192.168.1.10`". Resolving domain names is slow, so if you don't want the application to freeze, use the `lookup()` method, which requires a listener object with the `nsIDNSListener` interface. The request will then proceed asynchronously. Make this listener out of a pure JavaScript object.

16.2.7.2 How to Create a Socket

To create a socket connection, several steps are required.

To use a socket you must ultimately create an `nsITransport` object. After you have this object, you can ignore the socket to a degree and, instead, use the techniques described in "Data Transfer." Socket access is quite abstract and high level; there is no `ioctl(2)` API that can be used to configure socket options.

To get this `nsITransport` object, you must first deal with the possibility that there is a network proxy between the platform and the computer at the other end of the socket. Create an object that is an `nsIProxyInfo` object for the desired remote address if you aren't sure whether a proxy exists. An `nsIProxyInfo` object, can be created using the `newProxyInfo()` or `examineForProxy()` methods of this XPCOM pair:

```
@mozilla.org/network/protocol-proxy-service; nsIProtocolProxyService
```

Next, using the resulting `nsIProxyInfo` object, or `null` if it is certain no proxy exists, create the factory object that is responsible for creating `nsITransport` objects for sockets. That factory object is created with this XPCOM pair:

```
@mozilla.org/network/socket-transport-service;1 nsISocketTransportService
```

With that factory object, create an `nsITransport` object by passing the `nsIProxyInfo` object to the `createTransport()` method. The resulting object will also support the `nsISocketTransport` interface, which is the base socket. If a SOCKS socket is wanted, use the `createTransportOfType()` method instead and choose a type of "`socks`" for SOCKS 5.0

or "socks4" for SOCKS 4.0. A UNIX (IPC) socket can be created by specifying a type of "ipc". The final transport object thus created can be used as a socket or as any transport object can.

SOCKS-enabled sockets use digital encryption and can be created only if correct encryption modules and keys are installed and enabled via the preferences. These items are installed and enabled by default in the user profile of the Classic Browser and the Mozilla Browser.

The Mozilla Platform has a number of other socket-related interfaces, but none of them is available from JavaScript. When looking through socket XPIDL files, remember to check for [noscript] before the interface name's declaration. Such interfaces are not available to JavaScript.

Listing 16.4 is a simple Perl program that can be used as a test server for socket connections. It reads from all connecting clients and prints what it gets to stdout. It does not support SOCKS, nor does it write data back to the socket client.

Listing 16.4 Socket server implementation for testing.

```
use IO::Socket;

my ($server, $client, $host);

$server = IO::Socket::INET->new(
    Proto => 'tcp', LocalPort => 80, Listen => SOMAXCONN, Reuse => 1);

while ($server && ($client = $server->accept()))
{
  while ( <$client> ) { print; }
  close $client;
}
```

In order to work, this program also requires correct port setup at the operating system level.

16.2.7.3 How to Create an FTP Session

The Mozilla Platform does not directly support a whole FTP session. Each URL request made via an nsIChannel object is a standalone operation. This means that each FTP session consists of at most four FTP commands. In pseudo-code, these are

```
open {hostname and port}
cd   {directory}
dir OR get {file}
close
```

This FTP session is handled inside Mozilla. The FTP session information is not available to the application programmer, and the application programmer can't submit individual FTP commands. This means that the way to conduct an FTP session is to use a URL request that happens to be an ftp: URL. See "Downloading Files" and "Channels" for detailed instructions.

If the application code needs to walk through the FTP hierarchy of an FTP site, then there is still a way to do that. An FTP URL can represent an FTP directory rather than a single file. If that URL is submitted from the platform, the directory listing is returned, but it is converted to an HTML document. This returned document can be walked through to discover files and subdirectories in the original URL directory. Those files can then be retrieved in turn.

To upload (put) a file to an FTP server, the same environment applies. The FTP session is hidden behind a URL. See "Uploading and Posting Files" for how to do this.

If all else fails, two sockets can be set up from JavaScript that implement the FTP protocol directly. If this is done, care needs to be taken to ensure that performance is adequate. This approach is probably as much work as creating a new XPCOM component for FTP in C/C++.

16.2.8 Processes and Threads

Your task may not require a whole thread or process. It may only need to be scheduled as an event on an event queue. If that is the case, see Chapter 6, Events. For larger tasks, read on.

The simplest way to run a separate program is to hand a file to the desktop of the operating system and ask the desktop to activate that file as though it were invoked (usually double-clicked) by the user. Start by using this XPCOM pair to create a file object:

```
@mozilla.org/file/local;1 nsILocalFile
```

Associate the resulting object with a real file (see "File System Directory" for how) and then call the `launch()` method. On UNIX, `launch()` is handled by the GNOME desktop, not by the PATH environment variable. There is no way to shut down that launched application from Mozilla.

More generally, processes can be launched from the platform using this pair of XPCOM objects:

```
@mozilla.org/process/util;1 nsIProcess
```

Be aware that this interface is not yet fully implemented on all platforms. To use it successfully, proceed as follows. As before, create an `nsILocalFile` object associated with the required executable. Because processes are generally operating system–dependent, the non-portable `initWithPath()` method may be used. Pass that object to the `nsIProcess` `init()` method, and then call `run()` to create the process. That last step requires a method invocation like this:

```
var blocking = true;
var argv =     ["arg1","arg2"];
var result =   {};
nsIProcess_object.run(blocking, argv, argv.length, result);
```

The `result` object is required by the `run()` method; it receives a `value` property that is set to 0 (zero) if the process starts successfully. If blocking is set to `true`, then Mozilla will freeze while the process runs. In that case, windows will not be updated for any reason until the process ends. If set to `false`, Mozilla continues processing. In either case, when the process finishes, the `exitValue` property on the `nsIProcess` object will be set. Some testing is required to match the `exitValue` value to normal exit values returned by the operating system.

Threads are a more difficult matter than processes. To an application programmer, a thread is no more than a piece of code scheduled with `window.setTimeout()`. Although this creates the illusion of a second flow of control, in fact the scheduled code is queued up until the current flow of control (the current piece of script) ends. No script is ever started until the existing running script is complete.

Matters are like this because of the way the JavaScript interpreter (SpiderMonkey) is connected to Mozilla. Deep inside, the platform does support threads. Its XPCOM system has a threading system that is a simplification of Microsoft COM's threading system. This system is used in a number of places; most obviously to manage FTP connections, which require the separate monitoring of two connections. The JavaScript interpreter is embedded in Mozilla using just one of these threads. While the interpreter is built to handle multiple running instances, its use in the Mozilla Platform does not take advantage of that feature.

Even though true threads are not available in JavaScript, interfaces do exist for working with threads. They provide a neater way to organize chunks of code than `setTimeout()` and `setInterval()` provide. Listing 16.5 shows the steps required to create a thread:

Listing 16.5 Simple thread creation code.

```
var Cct = Components.classes["@mozilla.org/thread;1"];
var Cit = Components.interfaces.nsIThread;

var thread = { Run : function ()
                     { alert(this.foo+" thread underway"); }
               foo : "bar"
             };

var mgr = Cct.createInstance(Cit);

mgr.init(code, 0, Cit.PRIORITY_NORMAL, Cit.SCOPE_GLOBAL,
         Cit.STATE_JOINABLE);

mgr.join();
alert("thread created");
```

The code object supports the `nsIRunnable` interface and contains the code to run (the `Run()` method) and any other properties that might be useful to that

code. The `thread` object holds the implementation of the thread, which includes the code and some data. The `mgr` (manager) object holds the `thread`'s configuration and state. The `join()` method tells the platform threading system that this thread should be scheduled to run—or to continue running if it was previously interrupted. Such an interruption is not possible if the thread is written in JavaScript. `join()` is not equivalent to `eval()`—the thread is put on a queue that the JavaScript interpreter will get to eventually. Because the interpreter is single-threaded and not interruptible by other threads, the `alert()` in the last line of code will always appear before the `join`ed thread's alert.

There is no race condition in this arrangement—the current script must finish before anything else can run. This single-threaded arrangement means that it is not possible to create blocking or infinite loop threads whose code is written in JavaScript. Such threads may never end, and therefore no other threads will ever run. The only value, then, of using such a system from JavaScript is as a modeling strategy that collects code into handy thread objects or that works with non-JavaScript XPCOM objects that also have the `nsIRunnable` interface.

A JavaScript script can create genuine threads by interacting with Java. Such threads are pure Java threads only.

16.3 DATA TRANSFER

This topic describes the general-purpose structures used to read, write, and process content inside the platform. It picks up where locating files and folders left off. RDF fact processing is also covered, but parsing of XML documents is left to "Web Scripting."

16.3.1 Content Processing Concepts

A major task of the back half of Mozilla is to process content and data. To do this, the platform must have systems that can transfer data and content from place to place. Mozilla has many content and data processing concepts to pick from.

Chapter 6, Events, describes the concepts of listeners, observers, and broadcasters. Those concepts are event oriented and are used only for tiny pieces of information. They are not sufficient for a content-oriented system, although such a system might work better if observers or listeners are added. A content-oriented system must transfer information as large as a document in size.

Mozilla's content-processing concepts are files, folders, streams, sessions, channels, transports, sources (data sources), and sinks.

Files and *folders* are concepts common to all operating systems and are discussed in "General-Purpose Scripting."

A *stream* is Mozilla's lowest level way to transfer data. A stream works on a series of bytes, octets, or characters, just like a C++ or Java stream does, or like redirection (>) in a UNIX or DOS shell. Streams can be read or written,

depending on their source and destination. Mozilla also supports Unicode streams, where the fundamental character is a two-byte "wide character" rather than a one-byte "extended ASCII" character.

A *session* is a set of configuration information about a process, task, or activity that is underway. This configuration information is used by the process itself, although it sometimes receives events. It, however, is not the actual process at work; it is an onlooker, a specifier, and a controller.

An example session is an FTP file transfer. The FTP domain name, user name, password, and socket connection are part of the FTP session information. The actual transfer of the file data is done by some other piece of software. The session merely says who, what, where, and when. A session can also be used outside of the networking domain—a Mozilla drag-and-drop mouse gesture has a session object that tracks the progress of the gesture. Sessions are used in a number of places in Mozilla.

A *channel* is the piece of platform architecture that actually performs data transfer. A channel is a sophisticated version of a buffer. In Mozilla, channels are used mostly for retrieving documents that have a URL. Channels sometimes have extensive functionality, but their basic task is fairly simple: They make available a flow of raw data that originates elsewhere, such as behind a Web server. The data transfer might consist of actually copying data, or it might be a conceptual transfer in which the data, once available in form A, is now available in form B. A channel has a management role—actual read and write of data to the channel is usually handed off to a stream.

A *transport* is a piece of networking. While a channel provides a high-level concept for data transfer, a transport does all the leg work by implementing or using a specific network protocol like SMTP or socket-based TCP/IP.

Sources and *sinks* are treated separately next.

16.3.1.1 Sources and Sinks

Source and sinks are important concepts in Mozilla. They are used mostly for the wholesale processing of XML documents. They are the highest level information flow concepts in Mozilla and usually do content-specific processing.

Sources and sinks are also a general design concept in both science and engineering. Many programmers are introduced to sources and sinks when dataflow diagrams are taught. In dataflow diagrams (and in an ordinary kitchen sink), it is generally understood that data or content (or water) starts at the source (the tap) and ends at the sink (the drain hole). That, however, is just a matter of perspective. That example can be turned inside out.

If you are responsible for the kitchen sink itself, then matters are arranged in the standard way. You can see water leaving the tap (the source) and entering the drain hole (the sink).

If you are responsible for guiding the water from the tap to the drain hole, then matters are inside out. The first thing you do is collect the water from the tap into some intermediate place, like a jug. The jug is then a sink. The last thing you do is pour the water out from that intermediate place into

the drain hole. The jug is then a source. In this example, your data-transfer system starts with a sink and ends with a source, not the other way around.

This second perspective is relevant to Mozilla. Inside the platform, a sink is used to suck up the content of a document into main memory. If any of the content needs to be extracted, then a source is used to get it out.

No matter which way sources and sinks are arranged, in producer-consumer terms, a source is always a producer, and a sink is always a consumer. From the application programmer's perspective, if the document content is not yet available (perhaps because it is stored in an external file), then a sink must be set up first to load that content. A source is set up second, to retrieve the loaded content for application use. The application programmer's job is to set up the water flow. In some cases, the document is automatically loaded. In that case, only a source is required.

This type of arrangement has one complexity. Documents can be modified. Sinks are generally used for the initial loading of data, and so they process data one way. This means that it is up to a source to manage changes. Therefore, sources not only retrieve the document's content but generally can modify it as well.

Mozilla does not provide a general-purpose sink or source interface. There are only specialist interfaces for particular kinds of data. Mozilla's interfaces allow more than one sink or source to operate on an in-memory document at the same time.

16.3.1.2 Specialized Sources and Sinks

Sources and sinks operate as content processors and are high-level concepts. Mozilla's sources and sinks are all specialized to a particular purpose.

Data sources are used to process RDF content. Instead of working with tags, parser tokens, or DOM objects, data sources work with RDF facts. The template system of XUL uses RDF data sources extensively, and these templates and sources can be manipulated with scripts. There are no RDF fact sinks, only fact sources. These fact sources (data sources) can also perform insert, update, and delete operations on the fact store that holds the retrieved-to-memory RDF document. Some of Mozilla's data sources, called *internal data sources*, draw their content directly from the platform, rather than from an external RDF document. The origins of the facts in this case are usually data structures inside the platform, or the user's bookmark file.

A *parser* is another kind of sink. It takes a flow of content, usually originating in a document, and transforms it into a data structure. An example is a parser that reads XML and creates a DOM tree. Mozilla includes parsers for all applications of XML that it understands.

A *serializer* is a source that is the inverse of a parser. It turns a data structure into a stream of flat content, typically an XML document.

16.3.1.3 Content Processing Architecture

All the data processing concepts discussed here fit together into an informal set of layers, although this layer-

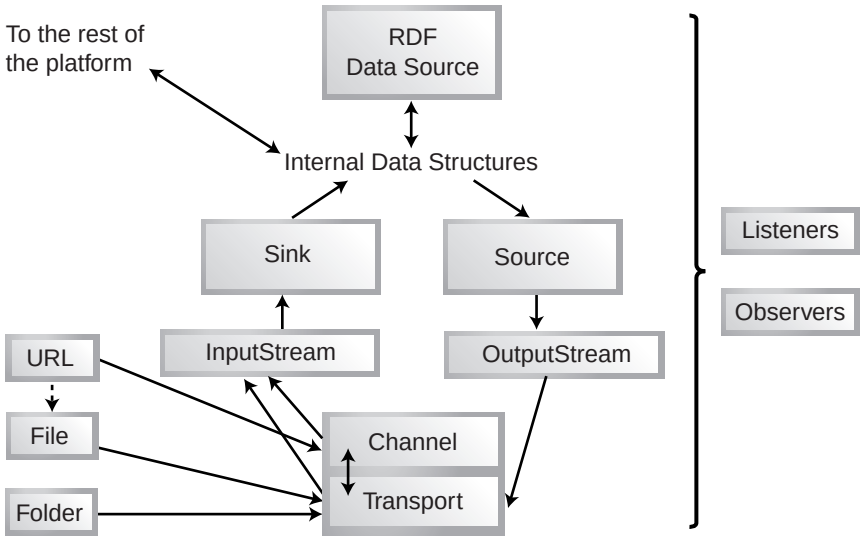


Fig. 16.1 Layered structure of content processing concepts.

ing is not as structured as a network protocol stack. Figure 16.1 illustrates these layered relationships.

Figure 16.1 is a conceptual diagram and does not show strict object relationships. It is, however, close to some specific object relationships inside the platform. We can see that channels and transports are closely connected, but not so closely connected that they can't be used separately. Input and output streams expose the data or content processed to the rest of the platform. Two typically large chunks of processing are the sources and sinks that contain parsers, serializers, and other transformation tools. RDF data sources are somewhat separate from the other concepts because they are very high level and work only on content already broken down into facts. All these concepts support listeners and observers; URLs and files are used to configure the lower levels of processing.

Not shown in Figure 16.1 are the many other interfaces provided by these central concepts, or the many other interactions that occur with the rest of the platform.

16.3.2 Streams

When a file or other information source is available, a stream must be created before that file's contents can be worked on. Streams are central to data processing in Mozilla, and there are a large number of stream-oriented interfaces available. These interfaces include stream creators, loaders, converters, and

managers. Several specialized streams also exist such as random-access and string-based streams. There is a stream interface available for all common tasks—just look for any interface with `Stream` in its name.

To illustrate this flexibility, Listing 16.6 shows four methods of creating a stream. This stream is used to read a local file that is a sequence of bytes.

Listing 16.6 Stream creation by several methods.

```
var Cc = Components.classes;
var Ci = Components.interfaces;
var mode_bits = 0x01; // from nsIFileChannel
var perm_bits = 0;    // from Unix/Posix open(2)
var file_bits = 0;    // from nsIFileInputStream

var stream;
var file = ... // same as Listing 16-3 or 16-2

// [1] Created directly

stream = Cc["@mozilla.org/network/file-input-stream;1"];
stream = stream.createInstance(Ci.nsIFileInputStream);
stream.init(file, mode_bits, perm_bits, file_bits);

// [2] Created from a transport

var trans = Cc["@mozilla.org/network/stream-transport-service;1"];
trans = trans.getService(Ci.nsIStreamTransportService);
trans = trans.createInputTransport(stream,0,-1,true);
var stream2 = trans.openInputStream(0,-1,0);

// [3] Created from a channel

var channel = Cc["@mozilla.org/network/local-file-channel;1"]
channel = channel.createInstance(Ci.nsIFileChannel);
channel.init(file, mode_bits, perm_bits);
stream = channel.open();

// In all cases, work on the stream from JavaScript

var s2 = Cc["@mozilla.org/scriptableinputstream;1"];
s2 = s2.createInstance(Ci.nsIScriptableInputStream);
s2.init(stream);

var bytes = 100;
var content = null;
content = s2.read(bytes);
```

In all three cases, the `nsILocalFile` object created earlier is passed in as an initialization argument at some point.

☞ **Example 1**. The file is read and written directly using a stream. The stream interacts with the file synchronously unless special arrangements are made.

- **Example 2**. This example is a little odd because it both starts and ends with a stream. The transport object must be based on something, and in the absence of a network protocol, a stream is the only other alternative. That underlying protocol is used to retrieve the file's content. The stream handed back by the transport to variable `stream 2` is different from the stream supplied. The handed-back stream will generally have data available on demand because the transport does work to collect the received data ready for the next user request. By comparison, example 1 does work only when the user makes a request. The transport will also close down the "connection" to the file automatically when no more data are available.
- **Example 3**. A channel allows the file to be retrieved without any assumptions about the retrieval mechanism.

Finally, streams cannot be read or written from JavaScript automatically. This is a piece of design intended to keep streams efficient. Instead, the stream object must be wrapped up in a special object that supplies read/write operations. That is the point of the last few lines, which also read at most the first 100 bytes of the supplied file, less if the file is short.

Examples 1 and 2 can be used to perform file writing instead of reading with only small coding changes. Example 3 cannot be rewritten because channels work only one way. When writing content, the default output is single-byte characters. Any content supplied as UTF16 Unicode strings (like JavaScript strings) is truncated character by character down to the least significant byte. This means that default file output is single-byte extended ASCII. To output Unicode (usually in a UTF8 encoding), content conversion is required. That is discussed next. (see "Stream Content Conversion.")

16.3.2.1 Stream Content Conversion

All character strings are represented as Unicode inside the platform. Plain files may be read as raw binary data (use `nsIBinaryInputStream`), as 8-bit character data (the default for plain files), or as correctly encoded Unicode. The last option occurs when XML files are identified and parsed into DOM hierarchies, when a stream of data originates from a source that supplies format information such as HTTP or MIME, or when DTD files are read.

To convert the content produced by a stream, this XPCOM pair will do the job:

```
@mozilla.org/intl/scriptableunicodeconverter;1
     nsIScriptableUnicodeConverter
```

Mozilla also supports many components with Contract IDs of this form:

```
@mozilla.org/streamconv;1?from={mime1}to={mime2}
```

`mime1` and `mime2` are MIME types. These components support the `nsIStreamConverter` interface. Such an object reads a given input stream and

Table 16.6 Stream conversions supported by Mozilla

Original MIME type	Converted MIME type
application/http-index-format	text/html
application/mac-binhex40	*/*
application/x-unknown-content-type	*/*
compress	uncompressed content
deflate	uncompressed content
gzip	uncompressed content
message/rfc822	application/vnd.mozilla.xul+xml
message/rfc822	*/*
message/rfc822	text/html
multipart/byteranges	*/*
multipart/mixed	*/*
text/ftp-dir	application/http-index-format
text/gopher-dir	application/http-index-format
text/plain	text/html
x-compress	uncompressed content
x-gzip	uncompressed content

converts its content. It makes available a new input stream that the converted content can be read from. Table 16.6 lists the conversions that the platform supplies. Such a converter can also be implemented in pure JavaScript.

The XPIDL description for `nsIStreamConverter` explains how such a conversion can be done with two `nsIStreamListener` objects instead of whole stream objects. This approach allows converters to work on any kind of stream, not just input streams.

16.3.3 Transports

Transport layer XPCOM objects are responsible for transferring content from inside the Mozilla Platform to outside it, and vice versa. Transports are therefore more general than streams, which are restricted to the platform and to the local disk. Where streams generally provide data synchronously and directly from a given source, transports can provide data both asynchronously and synchronously, from anywhere. Transports may also handle and buffer up data between user requests.

The currently available transport layers are shown in Table 16.7.

Table 16.7 Supported XPCOM transport layers

Implementation	Interface
@mozilla.org/network/stream-transport-service;1	nsIStreamTransportService
@mozilla.org/network/socket-transport-service;1	nsISocketTransportService
@mozilla.org/network/storage-transport;1	nsITransport
@mozilla.org/xmlextras/soap/transport;1?protocol=http	nsISOAPTransport
@mozilla.org/xmlextras/soap/transport;1?protocol=https	nsISOAPTransport

The five transports in Table 16.7 are responsible for: all streams, including local files; domain sockets; the browser cache; an HTTP transport binding for SOAP requests; and a secure (SSL-based) HTTP transport binding for SOAP requests.

The `stream-transport-service` implementation is fairly new (since 1.3) and replaces the no longer available `file-transport-service`. Be aware that some examples of code may use that older object.

16.3.4 Channels

A channel is a read-only mechanism for getting the content of a URL. Although a channel can work with file objects, it is more natural for a channel to use a URL. Channels are responsible for much of the content-specific work that Mozilla does when retrieving a document. The sole exception to the read-only rule is an upload channel, which is used for submitting forms, uploading files, and publishing Web pages.

In the normal programming case, channels are handled indirectly. Like `nsIFile` objects and streams, it is far more common for a channel to be created for you than to create one explicitly yourself. Just as a file and a stream are a closely associated pair of objects, so too are a URL and a channel. This analogy is not perfect. A difference is that while streams can be created directly, channels rarely are because channels are mostly buried behind a protocol. A second difference is that a channel is an enhanced request (an `nsIRequest` object), which in turn is an enhanced URL. So a channel and its URL are not strictly separate objects.

Use of channels usually starts with this XPCOM pair:

```
@mozilla.org/network/io-service;1 nsIIOService
```

This component provides the `nsIIOService` interface via `getService()`. As for transports, this interface is effectively a name service for URL schemes. Recall that a URL scheme is the characters before the first colon in a fully quoted URL. This component takes scheme names and returns software objects. The `nsIIOService` interface is therefore the fundamental jumping off point for retrieving the content of a URL.

The `nsIIOService` interface can create new URI objects (with `nsIURI` or `nsIURL` interfaces). These objects describe a given URL just as `nsIFile` objects describe a file. These interfaces also expose the protocol handler objects for a given scheme or URL. Those protocol handler objects implement channels. Each protocol handler supports one type of channel; some support more than one type. After a URL object is constructed, either from user input or from a plain JavaScript string, it can be used to find the protocol handler for that URL, and from that found handler a useful channel object can be obtained. The short way to do all this is just to call `newChannelFromURI()`.

When the channel object is available, it can produce a stream object and start the required processing. The stream object is used to deal with the retrieved content. The `nsIIOService` interface has a number of convenience methods so that it is easy to ignore the protocol handler entirely.

Channels do a great deal of processing on behalf of the URL requestor—they locate and retrieve the resource; perform content conversion; and record the MIME type and other configuration details. Table 16.8 lists the channels provided by the platform.

From Table 16.8, most channels are associated with a URL scheme. All channels support the `nsIChannel` core functionality, which consists primarily

Table 16.8 Supported XPCOM channels

Channel interface	URL scheme and/or contract ID implementing
nsIChannel	All the following entries
nsICachingChannel	http:
nsIDataChannel	data:
nsIEncodedChannel	http:
nsIFileChannel	file:
nsIFTPChannel	ftp:
nsIHttpChannel	http:
nsIImapMockChannel	imap:
nsIInputStreamChannel	@mozilla.org/network/input-stream-channel;1
nsIJarChannel	jar:
nsIMultiPartChannel	internal use only
nsIResumableChannel	ftp: (http: not yet supported)
nsIUploadChannel	file:, ftp:, http:
nsIViewSourceChannel	view-source:
nsIWyciwygChannel	wyciwyg: (not spelled 'wysiwyg')

of the `open()` and `asyncOpen()` methods. Those methods yield stream or stream listener objects. The other channel interfaces merely enhance the channel object produced by `nsIIOService` with extra configuration information. They do not represent fundamentally different channels—they are just add-ons.

Table 16.8 presents a few unusual cases. The `nsIUploadChannel` is oriented away from the desktop rather than toward it. It consumes an input stream, rather than providing one, all in order to send the supplied content to some server. The `nsIResumableChannel` is used for FTP downloads that are interrupted. The Download Manager in the Classic Browser does not yet use this functionality.

The second unusual case is that of trivial protocols. A channel need not be associated with a complicated protocol like HTTP or FTP. It can be associated with any simple transfer task. The existing implemented channels support a trivial memory-to-disk transfer (plain file access) and an even more trivial memory-to-memory transfer (which uses a stream object). Neither of these trivial protocols needs to interact with URLs, and so their channel objects can be manipulated directly by hand. That is the origin of the `input-stream-channel` Contract ID that appears in Table 16.8.

Listing 16.6 has an example of the trivial memory-to-disk protocol at work.

16.3.5 Data Sources

Data sources provide the fact-oriented support needed for XUL templates and manipulation of RDF fact stores. The Mozilla Platform contains substantial code that hooks data source objects up to such URIs. For manipulation of fact stores, XPCOM interfaces are coded against directly, and again the platform has substantial preexisting code that does this. The most obvious example is the default builders and content views used to activate templates. Equivalent or different support can be built by an application programmer using these interfaces.

The concept of a data source is expressed by the `nsIRDFDataSource` interface. This interface provides all the semantics of dealing with RDF facts. About 20 XPCOM components implement this interface. The “Hands On” session in this chapter has extensive examples of these interfaces at work. This topic attempts to classify and identify those interfaces.

Individual facts can be constructed from simple XPCOM objects based in the `nsIRDFResource` and `nsIRDFLiteral` interfaces. In general, a data source is always readable and occasionally writable, providing simple query-insert-update-delete functionality, sometimes called get-and-set. Unlike other processing concepts, data sources operate on logical objects (facts) rather than a stream of bytes or characters

Useful XPCOM data source interfaces fall into three categories:

- **Helper tools and utilities**. These are needed just to make something happen.
- **Creative extensions**. Some interfaces extend the functionality provided by the basic `nsIRDFDataSource` interface in novel ways.
- **Content support**. Some aspects of data sources determine content—the type and number of facts that can be accessed. This final category is divided into ordinary and internal data sources. Ordinary data sources draw facts from RDF files. Internal data sources draw facts from the Mozilla Platform itself.

If the wrong data source from this final group is chosen, then hours, days, or weeks can be spent wondering why nothing works. It is therefore important to have good product knowledge on each kind of content support.

The `nsIRDFDataSource` makes poor use of the word *source*. In that interface, *source* and *target* are used to mean fact subject and fact object, respectively—for example, `getSource()`. Source is also used elsewhere in Mozilla to mean source code. Don't automatically assume that source means data source.

16.3.5.1 Factory and Helper Objects If a XUL template is involved, then the DOM objects for that template provide access to `nsIRDFDataSource` objects. If no template exists, then a directly scripted use of data sources requires use of factory objects right from the start. Table 16.9 lists the components used for pure RDF manipulation:

In Table 16.9, the notation `{arg}` indicates that a number of alternatives are possible. The easiest way to see them all is to list the contents of the `window.Components.classes` array. Table 16.9 is divided into four parts as follows.

The first part is the starting point for application use of RDF. The `nsIRDFService` interface is used to create `nsIRDFDataSource` objects from a URI, including a URL based on the `rdf:` scheme. Those URLs are listed in Table 16.11. If an object for a fact subject, predicate, or object is required, it can also be created from JavaScript strings using this interface. The `nsIRDFService` interface's object is accessible via `getService()`, not via `createInstance()`.

The second section of Table 16.9 provides factory interfaces for creating and manipulating RDF containers. The two interfaces supplied provide straightforward ways to create data structures consisting of RDF resource objects. This section also includes the `resource-factory` components. When a `resource-factory` component is used, the fact items created have extra features. Depending on the argument chosen for `name=`, such a created object might contain address book, email, news, or file information, in addition to the basic subject, predicate, or object resource. This information, and its associated methods, follows the resource around as it is manipulated in the fact

Table 16.9 XPCOM components specific to RDF

Component name	Interfaces	Purpose
@mozilla.org/rdf/rdf-service;1	nsIRDFService	Starting point; creates nsIRDFDataSource data sources and nsIRDFNode subjects, predicates, and objects
@mozilla.org/rdf/container;1	nsIRDFContainer	Creates a <Bag>, <Seq>, or <Alt> tag object
@mozilla.org/rdf/container-utils;1	nsIRDFContainerUtils	Manipulates an RDF container tag object
@mozilla.org/rdf/resource-factory;1?name={arg}	various	Creates objects representing parts of a fact
@mozilla.org/rdf/content-sink;1	nsIExpatSink	Turns RDF-based XML objects into a fact store
@mozilla.org/rdf/xml-parser;1	nsIRDFXMLParser	Turns an RDF document into RDF-based XML objects
@mozilla.org/rdf/xml-serializer;1	nsIRDFXMLSerializer nsIRDFXMLSource	Turns a fact store into an RDF document
@mozilla.org/rdf/delegate-factory;1?key={arg}&scheme={arg}	nsIRDFDelegateFactory	Ties a resource (a fact item) to a custom object that synchronizes something else against that resource

store. Only the components that relate to email are fully revealed by XPCOM; the others are entirely unavailable to scripts. The ones that are revealed have `name=` set to the following values:

```
imap mailbox news moz-abdirectory moz-abldapdirectory moz-
     abmdbdirectory moz-aboutlookdirectory
```

The third section of Table 16.9 is only of use for directly parsing RDF documents. That is a deep customization of Mozilla and not particularly required by ordinary applications. The reverse action, which is to generate an RDF document from a fact store, is required for any application that intends to persist the data it manipulates. None of these interfaces is used to manipulate facts.

Finally, the `nsIRDFDelegateFactory` item in Table 16.9 requires a deep understanding of the platform architecture. It provides a way to attach a side effect to the creation or destruction of a resource used in a fact. Using such a delegate ties a fact to whatever system the delegate is designed for.

This is like the tie function in Perl. A delegate is an observer for an individual resource object. Use of that interface is beyond the scope of this book.

16.3.5.2 Structural Options Mozilla provides options that extend the functionality of data sources. These options make data sources more flexible rather than providing further access to their content. These options exist in the form of XPCOM interfaces that add extra functionality to the fact store that holds the data in the data source. Table 16.10 describes these interfaces.

The XUL template system not only uses composite data sources but also supports a list of arguments supplied to the XUL attribute of the data sources. A composite data source is no more than a container that holds other data sources. It in turn implements the `nsIRDFDataSource` interface. Each method of that interface just searches the contained data sources and calls the same method on each one as required.

In-memory data sources are at the core of several of the more complex data sources. Where the content of a fact store needs to be constructed by hand, rather than sourced from some other place, an in-memory data source is the logical starting point. Purging such a data source is just a way of resetting it back to empty. Stopping propagation (preventing broadcast of changed facts) slightly improves performance.

The `nsIRDFRemoteDatasource` interface provides a way to save and load a fact store back to or from the place it originated. That place is typically an RDF file or something equivalent. The file can be local or remote, but there is very limited support in both cases. Save is implemented by the `Flush()` method, and load, by the `Refresh()` method. `Flush()` is only supported on `file:` URLs; `Refresh()` is only supported on `file:` and `http:` URLs.

16.3.5.3 Content Options Data sources are not all created equal. They differ in content and in access to that content. These differences are the main reason why data sources are hard to use. It is not obvious from an application programmer's perspective which data sources can produce what facts or how those facts might be retrieved. All data sources are based on XPCOM components that follow this naming pattern:

```
@mozilla.org/rdf/datasource;1?name={arg}
```

Legal values for `{arg}` are listed in Table 16.12, leftmost column.

The content that a data source provides to the application programmer is obvious only for ordinary RDF files. Those files can be read directly by eye, which lays them completely open. Internal data sources, on the other hand, cannot be viewed at all except by (a) using the data source, (b) studying the platform source code, or (c) finding an example in the user's profile. The bookmark, history, search and local-store internal data sources all drop example files in the user profile; the others do not. Even where there are example files, those files generally aren't RDF documents.

Table 16.10 Interfaces extending the features of `nsIRDFDataSource`

Interface name	Implemented by	Purpose
nsIRDFDataSource	All data sources, but see Table 16.12 for limitations	Fundamental data operations on the data source's fact store
nsIRDFCompositeDataSource	@mozilla.org/rdf/datasource;1?name=composite-datasource	Provides a data source that is a union of the facts in one or more other data sources; inserted facts go into the first of those other data sources
nsIRDFInMemoryDataSource	@mozilla.org./rdf/datasource;1?name=in-memory-datasource	Provides a data source based on a fact store that is independent of all other facts
nsIRDFPurgeableDataSource	@mozilla.org/rdf/datasource;1?name=in-memory-datasource	Allows a data source to be emptied of facts
nsIRDFPropagatableDataSource	@mozilla.org/rdf/datasource;1?name=in-memory-datasource @mozilla.org/browser/bookmarks-service;1 @mozilla.org/rdf/datasource;1?name=bookmarks	Turns on or turns off broadcasting of fact changes to any observers
nsIRDFRemoteDataSource	@mozilla.org/autocompleteSession;1?type=history @mozilla.org/browser/bookmarks-service;1 @mozilla.org/browser/global-history;1 @mozilla.org/rdf/datasource;1?name=bookmarks @mozilla.org/rdf/datasource;1?name=history @mozilla.org/rdf/datasource;1?name=xml-datasource	Provides a way to coordinate the fact store of a data source against the original source of the facts. The XML data source contract ID is for plain RDF files

This anonymity of content is a big problem for XUL templates and scripts that try to navigate an internal data source. In both cases, the structure of the data source needs to be known beforehand. Fortunately, such internal data sources are needed for a narrow class of uses only.

The "Debug Corner" in this chapter has some code that reveals a data source's content. Table 16.11 lists the top-most fact subject and common predicates for most of the internal data sources. The special value `rdf:null` stands for no data source at all. It does not stand for an empty data source.

Beyond content, many of Mozilla's data sources have limited or restricted functionality. This means that even if you know what the content of the data source is, the data source object may not have enough implementation to make that content accessible. This means that even though an XPCOM component may state that it supports the `nsIRDFDataSource` interface, in truth many of the methods in that interface may simply return with an error or exception, without doing anything. Such data source objects are yet to be finished.

Table 16.12 indicates the level of support that is available for each of the implemented data sources. Table 16.12 is based on Mozilla version 1.4 and should be used as an *indication only* of the available functionality. Less-well-known data sources require obscure preparation steps before they are useful. Such steps are not yet covered here.

Data sources that do not have an `rdf:` URI cannot be used in XUL templates using XML attributes. They may still be attached to a template with a script. Data sources that are not registered with XPCOM in the default build of the platform cannot be used from XUL or JavaScript at all. The `composite-datasource` can only be asserted into if one of its collected data sources can be inserted into. It is recommended that you work with the individual data source directly, rather than work indirectly through the composite data source's interface.

16.4 WEB SCRIPTING

Web browsers perform tasks that operate in an environment different from traditional 3GL programs. On the Web, there is no such thing as a file or a file name. Instead, there are URLs and the documents that represent those URL resources. Frequently, such documents have complex structure and are XML-based. Such an environment requires a different scripting approach to that described under "Files and Folders" and under "Streams."

Web browsers are also actors in the emerging Web protocols stack, which could be better identified as the XML protocol stack. This stack, a set of standards, uses HTTP (or another protocol) as the first step in a series of application-enabling and application-specific sets of protocols. The XML protocol stack provides a data-oriented transaction system, rather than the simple document request and response-with-retrieval system that Web surfing requires.

Table 16.11 Starting points for internal data sources

rdf: URI	Topmost URN / URI	Predicates used to contain facts
rdf:addressdirectory	moz-abdirectory://	http://home.netscape.com/NC-rdf#child and http://home.netscape.com/NC-rdf#CardChild
rdf:bookmarks	NC:BookmarksRoot NC:PersonalToolbarFolder	Uses RDF containers
rdf:charset-menu	Many (e.g., NC:BrowserCharsetMenuRoot)	Uses RDF containers
rdf:files	NC:FilesRoot	http://home.netscape.com/NC-rdf#child
rdf:history	NC:HistoryRoot NC:HistoryByDate	http://home.netscape.com/NC-rdf#child
rdf:httpindex	URL of index	
rdf:internetsearch	NC:SearchEngineRoot NC:LastSearchRoot NC:SearchResultsSitesRoot NC:FilterSearchUrlRoot NC:FilterSearchSitesRoot SearchCategoryRoot LastSearchMode	http://home.netscape.com/NC-rdf#child
rdf:ispdefaults		
rdf:local-store	None; use any URI	No containers; each URI has a set of properties only
rdf:localsearch	Every find: URI is a root	No containers; each URI has a set of properties only
rdf:mailnewsfolders	No root; use server URLs	http://home.netscape.com/NC-rdf#child
rdf:msgaccountmanager	msgaccounts:/	
rdf:msgfilters	No root; use server URLs	
rdf:smtp	NC:smtpservers	http://home.netscape.com/NC-rdf#child
rdf:subscribe	No root; use server URLs	http://home.netscape.com/NC-rdf#child
rdf:window-mediator	NC:WindowMediatorRoot	Uses RDF containers

Table 16.12 nsIRDFDataSource interface support for each data source

Name used in Contract ID	Has rdf: URI?	Registered with XPCOM in the default build?	Content-specific XPCOM interfaces?	Supports assert()?	Supports ArcLabelsOut()?	Supports GetAllResources() ?	Supports commands?
addressdirectory	✓	✓	✓	✓	✓		✓
bookmarks	✓	✓	✓	✓	✓	✓	
charset-menu	✓	✓	✓	✓	✓	✓	
files	✓	✓			✓		
history	✓	✓	✓		✓	✓	
httpindex	✓	✓	✓	✓	✓	✓	✓
internetsearch	✓	✓		✓	✓	✓	
ispdefaults	✓	✓					
local-store	✓	✓			✓	✓	✓
localsearch	✓	✓			✓		
mailnewsfolder	✓	✓	✓	✓	✓		✓
msgaccountmanager	✓	✓	✓		✓		
msgfilters	✓	✓			✓		
smtp	✓	✓			✓		
subscribe	✓	✓	✓		✓		
window-mediator	✓	✓	✓	✓	✓	✓	✓
chrome				✓	✓	✓	✓
mailsounds					✓		
registry					✓		
relatedlinks					✓	✓	✓
in-memory-datasource		✓		✓	✓	✓	
composite-datasource		✓		maybe	✓		✓
xml-datasource (RDF files)		✓	✓	file: URLs only	✓	✓	✓

This emerging Web protocols stack consists of a set of standards and standards-in-progress. From lowest to highest, the most central standards in the stack follow:

- **XML and XML Schema**. These standards are used to define the underlying syntax of all the other standards. Mozilla implements XML and has some XML Schema utilities.
- **HTTP**. This protocol can act as a "transport binding" that is used to send and receive Web protocol stack messages. Mozilla implements HTTP.
- **SOAP and XML-RPC**. SOAP provides an XML-based format for messages. It adds timing, naming, identifying, data packaging, and message-passing semantics to plain XML. It is to XML what traditional RPC (Remote Procedure Calls) are to C/C++. There are two message formats in the W3C SOAP specification: one that maps efficiently to traditional RPC and one that is a "pure XML" format that uses XML schema types. The former is sometimes called XML-RPC. Mozilla implements both standards.
- **WSDL**. Built on top of SOAP, WSDL is a module packaging and definition language. Client software can exploit it to analyze and use SOAP-based facilities provided by servers. Mozilla support for WSDL is available from version 1.4 onward.
- **UDDI**. UDDI is a protocol that provides a name-mapping service for Web services, just as DNS provides a name-mapping service for TCP/IP addresses. UDDI gives an XML client the capbility to dynamically discover Web services that it doesn't yet know about. Mozilla does not implement UDDI, but UDDI is built on top of SOAP and so can be simulated at the cost of extra programming effort.
- **ebXML (enterprise business XML)**. Defined by the OASIS organization (*www.oasis-open.org*) and the United Nations, ebXML adds business transactions and business identification on top of SOAP. Mozilla does not implement ebXML.
- **Business process modeling standards**. These standards are high-level aggregation and specification standards intended to solve both general and specific business intercommunication problems between software applications. An example of the organizations developing these standards are the Workflow Management Coalition and the Open Application Group. Mozilla does not implement any of these standards.

Mozilla may not ever implement all these standards because some are intended to be used business-to-business, rather than consumer-to-business. The Web protocol stack is somewhat separate from the typical uses of Web browsers—displaying HTML content and email messages. Instead of using a channel, which is the standard way to work with URLs, these Web protocol features have their own separate interfaces, which must be specially scripted.

A Mozilla application might have no GUI at all. For example, it might be based on the `xpcshell` tool. In that case, the application can exploit the advanced XML support in the platform. It can implement servers that provide and use Web protocol stack concepts. A simple example is a content router that sends received XML documents to different destinations, depending on what they contain.

16.4.1 URIs, URLs, and URNs

URIs, and the specialist subformats of URL and URN, are described in the IETF's RFC 2396. It is intended, or at least hoped, that URIs be media-independent and highly portable. They therefore do not have the portability problems that file and path names have. Instead, URIs (especially URLs) have another problem: They are often badly typed, aliased, or carelessly shortened by users.

In Mozilla, a URI can be represented as a plain JavaScript string without loss of portability. If this string is to be usable with other XPIDL interfaces, then it needs to be converted to an object. This XPCOM pair is the most fundamental object available:

```
@mozilla.org/network/simple-uri;1 nsIURI
```

URLs are a specific form of URI and have a specialist object available, one that caters to all the common URL schemes, like `http:` and `ftp:`. This XPCOM pair is a widely used example:

```
@mozilla.org/network/standard-url;1 nsIURL
```

This interface also supports `nsIURI`. In fact, most XPCOM components with `uri` or `url` in their Contract ID support one or both of these interfaces.

If the user enters a URI, some validation may well be required. There are several defenses against bad syntax. This XPCOM pair makes the broadest attempt to fix a user-entered URI:

```
@mozilla.org/docshell/urifixup;1 nsIURIFixup
```

This interface has a method `createFixupURI()`, which can deal with keywords entered as URIs and lazy shortened forms entered as *www.test.com* or even *test.com* instead of *http://www.test.com*. Such a `docshell` component is exposed as an AOM object in a Mozilla Browser window; that structure is discussed in Chapter 10, Windows and Panes, in "`<iframe>`."

A second solution for syntax problems is to rely on this XPCOM pair:

```
@mozilla.org/network/url-parser;1?auth=maybe nsIURLParser
```

This interface will parse a URL according to RFC 2396, but in a lenient way so that many small syntax mistakes are both accepted and corrected. The component subspecifier can also be quoted as `yes` or `no`. In those cases, there are slight variations on the parsing algorithm used.

Finally, the base interface, `nsIURI`, supports a method named `resolve()`. This method compares a supplied relative URI against the current object's URI and returns a fully resolved (unshortened) URI equivalent to the relative one supplied.

There is no XPCOM object specifically for URNs.

The ultimate test of a URL's correctness is, of course, to retrieve the resource that it locates.

16.4.2 Downloading Files

To download a file or document from a remote location, you can use a channel, a URI, and the resulting stream. That is the straightforward approach covered under "Content Processing Concepts." There is also a very high-level alternative, which is to use this XPCOM pair:

```
@mozilla.org/embedding/browser/nsWebBrowserPersist;1
     nsIWebBrowserPersist
```

This interface accepts a URI or a DOM 1 `Document` interface and an `nsILocalFile` object. It performs the whole fetch-and-save operation with a single method call.

To perform a download asynchronously so that other tasks can be attended to during the download, create a content listener or observer object in pure JavaScript. Most interfaces, like `nsIChannel`, describe which content listeners and observers are supported. Each time a chunk of downloaded document content appears in the listener or observer, you process it, save it, or ignore it.

The most common way such a process-by-piece object is built is to implement the `nsIWebProgressListener` interface. Any object that supports the `nsIWebProgress` interface can register such a listener (or more than one), and many other interfaces accept such a listener object as an initialization argument. There are existing XPCOM objects that implement this interface, so for many applications the object you need to get the job done already exists.

To be advised of the progress of an asynchronous download, there are numerous options. Receiving progress advice is in theory a problem separate from that of receiving content. Advice is information about progress, whereas received content is the result of that progress.

The most primitive advice option is to enhance an ordinary content listener so that progress is noted as each chunk of content arrives. This option does not report completion or management events associated with content delivery; it only reports forward progress.

A better tracking option is to create a pure JavaScript object with the `nsIProgressEventSink` interface and to supply it to the object responsible for the download. This sink (an event listener) reports all changes to the status of the in-progress download. An alternative is to create a pure JavaScript object with the `nsIRequestObserver` interface and lodge it with the channel

or transport object. Such an observer only notices the beginning and ending of the download, not cancellations or suspensions.

An even more sophisticated tracking approach is to use the XPCOM objects responsible for the Mozilla Download Manager. These objects can be used with or without the Download Manager dialog box, but if that dialog box is employed, lots of scripting is required to tie it to the XPCOM objects properly. With or without a dialog box, this approach is limited to downloading files to be saved to disk. Such a use starts with this XPCOM pair, which implements a service:

```
@mozilla.org/download-manager;1 nsIDownloadManager
```

This single object manages all downloads underway. The `addDownload()` method of the manager is used to create and register a new object with the `nsIDownload` interface for each file to be downloaded.

Through a subtle arrangement, each download object is responsible for informing the Download Manager about progress because it records all the configuration details of a single download operation. These details are specified in the arguments passed to `addDownload()` and include a way of disposing of (saving) the downloaded item—this is the final `nsIWebBrowserPersist` argument. If this argument exists, the download object keeps the Download Manager informed of progress automatically, and the Download Manager uses the download object to clean up if the download is canceled or otherwise interrupted. If this final argument does not exist, then cleanup is up to the application programmer. The need for cleanup can be detected by lodging an observer on the individual download objects.

Mozilla also supports the concept of a load group. This is a variation on the `nsIRequest` interface, which allows a collection of URIs to have group identity. A load group is useful if a summary of the progress of a collection of requests is needed.

16.4.3 File and MIME Types

The MIME type of a file, URI, or file extension can be found with this XPCOM pair, which is a singleton service object:

```
@mozilla.org/mime;1 nsIMIMEService
```

This object consults the MIME information stored in the Mozilla user profile first. If an answer is not available there, then the desktop's operating system is consulted. On UNIX, the type is determined by the GNOME desktop, not by `file(1)`.

The `launch()` method of `nsILocalFile` allows an executable to be run or a data file to be loaded into its application software. This can be done without the application programmer needing to know anything about the file's type.

16.4.4 Uploading and Posting Files

Documents can be posted by means of the AOM `XMLHttpRequest` object. That object is discussed in Chapter 7, Forms and Menus, in "Form Submission." It is based on this XPCOM pair:

```
@mozilla.org/xmlextras/xmlhttprequest;1 nsIJSXMLHttpRequest
```

Uploading of documents is equally easy, if not easier. Follow the approach described in "Channels" in this chapter, specifying the destination of the upload when using the `nsIIOService` interface. That destination will either be a server-side program, in the case of an HTTP POST operation, or an FTP directory, in the case of FTP. After the channel is created, use `QueryInterface()` to obtain the `nsIUploadChannel` interface and supply that interface with an input stream containing the file contents to be sent. To send the content, obtain the `nsIChannel` interface again, and call `open()` or `asyncOpen()` as for any channel object.

16.4.5 Web Protocol Stack Objects

The basis of the Web protocol stack, HTTP, is widely used in Mozilla, and can be scripted in many ways, including direct use of the `XMLHttpRequest` AOM object. The other protocols supported by Mozilla require specific objects separate from the rest of the platform.

A useful set of documentation on support for Web protocols is available at *www.mozilla.org/xmlextras/*.

XML-RPC support is the simplest step up from HTTP. This XPCOM pair:

```
@mozilla.org/xml-rpc/client;1 nsIXmlRpcClient
```

is responsible for creating an XML fragment containing the RPC request, submitting it synchronously or asynchronously to the supplied URL over HTTP, and reporting back results or faults. Faults appear as objects with the `nsIXmlRpcFault` interface.

In traditional RPC, a tool like `rpcgen(1)` is used to create C code that does much of the work. That C code

- Maps native types to RPC portable types.
- Marshals nativeRPC calls into a portable XDR/NDR format "for the wire."
- Handles network communications and timing issues.
- Operates reasonably efficiently.

In Mozilla's XML-RPC, JavaScript is an interpreted language, and the platform is already compiled in most cases. The implementation details and interface are therefore different from traditional RPC. The `nsIXmlRpcClient` is responsible for marshaling JavaScript RPC calls into portable XML, but it

delegates the sending and receiving of calls to a Mozilla channel object. This means that timeouts need to be checked for on the channel. The supplied interface delegates to the application programmer the mapping of JavaScript to XML-RPC types. It provides factory methods for creating XML-RPC types, but the application programmer needs to populate and assemble them for use by the RPC request. Finally, the `nsIXmlRpcClient` is implemented in JavaScript and regularly resolves names into objects using the `window.Components` object, so it is not highly optimized for performance.

SOAP is the intended and popular replacement for XML-RPC. SOAP technology is information-dense and requires a book of its own. The SOAP and XML-P standards at the W3C are highly recommended reading. Only the utter basics are presented here. XML-P (P stands for *Protocol*) is the future name for SOAP, assuming the popularity of SOAP as an acronym can be overcome.

A SOAP call is a request message followed by a response message, and so HTTP is a natural transport for SOAP. Both messages are in XML format. Both messages consist of an envelope tag that holds one optional header tag and one mandatory body tag. These tags are defined by the SOAP standard. The body tag contains a document fragment consisting of other tags. Those other tags are defined by the application programmer, who should have gone to the trouble of creating or using a formal XML schema definition for them. Those tags are the data sent and received.

For a programmer to create a SOAP message, objects are needed for the following tasks:

- To manipulate XML schema definitions
- To construct SOAP envelopes and their internal structure
- To set up a connection to a SOAP-enabled server
- To make the SOAP call
- To deal with any exceptions, faults, and failures
- To extract any returned XML document

Mozilla's solution to each of these bullet points is an object based on the matching XPCOM pair in this list:

- @mozilla.org/xmlextras/schemas/schemaloader;1 nsISchemaLoader
- @mozilla.org/xmlextras/soap/call;1 nsISOAPMessage
- @mozilla.org/xmlextras/soap/transport;1?protocol=http; nsISOAPTransport
- @mozilla.org/xmlextras/soap/call;1 nsISOAPCall
- @mozilla.org/xmlextras/soap/fault;1 nsISOAPFault
- @mozilla.org/xmlextras/soap/response;1 nsISOAPMessage

Many minor and ancillary interfaces assist this core set of features. On top of all these things is the need to set up an HTTP and SOAP-enabled server so that something can respond to the outgoing SOAP request.

The interface `nsISOAPMessage` is also exposed as an AOM object named `SOAPCall` and can therefore be created very simply:

```
var soap_call = new SOAPCall();
```

The `nsISOAPParameter` interface is similarly reflected in the `SOAPParameter` AOM object. These two objects allow simple SOAP calls to be made without extensive preparatory use of the `window.Components` array.

The famously available Google SOAP service is a genuine SOAP service that can be used to test Mozilla clients that make SOAP calls. It is described at *www.google.com/apis/index*. It is hard, however, to learn much when you don't control both the client and the server. A better solution is available in the Mozilla source code. This Web-hosted portion of the source contains sample code useful for learning and testing SOAP:

```
http://lxr.mozilla.org/seamonkey/source/extensions/xmlextras/tests/
```

If you have a Web server available with CGI support, then the three small `.cgi` programs in this directory (written in Perl) can be used to receive SOAP requests and respond in kind. The `echo.cgi` version implements a "ping" operation, which by convention should be the first service implemented when a group of related SOAP calls are defined. The other two `.cgi`s provide a success response and a failure response. The success response also contains response content.

There is enough XPCOM SOAP support in Mozilla for the platform to act as a SOAP server instead of a client, provided that the clients all use the same transport (ultimately one socket or file descriptor). This means that the platform cannot yet accept SOAP requests sent from anywhere in the world.

The final Web protocol stack protocol that Mozilla supports is the WSDL protocol, or Web Services Description Language. It bundles together a set of individual SOAP calls into a single definition document. Very roughly speaking, it is the XML equivalent of a CORBA IDL file or a Mozilla XPIDL file. WSDL definitions are the responsibility of the application programmer.

WSDL in Mozilla is brand new as this is written. The best place to look for up-to-the-minute information is this URL, which contains the XPIDL definitions for WSDL interfaces:

```
http://lxr.mozilla.org/seamonkey/source/extensions/xmlextras/wsdl/
```

To see the Contract IDs for the components that implement these interfaces, either read the `.h` header files in this directory, or use the Component Viewer tool list component Contract IDs with this prefix:

```
@mozilla.org/xmlextras/wsdl/
```

Such a listing will only work on versions 1.4 and later, which is the minimum version for full WSDL support.

16.4.6 XSLT Batch Processing

The Mozilla XSLT processing system can be exploited by scripts using this XPCOM pair:

```
@mozilla.org/document-transformer;1?type=text/xsl nsIXSLTProcessor
```

An object with this interface accepts two DOM trees or subtrees as arguments: One is a tree of XSLT tags that is a set of processing instructions; the other is the content to be transformed. A third tree or subtree, which contains the processed output, is returned. XSLT parameters can also be supplied as arguments. This system cannot work in-place—the results must be attached to an existing DOM hierarchy if that is required.

16.5 PLATFORM CONFIGURATION

Some scripting tasks inspect, manage, and update the state of the Mozilla Platform itself. To do that, internal aspects of the platform must be revealed via XPCOM interfaces. This topic covers the cache, file system directory, preferences, security, and user profiles.

16.5.1 Cache Control

The Mozilla Browser cache is intended to be transparent to all operations, but it is possible to interact with it if necessary. The cache is at work for all URL requests performed by the platform, unless it is explicitly avoided or turned off. Low-level access to the cache can be had via this XPCOM pair:

```
@mozilla.org/network/cache-service;1 nsICacheService
```

An object built this way also needs access to the constants provided by the `nsICache` interface. The details are surprisingly complex because the cache supports simultaneous access sessions with a single-write, multiple-read locking model. This means that low-level access to the cache can fail as a result of resource contention. It is easier to stay away from the detail and let higher level services like transports and channels manage the interaction for you. One handy use of this interface is the `evictEntries()` method, which can be used to empty the cache.

A very simple use of the cache is prefetching. Prefetching brings an `http:` URL from its original location into the cache without necessarily consuming or displaying it. Prefetching only works for `http:` URLs that are not HTTP GET requests (a request must not have a `?param=` part). Prefetching is accomplished with this pair of XPCOM objects:

```
@mozilla.org/prefetch-service;1 nsIPrefetchService
```

Similar, but finer control is also available on the `nsIRequest` interface, which is the basis for channels and transports. The `loadFlags` property can be used on a per-URI basis to control how a retrieved URI and the cache interact.

16.5.2 File System Directory

The Mozilla Platform has a directory service that allows scripts to locate well-known files and folders.

A Mozilla directory is like a phone book: It is used to look up the detail associated with a given name. Directories are therefore entirely separate from the concept of an operating system's file system and from the concept of a file system directory. Mozilla directories are usually called directory services to emphasize the way in which they serve up details in response to requests. There are many directories in Mozilla.

Of the directories that Mozilla implements, some provide access to remote resources, others provide access to data files on the local file system like the local Mozilla Address book, and still others provide access to internals of the running platform. One of these internally maintained directories holds a set of well-known operating system file and folder names, all of which are used by the platform. That is the only directory service discussed here.

Access to these file and folder names is required by scripts if applications are to use the same file system locations as the platform. The benefits of reusing these locations are that the application is then (a) properly platform-integrated and (b) somewhat protected from portability problems.

This internal directory, called the file system directory service, is implemented by this XPCOM pair:

```
@mozilla.org/file/directory_service;1 nsIDirectoryService
```

Note that `directory_service` contains an underscore, not a dash. This directory holds the locations of all the files and folders about which application programmers and applications need to know. The files and folder locations available in this directory are therefore fundamental to Mozilla applications. After file or folder locations are retrieved from a directory, they can be operated on just like any file or folder.

The `nsIDirectoryService` interface is not that useful by itself. All it can do is manage a set of provider objects. A provider is an object that supplies a subset of the directory contents to the directory service. In the normal case, a directory service object provides none of its own contents. Instead, each directory service has zero or more providers registered. Each provider contributes to the directory and supports the `nsIDirectoryServiceProvider` interface. When a script consults the directory service, that service looks through its providers to see if any of them have the details for the name the script

asked about. Providers are entirely hidden from the script when they are arranged in this way.

Apart from providers, the directory service system also uses other interfaces. The `nsIProperties` interface is the standard interface used to retrieve details of a name recorded in the directory. A sought-after name, in the form of a short string (effectively an alias or a nickname), is passed into the directory service via the `nsIProperties` method `get()`. Any item in the directory that matches that alias has its details returned. The file system directory service implements this `nsIProperties` interface. Listing 16.7 shows code that uses this standard interface:

Listing 16.7 Retrieving a file system resource from a directory with an alias.

```
var Cc = Components.classes;
var Ci = Components.interfaces;

var dir = Cc["@mozilla.org/file/directory_service;1"];
dir = dir.getService(Ci.nsIDirectoryService);  // Initialized

// Put calls to dir.registerProvider(provider_object) here

var dir_props = dir.QueryInterface(Ci.nsIProperties);

var file = dir_props.get("myalias", Ci.nsIFile);

if (file == null )
  alert("No Such Location");
```

This code creates the directory service object, adds no providers at all, grabs the `nsIProperties` interface, and retrieves the detail for the `"myalias"` alias. Because the XPCOM file system directory service stores file and folder information, the information returned (an object) is expected to have the `nsIFile` interface.

In this example, the last line of code might produce an alert for two reasons. The string "`myalias`" is not one of the well-known aliases, and so is unknown to the directory. That is trivial to fix—use a known alias. More seriously, there are no providers for this directory service; therefore, we expect that no aliases would be recognized at all. That is a reasonable reading of the code, but in practice it is not true. In practice, this directory has at least two providers at all times.

- The first of these providers is added by the directory object itself when it is created. This provider adds application-level aliases to the directory. These aliases match the install area files and folders for the Mozilla Platform.
- The second of these providers is added when the platform starts up. This provider is associated with the current user profile. It adds profile-specific aliases to the directory. These aliases match files and folders that are part of the current user profile.

- If the platform is displaying a Web page, and that Web page contains a plugin or a Java applet, then a third provider is added, but only while that page exists. This provider is associated with the plugin manager. It adds plugin-specific aliases to the directory. These aliases are processed differently than the other aliases (as described shortly), but a file object matching the alias is still returned, if it exists.

A rather confusing aspect of the directory implementation is this: The object that implements the directory service also implements a provider. This provider is specified by the XPCOM pair:

```
@mozilla.org/file/directory_service;1 nsIDirectoryServiceProvider
```

This provider is in addition to the three providers just noted. It is never (or rarely) registered with any directory service. Instead, it can be scripted directly. It is not hidden as the other providers are. This last provider adds aliases relevant to the XPCOM system that is at the heart of the platform. These aliases are for the lowest level files and folders required by the platform and include a number of operating-system-specific locations.

This last provider can be scripted as shown in Listing 16.8.

Listing 16.8 Retrieving a file system resource from a provider using an alias.

```
var Cc = Components.classes;
var Ci = Components.interfaces;

var prov = Cc["@mozilla.org/file/directory_service;1"];
prov = prov.getService(Ci.nsIDirectoryServiceProvider);

var result = {};  // an empty object
var file = prov.getFile("alias", result);

if ( file == null ) alert("No such location");

// alert(result.value)
```

Because providers are usually managed by a directory, the `getFile()` method has arguments that suit a directory object. The second argument to `getFile()`, an empty object, allows the provider to return some status information back to the directory. An ordinary script can throw this information away—it is only needed if the script is implementing its own directory service. See the XPIDL file for `nsIDirectoryServiceProvider` for details.

The remainder of this topic lists the aliases supplied by all these producers, starting with this last, special provider.

16.5.2.1 XPCOM File System Aliases

These aliases are provided by the special built-in directory service provider that is accessed directly. Tables 16.13 to 16.16 list the alias options available. Table 16.13 applies to all platforms.

Table 16.13 All-platform XPCOM file system aliases

Alias	Description of matching nsIFile
ComRegF	XPCOM component registry file—unused
ComsD	Folder that holds XPCOM components
CurProcD	Folder of the executable for the currently running process; always $MOZILLA_FIVE_HOME on UNIX
CurWorkD	Folder that is the present working directory of the current executable
DrvD	Returns the top of the operating system's file system—Windows: usually C:; UNIX: /; MacOS: the root volume
GreComsD	Folder holding GRE (Gecko Runtime Engine) XPCOM components
GreD	Folder GRE is installed in
Home	Home folder for the current user—Windows: %HOME%; UNIX: $HOME; MacOS: the documents folder
TmpD	Operating system location for temporary files—Windows: %TMP%; UNIX: $TMP; MacOS: the temporary files folder

Table 16.14 lists further aliases and applies only to Microsoft Windows. The stated CSIDL constants are part of the Microsoft Windows User Interface APIs and are used by Windows functions like `SHGetFolderPath()`. Each alias stands for a well-known folder. For a full description of these constants, see *http://msdn.microsoft.com/library/en-us/shellcc/platform/shell/reference/enums/csidl.asp*.

Table 16.14 Microsoft Windows only XPCOM file system aliases

Alias	CSIDL equivalent	Alias	CSIDL equivalent
AppData	CSIDL_APPDATA	netH	CSIDL_NETHOOD
Buckt	CSIDL_BITBUCKET	NetW	CSIDL_NETWORK
CmDeskP	CSIDL_COMMON_DESKTOPDIRECTORY	Pers	CSIDL_PERSONAL
CmPrgs	CSIDL_COMMON_PROGRAMS	PrntHd	CSIDL_PRINTHOOD
CmStrt	CSIDL_COMMON_STARTUP	Prnts	CSIDL_PRINTERS
Cntls	CSIDL_CONTROLS	Progs	CSIDL_PROGRAMS
DeskP	CSIDL_DESKTOPDIRECTORY	Rcnt	CSIDL_RECENT
DeskV	CSIDL_DESKTOP	SndTo	CSIDL_SENDTO
Drivs	CSIDL_DRIVES	Tmpls	CSIDL_TEMPLATES
Favs	CSIDL_FAVORATES	WinD	CSIDL_WINDOWS

Be aware that the aliases prefixed with `Cm` will fail (and throw an exception) on single-user versions of Microsoft Windows, like Microsoft Windows 98.

Table 16.15 lists further aliases; it applies to the Macintosh only.

Finally, Table 16.16 lists the remaining miscellany of aliases. Aliases for OS/2, BeOS, OpenVMS, and others are not shown.

Together, these aliases define all the file system locations known to the XPCOM core of the platform. It is easy to see that application code can become nonportable if these aliases are used more than trivially.

16.5.2.2 Application File System Aliases These aliases originate from the provider that is attached to the directory service when it is created. It is always available. These aliases are standard across all platforms.

The XPCOM system is not the whole of the Mozilla Platform. On top of that core is a large collection of components and infrastructure that makes up the rest of the platform. That platform includes installation areas for browsers and other products, chrome, caches, registries, and so on. Those locations and the location of the user profile system are described by these aliases. They are shown in Table 16.17.

Table 16.15 Macintosh only XPCOM file system aliases

Alias	Folder	Alias	Folder
ApplMenu	The Apple Menu	Exts	The Extensions folder
ClassicPrfs	Mac Classic Profile folder	Isrch	The Internet Search folder
CntlPnl	The Control Panel	Prfs	The Preferences folder
DfltDwnld	The Default Download folder	Shdwn	The Shutdown folder
Docs	The Documents folder	Trsh	The folder holding the Trash
Desk	The folder holding the Desktop		

Table 16.16 Miscellaneous XPCOM file system aliases

Alias	Description of matching nsIFile
Fnts	Macintosh and Microsoft Windows: the folder holding system fonts
LibD	UNIX: /usr/local/lib/netscape
Locl	UNIX: /usr/local/netscape
Strt	Macintosh and Microsoft Windows: the startup folder
SysD	Macintosh OSX only: the system folder
UlibDir	Macintosh OSX only: the /usr/lib folder

Table 16.17 Application install file system aliases

Alias	Description of retrieved object	Path name relative to install area
AppRegF	The global application registry file	Located elsewhere (see Chapter 17, Deployment)
AppRegD	The folder holding the global application registry	Located elsewhere (see Chapter 17, Deployment)
DefRt	The top folder of the defaults area	Defaults
PrfDef	The folder holding the default preferences	Defaults/pref
profDef	The folder holding default profile values for the current locale	Defaults/profile/{locale}
ProfDefNoLoc	The folder holding default profile values for the default locale	Defaults/profile
DefProtRt	The top folder for all user profiles	Located elsewhere (see below)
Ares	The resources folder	Res
Achrom	The chrome folder	Chrome
SrchPlugns	The folder holding plugin search and download configuration files	Searchplugins
ApluginsDL	An nsIEnumerator list of available plugin files	Plugins/*
XPIClnupD	The folder holding uninstall programs	Uninstall
UserPlugins	The folder under the current user profile that holds profile-specific plugins	Located elsewhere
OSXUserPlugins	MacOS X only; the folder holding user plugins	Located elsewhere
OSXLocalPlugins	MacOS X only; the folder holding local plugins	Located elsewhere
MacSysPlugins	Mac Classic only; the folder holding system plugins	Located elsewhere

The `DefProtRt` alias returns the following per-platform values:

- UNIX: ~/.mozilla
- Windows: {CLSID_APPDATA}\Mozilla\Profiles
- Macintosh: :Documents:Mozilla:Profiles

16.5.2.3 Profile File System Aliases These aliases originate from a provider added to the directory service when the platform starts up. In a full distribu-

tion of the platform (i.e., one that is not embedded or otherwise cut down), it is always available. These aliases, in Table 16.18, are standard across all platforms.

16.5.2.4 Plugin File System Aliases Plugin file system aliases originate from a provider that is added to the directory service when plugins or Java is required. These aliases are used to retrieve a file implementing a plugin, but they work only on Microsoft Windows.

The other directory service providers merely translate an alias to an operating-system-specific file or folder. This provider does that translation too, but first it analyzes the available resources in more depth. It uses the alias to extract from the platform preference information, which states the minimum version of the plugin that is needed and whether that plugin is enabled. It then compares any minimum enabled version with product versions installed in the operating system. It returns a file for the operating-system-installed version if it is sufficiently new. The version comparison is done using the format described in Chapter 17, Deployment. The known aliases are listed in Table 16.19.

Table 16.18 Profile file system aliases

Alias	Description of retrieved object	Path name relative to user profile
PrefD	The folder holding the preference file; same as ProfD	.
PrefF	The file holding the user preferences	prefs.js
ProfD	The topmost folder of the current profile	.
Uchrm	The folder holding user chrome	chrome
LclSt	The file holding persistent data about the user's Mozilla windows	localstore.rdf
Uhist	Classic Browser URL history file	history.dat
Upanels	Classic Browser user-defined sidebar panels file	panels.rdf
UmimTyp	The platform-wide MIME type information	mimeTypes.rdf
Bmarks	Classic Browser bookmarks file	bookmarks.html
Dloads	Classic Browser download history file	downloads.rdf
SrchF	Classic Browser search engine configuration file	search.rdf
MailD	Folder holding local mail accounts	Mail
ImapMD	Folder holding IMAP mail accounts	ImapMail
NewsD	Folder holding Newserver configuration	News
MFCaD	File holding current visual settings of Classic Mail folders	panacea.dat

Table 16.19 Plugin file system aliases

Alias	Description of matching nsIFile
plugin.scan.SunJRE	The Mozilla OJI Java JRE plugin file
plugin.scan.Acrobat	The Adobe Acrobat plugin file
plugin.scan.Quicktime	The Apple Quicktime plugin file
plugin.scan.WindowsMediaPlayer	The Microsoft Windows Media Player executable
plugin.scan.4xPluginFolder	Netscape 4.x plugin folder

16.5.3 Preferences

The current profile's user preferences, the current global preferences, and a preference file stored anywhere on the local computer can all be manipulated from scripts. This XPCOM pair is responsible:

```
@mozilla.org/preferences-service;1 nsIPrefService
```

Preferences cannot be changed from scripts outside the chrome unless standard Web security restrictions are removed. A user can modify preferences directly in versions 1.3 and higher by right-clicking on the content displayed by the `about:config` URL.

16.5.4 Security

This topic explains how security is implemented. Because security is a big subject, we'll look at only those security constraints that directly bear on scripting.

In Netscape version 4.x browsers, security checks were handled by the Java subsystem of the browser. In Mozilla, that is no longer the case—the Mozilla Platform handles its own security needs with its own security implementation. No Java is required.

A piece of Mozilla code can be in one of four security states: Web Safe, Trusted, Certified, or Domain Policied. In practical terms, code means JavaScript scripts, but these security states also apply to all downloadable documents, including HTML.

New support for WSDL in the platform includes a further wrinkle on security. This wrinkle requires an additional security check when a remotely located Web service is first used. This check is designed to protect the server vending the Web service, not the platform calling that service. It requires that the platform ask the service for permission to use the service. As this goes to print, it is proposed that the `nsIWebScriptsAccessService` interface be the client-side entry point for this check.

Most, but not all, security issues are handled in the XPConnect code that connects JavaScript to the internals of the platform.

In all cases, if a security violation occurs, errors are reported to the JavaScript Console.

16.5.4.1 Web Safe Security Web Safe security is the default security applied to Mozilla applications. It is the security that applies to XUL-based applications installed outside the chrome, and it is the security applied to Web-based applications that run inside a window that displays HTML or XML (a browser). Web Safe security provides a nearly fully secure environment, by putting two obstacles in the way of scripts.

The first obstacle in Web Safe security is a set of restrictions designed to guarantee that the user interface is under end user control. An example restriction is the requirement that all windows be at least 100 pixels wide and high, so they are obvious to the user.

The second obstacle in Web Safe security is the Same Origin test, which is used pervasively throughout the platform. This policy says that a script can only use a resource that originates via the same protocol, and from the same domain name and IP port number, as the script itself. A script downloaded from *www.test.com* cannot affect a Web page downloaded from *www.pages.com* or from *ftp://www.test.com* or even from *www.test.com:99*, where *99* is a different port number.

The Same Origin test prevents scripts from affecting different-origin windows in a running application and from crossing frame boundaries into different-origin documents. Both the contents of the chrome and the entire set of XPCOM components are considered different-origin to all Web sites. Therefore, in the case of scripts with a remote origin, components are entirely unavailable under Web Safe security. They are also unavailable to local scripts stored outside the chrome.

The Same Origin test does not apply to the special URL `about:blank`, which is always accessible.

16.5.4.2 Trusted Security The opposite extreme to Web Safe security is Trusted security. Scripts in the Trusted security state have no security restrictions at all. They can access all XPCOM components automatically and all scripts and documents regardless of origin. Scripts and all other resources installed in the chrome are Trusted. In particular, resources in the chrome never need permissions from the user.

One outstanding issue with the use of chrome is that adding content into the chrome is hard to do securely. The XPInstall system, described in Chapter 17, Deployment, does not yet insist on authentication of chrome installation packages. Such authentication requires digital certificates and signatures. This means that there is no guaranteed check that a package intended for the chrome originates from the source that it claims to originate from. In theory, a malicious chrome package could lie about its origin, and when the user agreed to install it, it would be able to exploit its new, trusted state. In practice, no one has yet bothered to interfere with Mozilla in this way.

16.5.4.3 Certified Security In between the Web Safe and Trusted security states is the Certified state. Scripts and other resources can be decorated with digital certificates that contain public key encrypted digital signatures and that can be authenticated (confirmed accurate) by a respected organization. In this security state, all scripts are treated as Web Safe until their signing information is examined. If the signing proves acceptable, then the script can act as a Trusted script.

Digital certificates are a world of their own; only the consequences for application scripts are noted here. To sign scripts and other resources digitally, the SignTool tool is required. It is not provided by Mozilla, but it is available from Netscape at the *http://devedge.netscape.com* Web site, along with documentation. In addition to signing files digitally, this tool can produce a test certificate that can be used in the absence of a real certificate. Real certificates cost money to acquire.

The use of digital certificates requires two pieces of configuration. First, a database of certificates must be maintained by the platform. In Classic Mozilla, this is held automatically, although some of the more obscure panels in the preferences dialog box allow certificates to be user-managed. The second piece of configuration is that the user must give permission every time a signed script is encountered. That is extremely inconvenient, so browsers can also remember the permissions the user has granted in the past and reapply them silently on future occasions. All this configuration information is held in the user profile.

The difference between Trusted and Certified arrangements is that a Certified arrangement requires at least one interactive confirmation by the user. The only way to avoid this is to build a custom installation of a browser or the platform with the required certificates and permissions already bundled with it.

The Certified security state can be attained without any certificates. This user preference drops the need for certificates or digital signing, but it still requires that the programmer appeal for special privileges and that the user grant permission to use those privileges:

```
user_pref("signed.applets.codebase_principal_support", true);
```

This preference is only useful for developing applications where the ultimate security model will be the Certified model.

If Certified security is the model chosen, then each piece of code that wants to perform a Trusted operation (like using an XPCOM object) must appeal to the user for permission to use that operation. That permission is requested by preceding critical sections of code with a function call:

```
window.netscape.security.PrivilegeManager.enablePrivilege("P1 P2
     P3");
```

This call either asks the user for permission with a dialog box or silently succeeds if permission has been granted and remembered in the past. On success, the security of the following code is raised to the Trusted state for the

Table 16.20 Privilege strings used in code-signing security

Privilege	Affected targets*
UniversalBrowserRead	Reading of sensitive browser data; allows the script to pass the Same Origin check when reading from any document
UniversalBrowserWrite	Modification of sensitive browser data; allows the script to pass the Same Origin check when writing to any document
UniversalXPConnect	Unrestricted JavaScript access to XPCOM components using XPConnect
UniversalPreferencesRead	Read preferences using the navigator.preference() method
UniversalPreferenceWrite	Set preferences using the navigator.preference() method
CapabilityReferencesAccess	Reads or sets the preferences that define security policies, including which privileges have been granted and denied to scripts; also requires UniversalPreferencesRead and/or UniversalPreferencesWrite
UniversalFileRead	Display or submit files that have file: URLs

*The data in Table 16.20 appear courtesy Jesse Ruderman and mozilla.org.

specified privileges only. The Trusted state ends when the current JavaScript scope ends, which is usually at the end of a function or method call.

In this call, P1, P2, and P3 are a space-separated list of privilege keywords—at least one such keyword is required. Table 16.20 lists the available privileges and the targets to which they control access. A target is just any capability or functional feature of the platform.

Inside the platform, each of these privileges might be checked in a number of places so that the overall effect is that the target is fully protected by the security system.

16.5.4.4 Policied Security The final security model available to scripts is a set of user preferences. Policied security grants and denies Trusted access to all documents retrieved from specific origins. An origin is a protocol + domain + port combination, as used in the Same Origin test. This form of security controls use of specific JavaScript properties in those retrieved documents. Those specific properties are therefore the targets for this security model. The set of grants and denials is bundled up into one configuration item, which is called a policy. One policy can apply to several different origins. This security model is the least used of the security options Mozilla provides.

If this security system is not specified, retrieved documents follow the Web Safe security model. If this security system is used, retrieved documents may have more or less restrictions than the Web Safe model. Therefore, if the user's profile can be modified, Domain Policied security can be the most or least restrictive of all the security options.

The Policied security model has no direct user interface in the preferences system of the Mozilla applications. Some of the checkbox preferences in that system are implemented using this security system, but that is not obvious to the user.

This preference system exists for the following reasons:

- To support specific preferences that enable or disable useful functionality.
- To compete with Internet Explorer's zone-based preference system.
- To empower the user so that irritating Web sites can be individually disabled.
- To provide a powerful and flexible system in case it proves useful.

To use this security, new user preferences must be set. Three steps are required: define a policy name; define a set of origins that the policy applies to; and define access rules for the individual object properties that the security model controls. These steps are examined in turn.

There are three kinds of policy names—explicit, wildcard, and default. Every property that might have an access rule can be associated with one of each of these names, and these three names have a pecking order.

At the bottom of the pecking order are the default policies. There is one default policy per JavaScript property, and it is applied if no other policies exist. If no default policies are specified, then the single default policy named `"default"` applies to all properties. This `"default"` default policy may also be modified. It will shortly be clear why more than one default policy is useful.

Next from the bottom is the wildcard policy. It has the name `"*"` (asterisk). It is applied when it is explicitly stated, and in that case it overrides default policies.

At the top are the explicit policies. These policies are named when explicitly stated and are applied first and foremost. These policies override the other two kinds.

To name policies, two preferences are used:

```
user_pref("capability.policy.policynames","p1 test foo");
user_pref("capability.policy.default_policynames","normal,off");
```

The policy names are space- or comma-separated and may not contain a period character. The first line of the preceding code specifies three policies; the second specifies two default policies. The wildcard policy name is automatically recognized and doesn't need to be specified.

Having made up policy names, each policy is then provided with a list of origins. Each policy will be applied only for documents retrieved from those origin names. The names of the policies created are used in the preference string that specifies the sites. For example, the policy called `mypol` has its origins specified thus:

```
user_pref("capability.policy.mypol.sites", "http://test.com http://
     x.org");
```

The argument is a space- or comma-separated list of partial URLs, and the word "sites" is synonymous with "origins." The partial URLs cannot include specific subparts of the origin's Web site. There should be zero or one of these preferences per site. If the policy name is a default policy name, then the sites listed will have that default policy. This allows different defaults for different sites. If the wildcard policy is required, specify * instead of mypol.

After the policy names and origins are specified, all that remains is to create the access rules. There are three types of rules. A single preference line is required for each rule that is stated.

The first and most general rule syntax applies to all JavaScript properties, regardless of whether they are simple values or methods. For the policy mypol, it has the syntax

```
user_pref("capabilities.policy.mypol.Iface.Prop","Keywords")
```

Iface, Prop, and Keywords must be replaced with specific strings.

- Iface is the name of the JavaScript object holding the property of interest. In fact, it must be the shortened XPCOM interface name that has had the nsIDOM prefix removed. Example names are ChromeWindow, HTMLDocument, and XULImageElement. Some DOM objects have shorthand object names like Image, but the official HTMLImageElement name must be used.
- Prop is the property name to which the access rule applies. It is usually an attribute or method of an XPCOM interface, like the value property of many form controls.
- Keywords must be a space- or comma-separated list of privilege names from Table 16.20 *or* one of the sole keywords AllAccess, NoAccess, or sameOrigin. AllAccess is the same as specifying all the keywords from Table 16.20. sameOrigin means that the Web Safe security rules should apply. NoAccess means that the property cannot be read or written at all.

An example rule is

```
user_pref("capabilities.policy.*.History.back","NoAccess");
```

This rule says that the wildcard policy disabled the back() method of the nsIDOMHistory object. That object is used in the Mozilla Browser only, so this rule serves to prevent the user from navigating backward when surfing the Web.

The second rule syntax, which applies only to nonmethod JavaScript properties, gives control over the ECMAScript [[Get]] and [[Set]] operations on that property. The syntax is

```
user_pref("capabilities.policy.mypol.Iface.Prop.Access","Keyword");
```

`Iface` and `Prop` are the same as the earlier syntax. `Keyword` is restricted to one of the values `NoAccess`, `AllAccess`, and `sameOrigin`. `Access` is one of the strings `set` or `get`. This syntax therefore specifies up to two rules, one for getting and one for setting a property in question. An example that makes the title bar of a XUL window read-only is

```
user_pref("capabilities.policy.default.ChromeWindow.title.set","NoAcc
     ess");
```

The final rule syntax applies to the special case of JavaScript. This single preference can be used to enable or disable JavaScript entirely on a per-origin basis:

```
user_pref("capabilities.policy.mypol.javascript.enabled","Keyword");
```

In this case, only the policy name and `Keyword` vary. `Keyword` can be set to one of `NoAccess` (meaning disable JavaScript) or `AllAccess` (meaning enable JavaScript). If JavaScript is globally disabled, this rule is useless.

A policy typically includes a number of these rules, which together open up or close down access of a particular kind. In the case where access is to be denied, every property that represents an access loophole must be plugged if the policy is to be robust. To get a list of properties to consider, look at an example of the JavaScript object target using the DOM Inspector. Set the right pane of the Inspector to show the JavaScript Object option, and tick off any property that might be exploited.

XBL bindings are good examples of loopholes. It is common for a binding to include many convenience methods and properties. These methods and properties often have overlapping functionality. If the end purpose is to prevent a particular property on the bound object from being changed, a rule on that property is not enough. All bindings methods and properties that touch that property must also be disabled with a `NoAccess` security rule.

Using Policied Security to deny access is only a mild form of security according to the U.S. Department of Defense Orange book on security. This is because it is a discretionary system; enforcing security is up to the policy creator. Worse, it is a system that requires the policy creator to be aware of all existing security loopholes in advance.

16.5.4.5 Special Restrictions Beyond the structure imposed by Mozilla security restrictions are a few feature restrictions that are worth noting. These restrictions apply to version 1.4 at least.

- ☞ It is not possible to load a string bundle from a remote location because the platform code that fetches string bundles does not support HTTP.
- ☞ The JavaScript and Java security models could interoperate better. If a Java applet is signed, JavaScript cannot use the privileged object methods in that signed applet.

That concludes the discussion of Mozilla security.

16.5.5 User Profiles

The Mozilla directory service provides access to files and folders in the current user profile. It is also possible to access and manage the set of existing profiles. This XPCOM pair is responsible:

```
@mozilla.org/profile/manager;1 nsIProfile
```

File system locations inside a specified profile cannot be accessed with the `nsIProfile` interface. That access requires this XPCOM pair:

```
@mozilla.org/profile/manager;1 nsIProfileInternal
```

16.6 HANDS ON: SAVING AND LOADING NOTETAKER NOTES

This "Hands On" session is about reading and writing data to disk using XPCOM objects. It shows how to work with Mozilla's RDF support directly from JavaScript, in a way that complements existing XUL templates. It also shows how to get access to the current user profile, a rather trivial task.

We'll also complete the NoteTaker tool. That means saving, deleting, and loading notes. To do that, we must have suitable RDF data sources in place. We'll do our scripting with the basic XPCOM RDF interfaces rather than the RDFLib JavaScript library. This choice is merely designed to improve our understanding of the basic RDF services. The current note will always be stored in JavaScript; the fact store associated with the RDF data source will represent the pool of all currently configured notes.

We'll also do a bit of RDF query repair. Template-based queries have their limitations, and those limits don't apply to scripts. Up until now, we have had no way of finding a note for the current URL, unless the note's URL was an exact match. We'll fix that. Also, the summary textbox on the toolbar is really too simple for a template, so we'll reimplement that query as a script. In fact, we'll script all the templates a little.

Like all good programming jobs, we begin with a little design.

16.6.1 Data Source Design

In Chapter 14, Templates, we made our content dynamic using the XUL attribute of the data sources. Each template specified its own RDF data source. Although this is a brief and convenient way to proceed, it assumes that the URL of the `notetaker.rdf` file is known at application development time. Now that the `notetaker.rdf` file will be stored in the user's profile, no such fixed URL exists.

To manage this change, we move the integration of RDF from XUL to JavaScript. Instead of specifying the RDF file as an attribute in XUL, we specify it as an XPCOM object in JavaScript. From JavaScript, we can use other XPCOM objects to find the RDF file's location dynamically.

In the XUL template code, we'll still need a data source, or else the content isn't a template. We use Mozilla's empty placeholder data source, named `rdf:null`. After the XUL is loaded, we'll create a new data source from the XPCOM URL object and attach it to each template using JavaScript. In this way, one data source object will be responsible for all the RDF traffic to and from the data source.

This use of a single, coordinated data source is not the only way for a set of XUL templates to share RDF. If two templates have the same data source attribute, then all RDF facts still come from a shared, single set of facts (a single fact store). This must be the case, or else the code written in Chapter 14, Templates, would not have worked. All that we are doing here is detaching the data source from XUL so that we can supply a customized one. We could instead create a custom data source for each template. As long as they were all based on the same URL, they would all still operate on one shared set of facts.

After we have this data source, we can read and write it using our own JavaScript functions and the numerous RDF interfaces available. At the same time, the template system will access the same data source using the built-in XUL template builder.

To make the design neater, we'll complement the `Note` JavaScript object with a `NoteDataSource` object. The `Note` object was last visited in "Hands On" in Chapter 14, Templates. Each time we need to work on the data source, we'll have the option of capturing that work as a method of our new object.

16.6.2 Data Source Setup

To start with, we need to put a copy of the `notetaker.rdf` file (a test version) in the current user profile if testing is to do anything useful.

Our main setup task is to get access to that RDF file. That means starting with a file name and an idea of its location and ending up with an `nsIRDFDataSource` object. We'll hard-code the file name, but not its location. We'll use several of the facilities described in this chapter to ready the data source.

To locate a file portably, we must use a directory service. Inspecting the directory service tables of known aliases in this chapter, we conclude that the `ProfD` alias from Table 16.18 is the most portable way to reach the current user profile's folder. We turn that alias into an `nsIFile` that holds the profile folder, extend the path of that folder to specify our `notetaker.rdf` file, convert the resulting file into a URL, and then finally use that URL to create a data source. Whew. Listing 16.9 shows this code:

Listing 16.9 Finding and initializing a locally stored data source.

```
var Cc = Components.classes;
var Ci = Components.interfaces;

// Note session object
```

```
function NoteSession() {
  this.init();
}

NoteSession.prototype = {
  config_file : "notetaker.rdf",
  datasource : null,
  init : function (otherfile) {
    var fdir, conv, rdf, file, url;

    if (otherfile) this.config_file = otherfile;

    with (window) {
      fdir = Cc["@mozilla.org/file/directory_service;1"];
      fdir = fdir.getService(Ci.nsIProperties);

      conv = Cc["@mozilla.org/network/protocol;1?name=file"];
      conv = conv.createInstance(Ci.nsIFileProtocolHandler);

      rdf  = Cc["@mozilla.org/rdf/rdf-service;1"];
      rdf  = rdf.getService(Ci.nsIRDFService);
    }

    file = fdir.get("ProfD", Ci.nsIFile);
    file.append(this.config_file);

    if (!file.exists())
      throw this.config_file + " is missing";

    if (!file.isFile() || !file.isWritable() || !file.isReadable())
      throw this.config_file + " has type or permission problems";

    url  = conv.newFileURI(file);
    this.datasource = rdf.GetDataSource(url.spec);
  }
};

var noteSession = new NoteSession();
```

The `init()` method of this `NoteSession` object does all the work. In there, we set up three handy XPCOM objects. We extract the current user profile folder as `nsIFile`. The `append()` method makes an in-place modification to that folder so that it fully specifies our configuration file. The `append()` method does not return anything. Next we perform a couple of sanity checks to make sure that the configuration file is in place—in our completed tool we'll supply a skeleton copy at deployment time, so the file should always exist. In real life, some extra logic should be included here to re-create the file in case it has been deleted. Last, we convert from `nsIFile` to `nsIURL` using `newFileURI()`, then from `nsIURL` to `String` with `url.spec`, and finally from `String` to `nsIRDFDataSource` with `getDataSource()`.

This series of steps is a standard approach for readying a data source. If the data source is internal or remote, some steps might vary a bit. For example, if the URL of the data source is known in advance, little more than `GetDataSource()` is required.

16.6.3 Dynamically Allocating Data Sources to Templates

Now that we have a data source available, let's use it. We want to modify the existing templates so that their displayed data come from the data source's URL, not from a hard-coded XUL attribute. To do that, we'll use a placeholder attribute `datasources="rdf:null"` until the real data source is scripted in.

16.6.3.1 Toolbar Changes We will throw away altogether the template used on the NoteTaker toolbar `<textbox>`. We're making this change because it's unnecessarily complex—the textbox need only act as a simple form element. We only included this template in previous chapters to illustrate the simplest of template uses. Templates aren't a final solution for every problem. The plain `<textbox>` returns to

```
<textbox id="notetaker-toolbar.summary"/>
```

This textbox is filled by `refresh_toolbar()`. That function will now do a simple copy from our note object, instead of a template rebuild. That's all for the summary textbox on the toolbar.

The Keyword dropdown menu on the toolbar has a very standard template query. There would be no reason to change it if the profile-specific data source could be hard-coded. Because it can't be hard-coded, we must change code in both XUL and JavaScript.

This menu has been data-driven since Chapter 14, Templates, but it has not been as dynamic as it may seem. In Chapter 14, it was generated at XUL page creation time and remained static thereafter. From now on, it must change anytime a keyword is added. Any XUL content added or removed may cause the document, including the menu, to reflow. Reflow is an automatic process, but it works most reliably on simple tags. For complex tags like `<menulist>`, careful use of XUL is required. Used carelessly, the menu will appear broken.

To see this broken effect, recall that the original `<menulist>` and template has the form of Listing 16.10. This listing has `"rdf:null"` as the placeholder data source.

Listing 16.10 NoteTaker `<menupopup>` before dynamic support.

```
<menulist id="notetaker-toolbar.keywords" editable="true">
  <menupopup datasources="rdf:null" ref="urn:notetaker:keywords">
    <template>
      <menuitem uri="rdf:*"
        label="rdf:http://www.mozilla.org/notetaker-rdf#label"/>
```

```
    </template>
  </menupopup>
</menulist>
```

Note that only the <menuitem> tags are part of the template. With "rdf:null" in place, the complete menu, consisting of static XUL and generated template content, will appear instead as

```
<menulist id="notetaker-toolbar.keywords" editable="true">
  <menupopup datsources="rdf:null" ref="urn:notetaker:keywords">
  </menupopup>
</menulist>
```

Figure 16.2 shows the results of this changed code.

Fig. 16.2 Templated <menulist> with zero items.

This user interface has layout problems and interaction problems; the sources of these problems can be found in Listing 16.10. We might decide to overlook these problems. After all, our code will modify the template from an onload handler so that rdf:null is replaced right away with our handcrafted data source. This should generate <menuitem> tags for the menu, and all should be well.

Unfortunately, all is not well. The popup (dropdown) content is sized by the <menupopup> tag, which has a frame. That frame does not dynamically relayout after it is created, or at least not yet. This means the XUL code in Listing 16.10 will not work when its template is modified after display. Listing 16.11 shows a better version of that Listing 16.10:

Listing 16.11 NoteTaker <menupopup> after dynamic support.

```
<menulist id="notetaker-toolbar.keywords"
  editable="true"
  datasources="rdf:null"
  ref="urn:notetaker:keywords"
>
  <template>
    <menupopup>
      <menuitem uri="rdf:*"
        label="rdf:http://www.mozilla.org/notetaker-rdf#label"/>
    </menupopup>
  </template>
</menulist>
```

In this version, the <template> tag and associated attributes have been *moved up one* in the tag hierarchy. Now, the <menupopup> tag pair is regener-

ated every time the template runs. Only one <menupopup> tag pair will be generated because those tags are outside the spot where the uri attribute is declared. Recall that the uri attribute is the beginning point for per-query solution generation of template content. Because the <menupopup> is generated each time <menuitem>s are generated, there is opportunity for the <menupopup>'s frame, also created each time, to get its layout correct. This is the recommended approach for template-driven dropdown menus whose content must change after the initial display.

Even with this fix, the keywords menu may have one further usability problem, although this problem doesn't appear in our particular application. Figure 16.3 shows a test toolbar before and after a menu dropmarker has been clicked once. The top window is the before case.

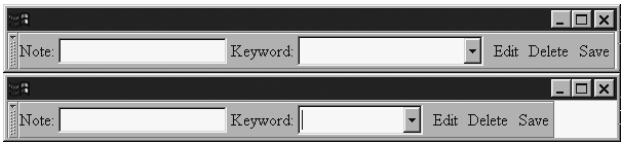


Fig. 16.3 Reflow problems with templated menus.

In this test, the <textbox> at the top of the menu has an initial width that is the default for a <textbox> tag. When the menu is clicked, the menu items are exposed, and the textbox is layed out again to match the width of the widest menu item. The net result is that the dropmarker for the menu jumps to one side. This is confusing for the user. A workaround is to set the width attribute on the <menulist> tag. Fortunately, this problem doesn't occur for NoteTaker, provided a real Web page is displayed.

JavaScript changes required for this newly dynamic menu are quite simple. The functions refresh_toolbar() and init_toolbar() must be changed to attach the new data source to the menu template. Listing 16.12 shows these two functions with data source changes.

Listing 16.12 NoteTaker toolbar changes for data-source-based templates.

```
// onload browser listeners work in the capture phase
window.addEventListener("load", init_handler, true);

// load RDF content for the toolbar. Relies on note object.
function init_toolbar(origin)
{
  if ( origin != "timed" ) {
    // avoid running inside any onload handler
    setTimeout("init_toolbar('timed')",1);
  }
  else
  {
```

```
    var menu = window.document.getElementById('notetaker-
          toolbar.keywords');
    menu.database.AddDataSource(noteSession.datasource);
    menu.ref = 'urn:notetaker:keywords';
    setInterval("content_poll()", 1000);
  }
}

// update the toolbar based on the latest content.
function refresh_toolbar()
{
  var box = document.getElementById('notetaker-toolbar.summary');
  box.value = note.summary;

  var menu = document.getElementById('notetaker-toolbar.keywords');
  menu.ref = 'urn:notetaker:keywords';
}
```

In "Hands On" in Chapter 14, Templates, these functions nervously called `rebuild()` every time a template changed in the least. Now, however, it's clear that the template data are based on the `xml-datasource` data source, which supports fully coordinated template updates. A call to `rebuild()` is therefore not required. If in doubt though, always call `rebuild()`.

The `init_toolbar()` function in this listing attaches the new data source to the dropdown menu template, updates the template's `ref` property, and starts `content_poll()`, which watches the content part of the browser for URL changes. Even though the `ref` property doesn't change value, this assignment tells the template to recalculate solutions for the query held.

The `setTimeout()` call is, as before, a workaround for outstanding defects in the onload event handler. Compare `refresh_toolbar()` with the `Refresh()` method of `nsIRDFRemoteDataSource`. The latter method refreshes the fact store on which a given data source is based. The `refresh_toolbar()` function refreshes only XUL content, including XUL content that results from a template query.

That concludes the display-oriented changes to the NoteTaker toolbar. We'll return to the toolbar when we script up support for user data entry.

16.6.3.2 Edit Dialog Changes The NoteTaker Edit dialog box is the other part of the NoteTaker tool that contains templates. Those templates also require script-initialized data sources. The Edit panel of the dialog box doesn't have any templates at all. The Keyword panel has a template on a `<listbox>` and another on a `<tree>`.

The procedure for attaching a data source to these two templates is very similar to the procedure used on the toolbar. We replace `datasources="notetaker.rdf"` with `datasources="rdf:null"` in two places in `editDialog.xul`. We create a function `init_dialog()` in

dialog_action.js, and we modify the existing refresh_dialog() function. Those updated functions are shown in Listing 16.13.

Listing 16.13 NoteTaker dialog changes for data-source-based templates.

```
window.addEventListener("load", init_dialog, "true");

function init_dialog()
{
  if ( origin != "timed" ) {
    // avoid running inside any onload handler
    setTimeout("init_dialog('timed')",1);
  }
  else
  {
    var listbox = document.getElementById('notetaker.keywords');
    listbox.database.AddDataSource(window.opener.noteSession.datasource);

    var tree = document.getElementById('notetaker.related');
    tree.database.AddDataSource(window.opener.noteSession.datasource);

    refresh_dialog();
  }
}

function refresh_dialog()
{
  var listbox = document.getElementById('dialog.keywords');
  listbox.ref = window.opener.note.url;
  //listbox.ref = "http://saturn/test1.html";  // test case

  var tree = document.getElementById('dialog.related');
  tree.ref = window.opener.note.url;
  //tree.ref = "http://saturn/test1.html";     // test case
}
```

The init_dialog() function is nearly identical to the init_toolbar() function, adding the same data source to each of the two keyword templates. The refresh_dialog() function is also similar and includes some example URLs from the notetaker.rdf test data that can be used to unit test the template updates. These changes make no difference to the user interface; they merely support the relocated notetaker.rdf file.

In the "Hands On" session in Chapter 13, Listboxes and Trees, we experimented with dynamic listboxes scripted up using the DOM interfaces and no templates. That code required about 30 lines of JavaScript. In the "Hands On" session in this chapter, we have achieved the same effect using a template and only a few lines of scripting.

With these changes in place, all the NoteTaker tool's templates are now driven from the notetaker.rdf file located in the user profile.

16.6.4 Scripted RDF Queries Using XPCOM Interfaces

XUL's template system is just one way to create a query on a set of RDF facts. It is a declarative approach similar to SQL. Another approach is to pick through the RDF facts by hand, using a script. This is equivalent to navigating a data structure, so it is an algorithmic or algebraic approach. This second approach means using the many XPCOM interfaces that are available for manipulating RDF content. Those interfaces provide some navigation assistance, so the scripting effort required is as large as it might seem.

The NoteTaker tool has one query that benefits from a scripted solution. That query is responsible for looking up any existing note for the currently displayed URL. Templates are not an automatic solution for this case for several reasons:

- The destination of the data is JavaScript, not XUL, and templates don't support the `<script>` tag as content.
- This query has no visual output.
- The "Chop Query" feature of the dialog box works two ways. Not only does it optionally remove the parameters from the URL for an HTTP GET request, but it also demands that such a URL be matched to a note with or without the parameter string present. That kind of matching means string operations on the URL.

Templates don't provide string operations, but scripts do, so we'll implement this lookup query with a script.

This lookup query is implemented by the `resolve()` method of the note object, in `notes.js`. This method was created as a stub in past chapters and now gains a full implementation. It loads the RDF details for a note into the properties of the note object. Listing 16.14 shows its implementation.

Listing 16.14 NoteTaker script-based RDF query.

```
resolve : function (url) {
  var ds = window.noteSession.datasource;
  var ns = "http://www.mozilla.org/notetaker-rdf#";

  var rdf = Cc["@mozilla.org/rdf/rdf-service;1"];
      rdf = rdf.getService(Ci.nsIRDFService);

  var container = Cc["@mozilla.org/rdf/container;1"];
      container = container.getService(Ci.nsIRDFContainer);

  var cu = Cc["@mozilla.org/rdf/container-utils;1"];
      cu = cu.getService(Ci.nsIRDFContainerUtils);

  var seq_node = rdf.GetResource("urn:notetaker:notes");
  var url_node     = rdf.GetResource(url);
  var chopped_node = rdf.GetResource(url.replace(/\?.*/,""));
```

```
  var matching_node, prop_node, value_node;

  if (!cu.IsContainer(ds,seq_node)) {
   throw "Missing <Seq> 'urn:notetaker:notes' in " +
          noteSession.config_file;
   return;
  }
  container.Init(ds,seq_node);

  // Try the full URL, then the chopped URL, then give up

  if ( container.IndexOf(url_node) != -1) {
    matching_node = url_node;
    this.url = url;
    this.chop_query = false;
  }
  else if ( container.IndexOf(chopped_node) != -1 ) {
    matching_node = chopped_node;
    this.url = url.replace(/\?.*/,"");
  }
  else {
    this.url = null;
    return;
  }
  else
    return;

  // Something found; grab all the note properties for it.

  var props = ["summary", "details", "width", "height", "top", "left"];

  for (var i=0; i<props.length; i++)
  {
    pred_node  = rdf.GetResource(ns + props[i]);
    value_node = ds.GetTarget(matching_node, pred_node, true);
    value_node = value_node.QueryInterface(Ci.nsIRDFLiteral);
    this[props[i]] = value_node.Value;
  }
}
```

First, this method readies the three main service objects that the RDF system provides. The `nsIRDFService` object is used to turn plain URL strings into `nsIRDFResource` objects, which are a subtype of the generic `nsIRDFNode` type. Most RDF methods do not accept string arguments; `nsIRDFNode` objects are generally required. We create such objects for both the full URL and the chopped URL. The sole use of the `nsIContainerUtils` interface follows. It is used to confirm that the `urn:notetaker:notes` resource is a container in the `notetaker.rdf` file. If this much is not in place, then there is a problem with that file, and the method aborts with an error. The `nsIRDFContainer` interface is then used to link the container (`<Seq>`) URI with the data source, and that link is initialized. Normally, access to the data source is on a

fact-by-fact basis. This last interface allows an RDF container and its members to be treated as though they were a data structure. That structure is the RDF equivalent of a table index. With that, the initialization part of the method ends.

The `if .. else` cascade contains the start of the scripted query. In this case, that query is quite trivial. It says: Search for a fully matching resource in the data source, and if that fails, search for a resource that matches the displayed URL without its query parameters. If both fail, give up.

The remainder of the query pulls out all the facts that represent property/value pairs for the found note. `rdf.GetTarget()` always returns an `nsIRDFNode`, so that object must be converted to the type we really expect for the property's value, which is a literal string. We're not storing window measurements as integers. Finally, those retrieved values are copied into the matching properties of the note object. This part of the code assumes that the note is well formed (properly created) in the `notetaker.rdf` file.

Overall, this query is a two-fact query that follows the general pattern of a simple syntax template query, except for the special checks at the start of the different URL strings.

If you test this `resolve()` method, perhaps by adding test code such as this

```
note.resolve("http://saturn/test1.html");
```

then the code will almost certainly fail with unexpected errors. Typically it is the first use of an RDF interface that fails, but failure might occur deeper in the code, or worse, intermittently. The culprit causing these failures is outside the note object—it is in the `noteSession` object. There, the data source for the RDF file is initialized in the `init()` function with this call:

```
this.datasource = GetDataSource(url.spec);
```

This initialization is wrong for our purposes. It causes the data source to be loaded asynchronously so that the fact store for that data source will only fill over time. Meanwhile, our scripts have raced forward and the note object is trying to probe the data source before it is ready. No wonder that RDF methods complain that expected containers or resources aren't present in the data source. The solution is to load the data source differently:

```
this.datasource = GetDataSourceBlocking(url.spec);
```

This causes a tiny delay when the browser window is first displayed, but it's livable for our simple case.

We could work around this tiny delay with a more sophisticated strategy that perhaps uses the `nsIRequestObserver` or `nsIStreamListener` interfaces of the `xml-datasource`'s XPCOM component. Those interfaces can be used to detect the ending of an asynchronous load. Some XPCOM objects created in this method are also created in other methods. Overhead could be

reduced by adding created XPCOM objects to the `noteSession` object, where they would be available for reuse. That's a job for another day.

In past chapters, we wrote scripts to push the note object's data out into the form fields and into the HTML document of the browser's GUI. Now we've connected the note object to the RDF fact store and configuration file that persistently holds the notes. As a result, the display of existing notes works. We only need to change a small omission in the `toolbar_action.js` file. In function `content_poll()`, this

```
display_note()
```

should read this

```
if (note.url != null ) display_note()
```

That leaves Web pages without notes free of any decoration. Much better!

16.6.5 When to Move User-Entered Data into RDF

In addition to displaying RDF content, the NoteTaker tool is designed to capture it. The last time this was properly organized was in Chapter 7, Forms and Menus, when we sent the captured data to a Web server. This session puts that data into an RDF fact store, and ultimately into a local file. The main alternate solution is to use a relational database.

For our purposes, entering data means adding it to a data source. It will sit in memory until either the user chooses an action that makes it permanent or the platform is shut down. That is a design choice.

Data can be entered either via the NoteTaker toolbar or via either panel in the dialog box. Let's look at each of these, starting with the toolbar.

The summary and keyword fields of the toolbar provide a quick way to create or update a note. Such a note can have its summary modified and a single keyword added. If the user fills these fields, but doesn't press any of the toolbar buttons (Edit, Save, or Delete), then nothing happens. Therefore, user changes to these fields can be handled in the commands available from the toolbar. There is no need for onchange event handlers or anything like that.

The Edit panel of the dialog box is the same as the toolbar. Changes made by the user only need to be recorded if the user presses the Ok button; they can be discarded if the dialog box is canceled. The Keywords panel, however, is more complex.

The Keywords panel allows any number of new keywords to be collected using the Add and Delete buttons on that panel. The question is: Where should these values be kept while the dialog box is displayed? If the user ultimately cancels the dialog box, these new keywords should be thrown away. If the user ultimately accepts the changes, these new keywords should be preserved. The problem is that we want the `<listbox>` and `<tree>` parts of the panel to update when keywords are added. This means that those keywords

must be stored in RDF where the templated tags can find them, even when we're not sure if they are ultimately to be kept or not.

In a nutshell, we have an undo or transaction rollback problem to solve. We want to be able to insert keywords into a data source where they're shared, but possibly remove them later if they're not ultimately wanted. The solution we choose is to implement a new command controller. That controller will record the keyword changes in an undo buffer. If it receives a rollback command, it will reverse all the commands made to date, and consequently any RDF changes. This solution is a design choice, but it is easily applied to most applications.

The result of this design is that all data entry processing is processed behind the command infrastructure. That is a very neat arrangement. In summary, data sits passively in form elements until the user causes a command to run. The command may push that data into a fact store where it is then more generally available across the application. This is particularly useful for templates. If the command is responsible for persisting the information, then the fact store will also be flushed to disk or sent over a network.

16.6.6 Enhancing Commands to Process RDF Content

Finally, we turn to the code that pushes data from the user to disk, rather than the other way around. We will update the `action()` function for the toolbar and the dialog box and implement a new controller for the special keyword support in the dialog box.

The toolbar `action()` function supports the `notetaker-open-dialog`, `notetaker-save`, `notetaker-display`, and `notetaker-delete` commands. Only `-save` and `-delete` require RDF processing. These two commands are quite lengthy, so Listing 16.15 only shows the simpler `notetaker-save` command.

Listing 16.15 NoteTaker script-based RDF update and save.

```
function action(task)
{
  var ns = "http://www.mozilla.org/notetaker-rdf#";

  var rdf = Cc["@mozilla.org/rdf/rdf-service;1"];
      rdf = rdf.getService(Ci.nsIRDFService);

  var container = Cc["@mozilla.org/rdf/container;1"];
      container = container.getService(Ci.nsIRDFContainer);

  var url_node;

// ... other commands removed ...

  if ( task == "notetaker-save" )
  {
```

```
var summary = document.getElementById("notetaker-toolbar.summary");
var keyword = document.getElementById("notetaker-toolbar.keywords");

var update_type = null;

if ( note.url != null )
{
  if ( keyword.value != "" || summary.value != note.summary )
  {
    update_type = "partial"; // existing note: update summary,
      keywords
    url_node = rdf.GetResource(note.url);
  }
}
else if ( window.content && window.content.document
     && window.content.document.visited )
{
  update_type = "complete"; // a new note
  url_node = window.content.document.location.href;
  url_node = url_node.replace(/\?.*/,""); // toolbar chops any query
  url_node = rdf.GetResource(url_node);
}

if ( update_type == "complete" )
{

  // add the note's url to the note container
  var note_cont = rdf.GetResource("urn:notetaker:notes");
  container.Init(noteSession.datasource,note_cont);
  container.AppendElement(url_node);

  // add the note's fields, except for keywords
  var names = ["details", "top", "left", "width", "height"];
  var prop_node, value_node;

  for (var i=0; i < names.length; i++)
  {
    prop_node  = rdf.GetResource(ns + names[i]);
    value_node = rdf.GetLiteral(note[names[i]]);
    noteSession.datasource.Assert(url_node, prop_node, value_node,
      true);
  }
}

if ( update_type != null)
{
  // update/add the summary
  var summary_pred = rdf.GetResource(ns + "summary");
  var summary_node = rdf.GetLiteral(summary.value);
      noteSession.datasource.Assert(url_node, summary_pred,
      summary_node, true);

  // begin work on a single new keyword
```

```
    var keyword_node = rdf.GetResource("urn:notetaker:keyword:" +
        keyword.value);
    var keyword_value = rdf.GetLiteral(keyword.value);

    // make this keyword related to one other keyword for this note
    var keyword_pred = rdf.GetResource(ns + "keyword");
    var related_pred = rdf.GetResource(ns + "related");
    var keyword2 = noteSession.datasource.GetTarget(url_node,
        keyword_pred, true);
    if (keyword2)
      noteSession.datasource.Assert(keyword_node, related_pred,
        keyword2, true);

    // add the keyword to this note
    noteSession.datasource.Assert(url_node, keyword_pred, keyword_node,
        true);

    // state the keyword itself
    var label_pred   = rdf.GetResource(ns + "label");
    noteSession.datasource.Assert(keyword_node, label_pred,
        keyword_value, true);

    // add the keyword to the container holding all keywords
    var keyword_cont = rdf.GetResource("urn:notetaker:keywords");
    container.Init(noteSession.datasource,keyword_cont);
    container.AppendElement(keyword_node);
  }

  // write it out

        noteSession.datasource.QueryInterface(Ci.nsIRDFRemoteDataSource)
        .Flush();

  note.resolve();
  display_note();
}
```

This code contains the RDF equivalent of one database transaction. It starts with some standard preparation—access to the XPCOM RDF interfaces—and then examines the GUI to see what kind of save is required. By collecting the summary field, the keyword field, the `note` and `noteSession` objects, and the state of the currently displayed URL, the code determines whether a note already exists. We cheat a little and reuse some information from the `content_poll()` function, such as the visited property.

If the note already exists, the only changes must be toolbar changes, so the saving of the note is a partial update of the existing note facts. If the note doesn't exist, then the note needs to be added (inserted), which requires a complete update of those facts. When a note is added from the toolbar, we also chop off any HTTP GET query string from the URL. At the end of all that examination, the `update_type` variable says what to do.

Because a partial update is a subset of a complete update, the partial case is shared by all updates. This branch in the code

```
if ( update_type == "complete" )
```

contains the complete update logic, except for the shared part; this branch

```
if (update_type != null )
```

holds the shared part used by both the complete and partial updates. Let's look at each one in turn.

The complete update code grabs the `urn:notetaker:notes` container and adds the URL for the note to it. That's one fact. It then steps through all the properties that a note has, except for the summary and a keyword. It adds those as well. That's five more facts for a total of six. All strings must be converted to `nsIRDFNote` objects or equivalent subtypes before they can be submitted to RDF.

The partial update code is then called in all cases where it's possible to create a note. You can't create a note for an `about:blank` URL or for an FTP site, so it's possible that the `notetaker-save` action will do nothing. The summary is straightforward—we just add one more fact. If the fact already exists, then the `Assert()` statement that adds it again will have no effect. By default, and in all normal circumstances, duplicate facts aren't allowed in a data source, so it's safe to assert a fact that might already be there.

The partial code then addresses the trickier matter of an entered keyword. If the note already has a keyword, then we want this keyword to be "connected" (related) to the other keywords in the note. That means a fact stating that (any) one keyword in the note is related to this new keyword. So we fish out an existing keyword value; if one's found, we add a fact stating the relatedness of the new and existing keyword. We do this first to avoid relating our new keyword to itself. That might happen if we added the new keyword first. The remaining code is straightforward: We add the keyword to the note; we add the keyword to the list of all keywords in the `urn:notetaker:keywords` container; and we add the keyword itself. That is four more facts.

At the end of this code, we've added 1 + 4 (+ optional 5) facts to the data source. Because the data source is based on the fully featured `xml-datasource`, these changes are automatically pushed to all templates using the data source. We then call the `Flush()` method to push the data source out to disk. Be aware that this command will write out the `notetaker.rdf` file with the facts in a near-random order, so any pretty formatting of that file will be lost. To finish up, we update and display the note, bringing our non-RDF data structures and the GUI into agreement with RDF.

That concludes the processing required for the `notetaker-save` command. The `notetaker-delete` command is equally detailed; the challenge in that command is to identify keywords no longer needed and to identify keywords still needed by other notes. That requires some analysis of the many dif-

ferent cases that are possible, which we won't do here. The Delete button on the Keywords panel has very similar logic; we'll discuss it shortly.

The Edit dialog box's `action()` function supports commands `notetaker-nav-edit`, `notetaker-nav-keywords`, `notetaker-save`, `notetaker-load`, and `notetaker-close-dialog`. Of these, the `notetaker-save` command is the only one requiring RDF work. In fact, we can reuse the `notetaker-save` command on the toolbar if we're organized enough: Listing 16.16 illustrates.

Listing 16.16 Improvements to the dialog box `notetaker-save` command.

```
if (task == "notetaker-save")
{
  var field, widget, note = window.opener.note;

  for (field in note)
  {
  widget = document.getElementById("dialog." + field.replace(/_/,"-"));

  if (!widget) continue;

  if (widget.tagName == "checkbox")
      note[field] = widget.checked;
  else
     note[field] = widget.value;
  }
  window.opener.setTimeout('execute("notetaker-save")',1);
}
```

This single extra line runs the toolbar's `notetaker-save` command. We can't call `window.opener.execute()` directly because the function would run in the dialog window's context. We want it to run in the browser window's context. Calling the browser window's `setTimeout()` method ensures that the right window context starts up when the timed command is run.

Finally, extra commands are required for the Keywords pane of the Edit dialog box. These commands will be collected into a controller that supports `commit` and undo operations. It will support these commands: `notetaker-keyword-add`, `notetaker-keyword-delete`, `notetaker-keyword-commit`, and `notetaker-keyword-undo-all`. Because these commands are closely tied together and will share data, it's not convenient to implement them separately in the `action()` function. Instead, they'll be implemented directly in the controller. We'll make a new file named `keywordController.js` for this controller. Listing 16.17 shows the structure of this controller.

Listing 16.17 Command controller for dialog box's RDF keywords.

```
var keywordController = {
  _cmds : { },
  _undo_stack : [],
```

```
  _rdf : null,
  _ds : null,
  _ns : "http://www.mozilla.org/notetaker-rdf#",
  _related : null,
  _label : null,
  _keyword : null,

  init             : function (ds) { ... initialize ... },
  _LoggedAssert    : function (sub, pred, obj) { ... },
  _LoggedUnassert  : function (sub, pred, obj) { ... },

  supportsCommand  : function (cmd) { return (cmd in this._cmds); },
  isCommandEnabled : function (cmd) { return true; },
  onEvent          : function (cmd) { return true; },
  doCommand        : function (cmd) {
     ... preparation code ...
    switch (cmd) {
    case "notetaker-keyword-add":
    case "notetaker-keyword-delete":
    case "notetaker-keyword-commit":
    case "notetaker-keyword-undo-all":
    }
  }
};

keywordController.init(window.opener.noteSession.datasource);
```

Like all command controllers, this controller has the standard four commands, starting with `supportsCommand()`. The `doCommand()` method implements a different case statement for each command attempted. The controller also has many custom features. The controller holds a number of variables, and an array called `_undo_stack` will hold the steps that need to be reversed. The `_LoggedAssert()` and `_LoggedUnassert()` methods perform RDF manipulation as for `Assert()` and `Unassert()`, but they also make a record of their actions in the undo stack. Let's first make the `init()` method, which is trivially shown in Listing 16.18:

Listing 16.18 Initialization of keyword command controller.

```
init             : function (ds) {
    this._rdf = Cc["@mozilla.org/rdf/rdf-service;1"];
    this._rdf = this._rdf.getService(Ci.nsIRDFService);
    this._ds = ds;
    this._related = this._rdf.GetResource(this._ns + "related");
    this._label   = this._rdf.GetResource(this._ns + "label");
    this._keyword = this._rdf.GetResource(this._ns + "keyword");
    window.controllers.insertControllerAt(0,this);
  },
```

This method assigns some handy objects to the controller—the RDF service, the supplied data source, and three commonly used predicate terms. The

controller then registers itself with the dialog window. By putting it first in the controller chain, we ensure that it is the first controller to be examined for any commands that might occur.

The next two functions `_LoggedAssert()` and `_LoggedUnassert()` show how a controller can retain and share information about the commands it executes. In this case, that information is `undo` history about the RDF facts asserted and removed by commands. Listing 16.19 shows these two functions.

Listing 16.19 Implementation of fact assertion undo log.

```
_LoggedAssert     : function (sub, pred, obj)
  {
    if ( !this._ds.HasAssertion(sub, pred, obj, true))
    {
      this._undo_stack.push( { assert:true, sterm:sub, pterm:pred,
          oterm:obj } );
      this._ds.Assert(sub, pred, obj, true);
    }
  },

  _LoggedUnassert   : function (sub, pred, obj)
  {
    if ( this._ds.HasAssertion(sub, pred, obj, true))
    {
      this._undo_stack.push( { assert:false, sterm:sub, pterm:pred,
          oterm:obj } );
      this._ds.Unassert(sub, pred, obj, true);
    }
  },
```

Each function is a simple replacement for `nsIRDFDataSource.Assert()` and `nsIRDFDataSource.Unassert()`. In both cases, the fact store is first tested to see if the RDF change would have any effect. If it would, then a record of the change to be made is created (as a four-property object) and that record is put on the `undo` stack. The property `assert` states whether the fact is asserted or unasserted. The genuine RDF change is then made as normal.

These two functions are directly complemented by the `notetaker-keyword-commit` and `notetaker-keyword-undo-all` commands. The fragment of the `doCommand()` method responsible for these two commands appears in Listing 16.20.

Listing 16.20 Committing and undoing fact changes using an undo log.

```
case "notetaker-keyword-commit":
 this._undo_stack = [];
 break;

case "notetaker-keyword-undo-all":
  while (this._undo_stack.length > 0 )
```

```
  {
    var cmd = this._undo_stack.pop();
    if ( cmd.assert )
      this._ds.Unassert(cmd.sterm, cmd.pterm, cmd.oterm, true);
    else
      this._ds.Assert(cmd.sterm, cmd.pterm, cmd.oterm, true);
  }
  break;
```

The `notetaker-keyword-commit` command is trivial; it forgets the existing undo commands so that they can't be accidentally undone. The `notetaker-keyword-undo-all` command is marginally more complex. It steps through the stack `Unassert()`'ing every previously `Assert()`'ed fact, and `Assert()`'ing every previously `Unassert()`'ed fact. At the end of this processing, no items remain on the stack, so in this implementation, it's not possible to "undo an undo."

Even though this undo system works on fact assertions, not commands, it is easy to see how the stack could hold records of whole commands as trivially as it holds records of whole facts. That possibility is also suggested in Chapter 9, Commands.

The remainder of the `doCommand()` method appears in Listing 16.21.

Listing 16.21 `doCommand()` initialization with save and delete operations.

```
doCommand        : function (cmd) {

  var url     = window.opener.content.document.location.href;
  var keyword = window.document.getElementById("dialog.keyword").value;

  if (keyword.match(/^[ \t]*$/))
    return;

  var keyword_node = this._rdf.GetResource("urn:notetaker:keyword:" +
          keyword);
  var keyword_value = this._rdf.GetLiteral(keyword);
  var url_node     = this._rdf.GetResource(url);

  var test_node, keyword2, enum1, enum2;

  switch (cmd) {

  case "notetaker-keyword-add":
    // This keyword should be related to an existing keyword, if any
    keyword2 = this._ds.GetTarget(url_node, this._keyword, true);
    if (keyword2)
      this._LoggedAssert(keyword_node, this._related, keyword2);

    // add this keyword
    this._LoggedAssert(keyword_node, this._label, keyword_value);

    // add this keyword to the current note.
```

```
    this._LoggedAssert(url_node, this._keyword, keyword_node);
    break;


  case "notetaker-keyword-delete":
    // remove this keyword from the current note.
    this._LoggedUnassert(url_node, this._keyword, keyword_node);

    // remove this keyword and related facts if it's not used elsewhere
    enum1 = this._ds.GetSources( this._keyword, keyword_node, true);
    if (!enum1.hasMoreElements())
    {
      // this keyword
      this._LoggedUnassert(keyword_node, this._label, keyword_value);

      // this keyword is related to that keyword
      enum2 = this._ds.GetTargets(keyword_node, this._related, true);
      while (enum2.hasMoreElements())
        this._LoggedUnassert(keyword_node, this._related,
          enum2.getNext().QueryInterface(Ci.nsIRDFNode));

      // that keyword is related to this keyword
      enum2 = this._ds.GetSources(this._related, keyword_node, true);
      while (enum2.hasMoreElements())

          this._LoggedUnassert(enum2.getNext().QueryInterface(Ci.nsIRDFNod
          e), this._related, keyword_node);
    }
    else // this keyword is used elsewhere.
    {
      // delete related facts where keywords that this keyword
      // relates to are only found in the current note.
      enum1 = this._ds.GetTargets(keyword_node, this._related, true);
      while (enum1.hasMoreElements())
      {
        keyword2 = enum1.getNext().QueryInterface(Ci.nsIRDFNode);
        enum2 = this._ds.GetSources(this._keyword, keyword2, true);

        test_node = enum2.getNext().QueryInterface(Ci.nsIRDFNode);
        if (!enum2.hasMoreElements() && test_node.EqualsNode(url_node))
          this._LoggedUnassert(keyword_node, this._related, keyword2);

        // delete related facts where keyword that relates to this
        // keyword are only found in the current note.
        enum1 = this._ds.GetSources(this._related, keyword_node, true);
        while (enum1.hasMoreElements())
        {
          keyword2 = enum1.getNext().QueryInterface(Ci.nsIRDFNode);
          enum2 = this._ds.GetSources(this._keyword, keyword2, true);

          test_node = enum2.getNext().QueryInterface(Ci.nsIRDFNode);
          if (!enum2.hasMoreElements() && test_node.EqualsNode(url_node))
            this._LoggedUnassert(keyword2, this._related, keyword_node);
        }
```

```
    }
  }
  break;
```

The ten or so lines of code prior to the `switch()` statement initializes some local variables and aborts the command if there's no current note. The switch statement shows the `notetaker-keyword-add` and `notetaker-keyword-delete` cases. Adding and removing keywords would be easy if we didn't try to maintain a sense of which keywords are related to which other keywords. That information makes the adding and removing tasks longer.

Both commands assume that a note either already exists or is currently being created for that URL. So the keywords added and removed are done so in the context of a particular URL. The comments in the code describe the steps involved, but here is a more explanatory discussion.

All this code is constrained by the fact that duplicate facts don't exist in a normal fact store. There are no variables in a fact store, so we can't set both A and B equal to 5. Every fact stored is unique. We need to manage the fact store globally; we must consider the impact of adding or removing a fact on all other facts in the fact store.

The keyword addition case is the easier case. It adds these facts to the data source:

```
<- keyword-urn, related, keyword2-urn -> (optional)
<- note-url, keyword, keyword-urn ->
<- keyword-urn, label, keyword-literal ->
```

We want to ensure that all related keywords can be found for a given note. That means any keyword belonging to a note with existing keywords must be related to all the other keywords in the note. In our RDF model, we capture this information by relating that one keyword to at least one of the other keywords for that note. So we first check for other keywords and, if there are any, relate the new keyword to one of them. After this is done, we're free to add this keyword in as well, first to the note as a fact object term and then in its own right as a fact that states that the keyword exists and what its value is. Each of the required facts asserted is done so through the `_LoggedAssert()` interface.

The keyword deletion case is quite complex. It is easy to remove information specific to one note, but keywords may be used by several notes, which means that keyword-to-keyword relationships can also be used by several notes. The first and last of the three facts stated in the addition case therefore can't be deleted without some careful checks of their use elsewhere in the fact store. How we proceed depends on whether the keyword to be deleted is used in other notes. The required tortured logic follows.

If the keyword in question is used only in a single note, then all information about that keyword is confined to a single note. We can delete all record of the keyword in a straightforward manner.

If the keyword in question is used in several notes, we proceed more care-

fully. We can remove the keyword reference from the current note, but we can't delete the keyword's own fact. Removing facts where this keyword is related to some other keyword is the hard part. If a keyword-related-to-our-keyword fact is used by another note, we can't remove it. We'll know if that fact is so used by checking the other keyword in the keyword. If that other keyword appears in any other note in the fact store, the fact applies to that other note as well as the current note, so leave it. Otherwise, remove it. We do this twice because the keyword to be deleted could be fact subject or fact object in such a related fact.

Astute readers will note that the `urn:notetaker:keywords <Seq>` should also be updated by these `-add` and `-delete` commands. We haven't done that because these commands are complicated enough as it is, and some trickery is required to fit those further updates in with the undo system. In fact, a more general solution is to couple data source observer objects with the undo stack—a project for another day. That concludes RDF enhancement of the NoteTaker commands.

To get all this working, we need to hook the new controller and command calls into the dialog box. Several pieces of code are required. The `editDialog.xul` file requires an additional `<script>` tag:

```
<script src="keywordController.js"/>
```

That file also needs extra handlers on the `<dialog>` tag. These handlers are for the keyword changes made in the dialog box.

```
<dialog xmlns="http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul"
  id="notetaker.dialog"
  title="Edit NoteTaker Note"
  onload="execute('notetaker-load');"
  ondialogaccept="execute('notetaker-keyword-commit');
                  execute('notetaker-save');
                  execute('notetaker-close-dialog');"
  ondialogcancel="execute('notetaker-keyword-undo-all');
                  execute('notetaker-close-dialog');"
  >
```

These handlers are getting large, and any further changes should probably be aggregated into single functions or some kind of transaction. The dialog box has some other handlers in the file `dialog_handlers.js`. Two of these other handlers reduce to trivial code now that the keyword controller has been written:

```
function add_click(ev)
  {
  execute("notetaker-keyword-add");
}

function delete_click(ev)
{
  execute("notetaker-keyword-delete");
}
```

With these last changes, the NoteTaker tool is complete—or at least as complete as space allows in this book.

16.6.7 Custom RDF Tree Views and Data Sources

In "Hands On" in Chapter 13, Listboxes and Trees, we briefly experimented with custom views. That experiment can be extended to RDF if desired. If that is done, then the `<tree>` tag with the custom view can be powered from an RDF data source without using any template. Space here does not permit a long examination of that implementation option, but a few remarks are worth making:

That Chapter 13, Listboxes and Trees, experiment implemented a method named `calcRelatedMatrix()`, which built information out of a constant JavaScript array named `treedata`. If that method is reimplemented to extract the pairs of related keywords from a data source instead of an array, then that experiment will work immediately, but using RDF data instead of JavaScript data.

Such a simple replacement strategy is, however, a primitive use of the facilities of data sources. A better solution is to use the `nsIRDFObserver` interface. If the JavaScript object implementing a custom view also implements this interface, then that object can be registered as an observer on a data source (it must be an `nsIRDFCompositeDataSource`). The view will them receive notification every time a fact in the data source changes and can incrementally update the tree's view rather than recalculate the whole view in one batch. That is a more sophisticated strategy that supports event management consoles and other "server push" data systems.

Finally, we point out that an object with the `nsIRDFDataSource` interface can be created entirely in JavaScript. Such an object can be used to pretend that the data it supplies is RDF-based. Alternately, such an object might wrap itself around one or more other data sources, in which case it is a variation on the composite data source implementation supplied by the platform. Either way, such an object can be lodged (via JavaScript only) with a template and can drive the appearance of the GUI just as the presupplied data source does.

16.7 DEBUG CORNER: WORKING WITH DATA SOURCES

Some of the most common problems that hit when working with data sources include the following:

- **Capitalization**. Unlike the rest of XPCOM, data source interface methods are stated in `InitCaps`, not in `initCaps`. Thus it is `GetResource()` not `getResource()`.
- **Asynchronous loading**. If the `nsIRDFDataSource` interface's `GetDataSource()` methods is used to create a data source instead of `Get-`

`DataSourceBlocking()`, the data source loads "in the background." In that case, any statements manipulating that data source immediately after its object is created are at risk. The risk is that the data source has not finished loading yet, and so not all anticipated facts may yet be present.

- **Syntax problems in test data**. If RDF files containing test data have syntax errors, then facts in that RDF file will be loaded into a data source only up to the point where the syntax error occurs. No error messages will be given.
- **Attempting to push content back over the Web**. Data sources originating from over a network cannot yet be "saved" back to their origin directly. Only local files can be saved (updated). To push a changed data source back over the Web (or over FTP), turn the data source into an RDF document using a content source interface and then use the file upload system. For a lower level solution, use a socket.
- **Using `false` as an argument to `Assert()` or `Unassert()`**. The fourth argument to these methods should always be true. Using `false` expands the logic system used in the RDF facts stores in an unhelpful way. This fourth argument says nothing about the existence of facts. Always use `true`.
- **Passing strings to `Assert()` or `Unassert()`**. These methods only accept objects of type `nsIRDFNode` and subtypes of that type.
- **Problems with multiple return values.** Methods such as `GetTargets()` return an `nsISimpleEnumerator` object that provides a list of possible URIs that fit the fact requested. Each object returned by this enumerator has interface `nsISupports`, and `QueryInterface()` should be used to extract a more useful `nsIRDFNode` interface, or a subtype of that interface.
- **The `nsIRDFContainerUtil` interface and objects implementing it generally have no life of their own**. They work on other objects passed in as arguments.

16.7.1 Revealing Data Source Content

RDF internal data sources are one of the trickiest aspects of the Mozilla Platform. Listing 16.22 is a piece of code that can be used to probe their contents.

Listing 16.22 Stream creation by many methods.

```
function _dumpFactSubtree(ds, sub, level)
{
  var iter, iter2, pred, obj, objstr, result="";

  // bail if passed an nsIRDFLiteral or other non-URI
  try { iter = ds.ArcLabelsOut(sub); }
  catch (ex) { return; }
```

```
  while (iter.hasMoreElements())
  {
    pred = iter.getNext().QueryInterface(Ci.nsIRDFResource);
    iter2 = ds.GetTargets(sub, pred, true);

    while (iter2.hasMoreElements())
    {
      obj = iter2.getNext();
      try {
        obj = obj.QueryInterface(Ci.nsIRDFResource);
        objstr = obj.Value;
      }
      catch (ex)
      {
        obj = obj.QueryInterface(Ci.nsIRDFLiteral);
        objstr = '"' + obj.Value + '"';
      }

      result += level + " " + sub.Value + " , " +
                pred.Value + " , " + objstr + "\n";

      result += dumpFactSubtree(ds, obj, level+1);
    }
  }
  return result;
}

function dumpFromRoot(ds, rootURI)
{
  return _dumpFactSubtree(ds, rootURI, 0);
}
```

The function `dumpFromRoot()` is the API to use. It relies on very few aspects of the `nsIRDFDataSource` interface and should work for most internal data sources and for all plain RDF files. It performs a recursive breadth-first search of a fact store using a given starting point and assumes that the RDF graph in the fact store is structured as a tree.

This function should be passed an `nsIRDFDataSource` object and an `nsIRDFResource` object. The data source the first object represents should be fully loaded, or else the report generated in the result string will be incomplete. The `rootURI argument` should be a URI that is suspected of being a container or container-owner in the RDF graph for the data source's fact store. The URIs listed in Table 16.11 are typical candidates. If the RDF graph contains cycles, then the code will recurse forever, probably crashing the browser eventually. It's only a simple testing tool, so treat it that way.

16.8 SUMMARY

The Mozilla Platform contains more object facilities than can possibly be covered here. Because of its portability requirements, and because of its application focus, those objects tend to be high-level ones. Perhaps one day there will be a full POSIX interface, but the high-level application focus of the platform reduces the urgency of any such need.

Mozilla's XML processing facilities are particularly strong, which is no surprise. Heavyweight XML-based networks tend to be business-to-business rather than consumer-to-business, but Microsoft's .NET initiative suggests that there is plenty of need for sophisticated XML-based client-oriented software.

Having explored first the front half and now the back half of Mozilla-based applications, we have only to deploy those applications. Mozilla's build system, which is used to create compiled applications, is complemented by a remote install system. That system, XPInstall, is the topic of our last chapter, which follows.