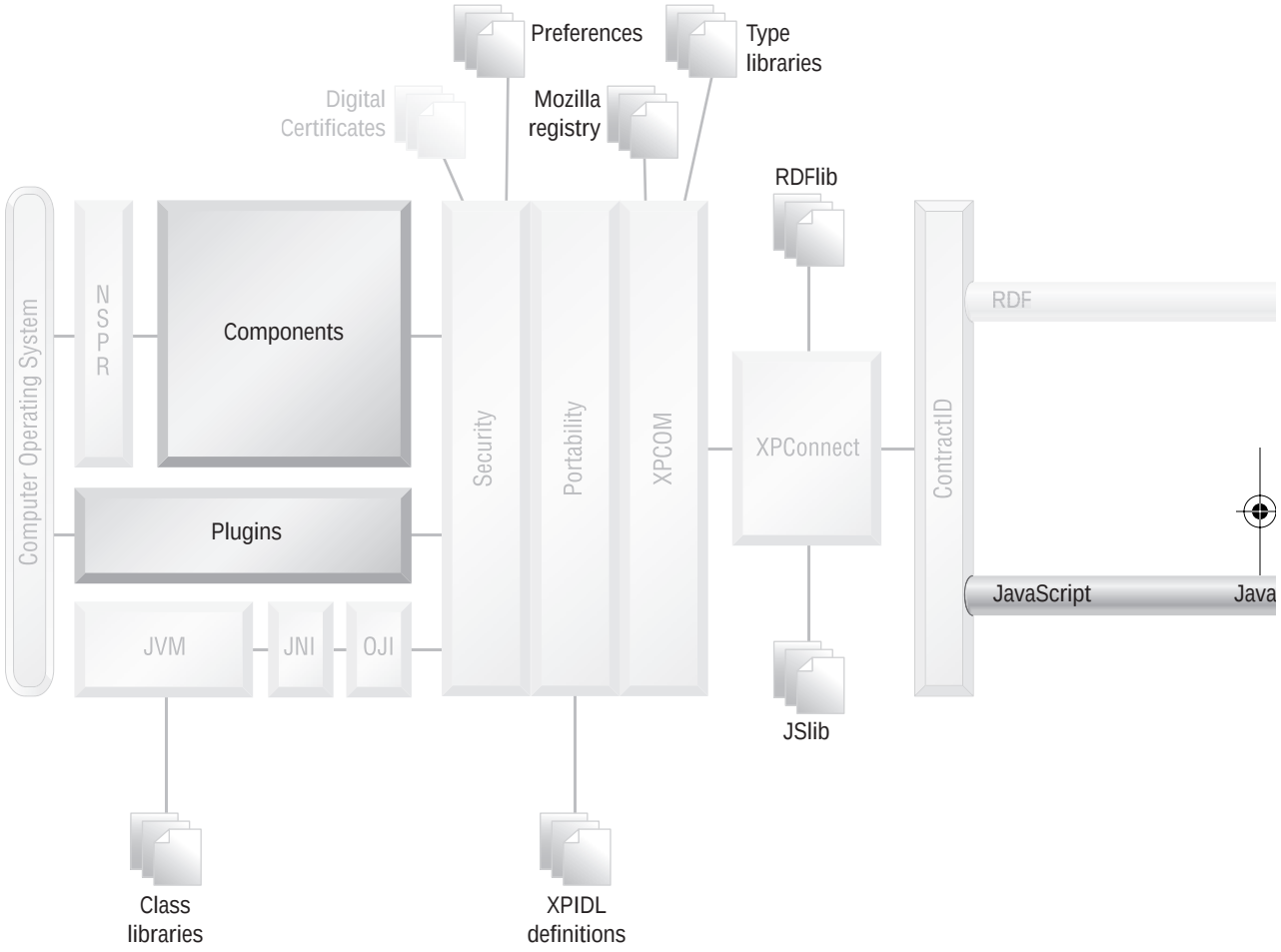
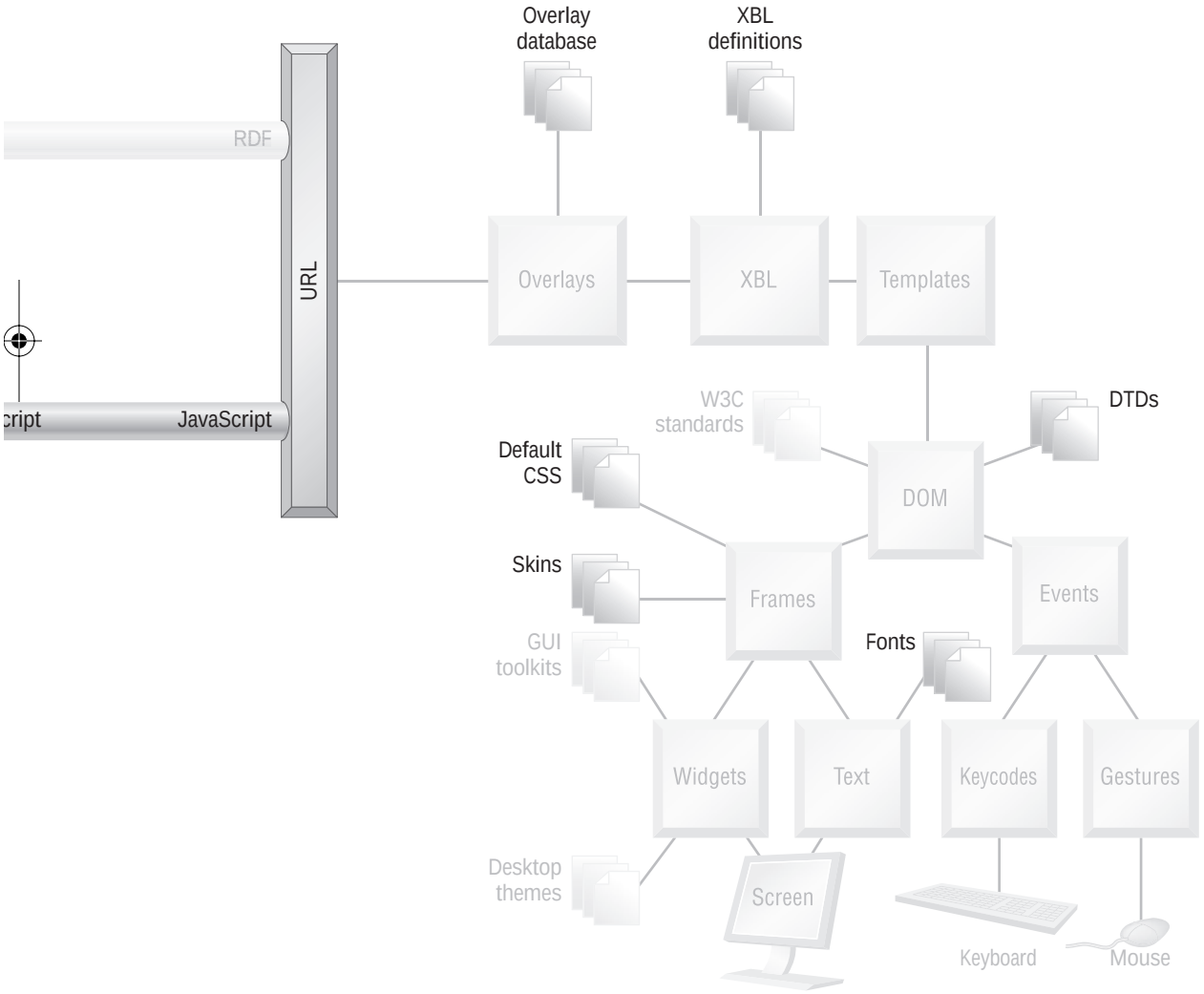

Preferences
Type libraries
Digital Certificates
Mozilla registry
RDFlib
Computer Operating System
NSPR
Components
Security
Portability
XPCOM
XPConnect
ContractID
RDF
Plugins
JavaScript
Java
JVM
JNI
OJI
JSlib
Class libraries
XPIDL definitions

CHAPTER 17

Deployment



This chapter describes how to distribute your application to the world from a Web site. Doing so is one use of Mozilla's XPInstall (Cross Platform Install) system.

Most Mozilla applications are built to be used. Before one can be used, it must be installed on some computer. Installation is part of the general problem of deployment. Mozilla supports a variety of deployment strategies. This chapter notes all those strategies but focuses on automatic deployment of applications from a URL served up by an ordinary Web server.

Deploying software from a remote server has been a glamorous idea ever since Java applets first appeared, and now is part of Microsoft's .NET strategy. Such an approach makes reaching the user or customer easy, reduces the cost of distribution to nearly nothing, and naturally fits with traditional client-server architectures. As the speed of the Internet increases, the arguments for locally managed applications weaken in favor of service providers, especially in business. Even when the application is a standalone one, a flow of patches, revisions, and news items can preserve a vital communication channel with the users.

Mozilla contains a portable installation system called XPInstall. There is no need for tools outside Mozilla like InstallShield or `rpm(1)`; XPInstall is all you need. XPInstall can be user activated from any running Mozilla application, like a browser, or it can accompany a batch-oriented standalone executable.

Deployment doesn't have the same glamour as throwing together a bunch of windows for a demo—deployment is supposed to just work. That supposedly simple goal, however, is a major test of your ability to be organized. If your deployment system is well organized, the world will give you silence, but may use your application. If your deployment system has any flaws at all, your application will probably sink without a trace.

Central to deployment is the idea of a bundle. A *bundle* is just a collection of files and scripts that make up an application. RPMs, tarballs, and executable archives are all examples of bundles. By talking about bundles, this chapter avoids clashing with other terms like *package*, *application*, and *installer*, which all have their own meanings within Mozilla. In Mozilla, bundles are XPI (`.xpi` suffix) files or executables. XPI is an acronym for Cross(X) Platform Install. XPI files are just ZIP files, with a few extra conventions imposed.

The other side of deployment is installation. In this chapter, installation means copying pieces of the bundle to the local computer. It is possible to bundle an application so that it can be installed on any operating system that Mozilla supports. Because operating systems have their differences, installation sometimes has to dip down into platform-specific details. Nevertheless, at least 90% of bundle preparation can be done in a portable way.

In the case of remote deployment, XPInstall retrieves XPI files like Java's JVM retrieves applet JAR files. Unlike JAR files, XPI files are not held within a "sandbox." They can install to any part of the platform or to any part of the underlying operating system. Both the platform and the operating sys-

tem can be damaged if a bundle is poorly organized. The safest and most common strategy is to install only into the chrome area.

Mozilla's Platform source code is not required to make a deployable application, but if a complex install strategy is chosen, then access to a working Mozilla compilation environment becomes more important.

The NPA diagram at the start of this chapter shows the impact that XPInstall has on the Mozilla Platform. From the diagram, XPInstall can be used to install all the various files that a running platform instance relies on. In fact, the whole NPA diagram could be highlighted because XPInstall can be used to replace even dynamic link libraries and executables—everything. Such broad-brush changes are a rare event, however. Two things are entirely missing from the diagram: application files and XPInstall itself. Application files, like XUL, CSS, RDF, and JavaScript files, are separate from the platform proper but are the most commonly installed files. XPInstall is a small world unto itself, which is why it doesn't appear. It is best seen as a specialized download tool like an FTP client or WinAMP.

XPInstall both uses and provides familiar technologies. The deployment process is scripted with JavaScript, and there are specialist objects available that assist. XPInstall presents a series of interactive windows to the user. A few XUL tags aim to do the same thing.

This chapter begins with a quick review of all the install options. It follows that with a description of remote install and finishes with a run-down of the technologies involved.

17.1 Overview of Install Strategies

Every option you can imagine is available for installing Mozilla-based applications. The options considered here are *no install*, *manual install*, *piggy-back install*, *native install*, and *custom install*. *Remote installs*, the main subject of this chapter, have a separate and extensive discussion of their own.

The simplest way to access a Mozilla-based application is with *no install* at all. In this case, the users already have the Mozilla Platform on their computers. Your application is served over the Web as a series of XUL files and their various inclusions, such as overlays, stylesheets, and scripts. To serve XUL documents from a Web site, just make sure that the Web server sets the MIME type for `.xul` files to be

```
application/vnd.mozilla.xul+xml
```

If suitable digital security is in place, such an application can have as much access to the local computer as any chrome-based application.

The Web is still slow, so some attention to performance helps. XUL documents, scripts, and stylesheets are cached in the browser cache just as all Web documents are, so if cache space is available, that is a start. Correct cache settings in the client browser and on the server can reduce to near-zero the over-

head of downloading. On the client, the XUL cache should be enabled, and Quick Launch functionality should be turned on for the Windows platform. FastLoad raises the platform into memory at operating system boot time, just like Internet Explorer. FastLoad is enabled when Mozilla or Netscape is first installed; it is a simple preference. XUL applications can also be deployed as JAR files, which further assists with performance. The URL of a file stored in a JAR file has the form

```
jar:{url}!{path}
```

where `{url}` is the location of the JAR archive, and `{path}` is the location of a given file within it. If the JAR file `example.jar` is stored at the top of the chrome and contains the file `test/sample.xul`, then the full path name for `sample.xul` would be

```
jar:resource:/chrome/example.jar!test/sample.xul
```

As discussed in Chapter 12, Overlays and Chrome, the `resource:` URL scheme is a scheme that points to the top of the platform install area, and `chrome:` URLs are typically mapped to `resource:` URLs.

If the `-chrome` command-line option or an application-specific command-line option is used (like `-jsconsole`), a XUL application may be started so that no hint of browser-like functionality appears. Such an application looks the same as any native executable.

If an install system seems like a good idea, then the options are as follows.

If a local install is required, then the most primitive solution is to do it manually. *Manual installation* requires access to the file system and operating system shell of the target platform.

There are good reasons for doing a manual application installation. For developers, it is a quick way to test work that is in progress. For system administrators, manual installation steps can be rolled up into existing deployment tools and processes. For systems integrators, the resources used by manual installations are the COTS (common-off-the-shelf) interfaces needed to combine applications into larger, integrated systems.

The different kinds of manual installations are *platform install*, *component install*, *application install*, and *security install*. Platform install means installing the Mozilla Platform itself. This always requires an operating-system–specific executable and is called a *native install* here. After that is done, other types of manual installations are possible.

Manual component install adds new XPCOM components and interfaces to the installed platform. These components are then available to all applications installed. To install new components and interfaces:

1. **Create a Mozilla module**. A module is an implementation of one or more components and one or more interfaces, so this is a programming task. The module will be either a JavaScript `.js` file or a compiled language like C or C++.

2. **Create components**. To do this, turn the module into executable code. Nothing needs to be done for a `.js` file. C/C++ implementations need to be compiled into a dynamic link library using the Mozilla build environment.
3. **Create interfaces**. An interface is specified in an `.idl` file, which the component creator must write. Run the XPIDL `.idl` file through the `xpidlgen` tool to produce an `.xpt` type library file. `xpidlgen` is part of the build environment of Mozilla, so interface creation requires that you build the Mozilla source, or find a build with debug turned on.
4. **Copy the `.js` and `.xpt` file to the `components` directory under the platform install directory**. Alternately, copy the dynamic link library there. Ensure that copied files are readable (and executable if libraries).
5. **Run the tool `regxpcom` from the Mozilla install directory**. This tool is supplied with the platform. It generates manifests in files called `compreg.dat` and `xpti.dat`.
6. **Restart the platform**, which then benefits from the new `.dat` files. The component and its interfaces can now be used from scripts.

The Mozilla source code contains an example component called `nsSample` with an example interface called `nsISample`. See the source code directory `xpcom/sample`.

Manual application install adds new XUL-related files to the chrome. Such files are divided into packages. To add packages to the chrome, follow these steps:

1. **Create a package**. A package is a set of files layed out using the standard chrome directory structure of content, skins, and locales. Such files may be bundled into a JAR archive or left as a simple hierarchy of folders and files.
2. **Create and add `contents.rdf` files to the package**. A package is not really a package without these files. There should be one for the content directory, one for each skin, and one for each locale.
3. **Copy the package or folder hierarchy to the chrome directory**.
4. **Update the file `installed-chrome.txt`**, located in the chrome directory. At most one line should appear for each of the content, locale, and skin subparts of the package.
5. **Delete the `chrome/overlayinfo` directory**, if this package has been installed before (you are updating it). This will cause the overlay system to be recalculated.
6. **Restart the platform**.

All of these steps are illustrated with the NoteTaker examples in the first five chapters of this book. Chapter 12, Overlays and Chrome, discusses the mechanics of the chrome registry, which is the user of these files.

Lines added to `installed-chrome.txt` should be of the form

```
skin,install,url,{url}/
locale,install,url,{url}/
content,install,url,{url}/
```

where `{url}` must refer to a local directory and should generally be a `jar:` or a `resource:` URL. The `resource:` scheme points to the very top of the Mozilla installation area—the parent directory of the chrome directory. Locale URLs must include a locale name, like `en-US`; skin URLs must include a theme name, like `modern`.

A security install may be required for applications installed outside the chrome. Such applications can be installed on a local disk, or served from a Web site, and still run in a secure environment. If this secure environment is to be available without user effort, custom configuration files must be added to the platform. Such files must be created manually but can be installed either by hand or in an automated way.

To create these files, start by creating a new user profile using the Mozilla Profile Manager. Copy all files in that profile to one side so that the originals are preserved. Next, install the application in its final location. Run the application using that profile, with no special preferences or other changes. Every time the platform asks you to grant security access (perhaps to a digitally signed file or to a form submission), agree. Every time the platform offers to remember such a decision, confirm that it should. When all security aspects have been run through, shut down the application and copy any files in the profile that have changed from the originals. These are the files that need to be manually installed into the user profile on all computers to which they are deployed. Alternately, these can be installed into the default user profile.

Both piggy-back and native installs require use and modification of the Mozilla build environment. The build environment is not addressed in this book, but a few remarks are worth making.

A piggy-back install is a normal distribution of Mozilla modified to include extra application files. Such a distribution has two strengths: It contains an absolutely standard version of the platform (and Mozilla application suite), and it collects all required install tasks together into one familiar bundle. When the platform is installed, the additional applications are automatically available.

To make this work, the Mozilla build system must be altered. Fortunately, some of that system is data-driven. A small start is to look at example files under the directory `xpinstall/packager`. At least three changes are required:

1. The manifest that lists all the files to put into the final bundle must be updated. Files like `packages-unix` should be changed to do this.
2. The configuration of the interactive wizard that installs the platform must be updated. This configuration exists in files with `.it` suffixes.

3. The additional application files must be made available so that they can be included. Somehow they must appear in the `dist` (distribution) directory created by the build process at the top of the source tree. That directory is where the results of the make process are collected together. Hand-copying the required files is at best a temporary hack, but it can work.

If these first steps are done correctly, then a full compile of the product will produce a modified installation, but with one caveat. That caveat is that the build system has many subtlies and so no quick changes are immune from problems. Extensive study is likely before all this will work in a polished manner.

A native install involves a core part of the XPInstall subsystem. This core part is a small piece of platform-specific code. That code does not use XPCOM or any of the facilities of the Mozilla Platform; it is an independent program with its own support for TCP/IP, FTP, and HTTP. To do anything with this code requires familiarity with the Mozilla source code. It can be used three ways:

- In a full install, this XPInstall code is part of a large archive holding all the platform and application, which on Microsoft Windows is also an auto-install binary.
- In a so-called stub install, the distribution file is small (a stub) and contains this XPInstall code and a little configuration information. When the installation is started, the code connects to the Internet and downloads the platform components selected by the user.
- In an application install, this XPInstall code is used as a specialist installer for a particular application, separate from the core platform. This use is as close as XPInstall gets to acting like InstallShield or the `rpm` system. This use is not yet common and is really a variation on a full install.

The Netscape Client Customization Kit can slightly customize a native install, provided it is based on a Netscape 7.0 release. This tool is available at *http://devedge.netscape.com*. It has a very restrictive license, which makes it nearly useless for Open Source purposes, or even for commercial purposes.

XPInstall's native code also has some portable features. It contains a JavaScript interpreter, some XUL-like GUI code, some objects, and some operating system access. Together these are enough to extract the contents from one or more XPI bundles and to place them in the operating system's file system. This portion of XPInstall deploys both the platform and applications in the style of InstallShield.

The remote install case that is examined in this chapter uses this same native infrastructure. Not only is that infrastructure available in an installation binary, but it is also available inside the running Mozilla Platform. A hook in the browser object model allows an XPI file to be passed to this special native code, causing installation to commence.

Finally, a Mozilla application can ignore the XPInstall system altogether. That is a custom install. XUL and XPCOM technology is sufficiently powerful that all steps required to install an application can be done from the chrome. If you really need your own installation system, then there is nothing stopping you from creating one. The XUL tags described under the topic "Install Technologies" allow a dialog window that acts like an installation wizard to be easily created.

Netscape's Smart Update feature is an application built on top of XPInstall.

17.2 Steps Toward Remote Deployment

Overall, remote installation is an information distribution method that follows the publishing model of the traditional media. To build an application bundle for remote deployment via the Web is to be a publisher.

To deploy an application remotely, several things must happen. The application programmer, in the role of a release engineer, must prepare a little and write two scripts. The end user must agree to install the application. The platform itself must act on the instructions provided.

Remote deployment relies on the use of a URL. Mozilla's remote deployment system can use a `file:` URL just as easily as an `http:` URL. Therefore, all the remarks made here can also be applied to a deployment that starts and ends on the local file system.

17.2.1 What the Programmer Does

Here are the tasks that the programmer needs to complete so that an application can be deployed remotely by the user. The deployment system can be developed in parallel with other application development tasks.

17.2.1.1 Assigning Names and Versions The first step of deployment is to give the application names. Several names are needed for XPInstall to work. These names include a *text name*, a *package name*, a *registry application name*, and a *version*. Platform-specific names are also required. On Microsoft Windows, a Windows registry key is useful. Macintosh Aliases and Microsoft Windows Shortcuts might also be required. It is sensible for all these names, except for versions, to have the same root. A root is just a word that other words grow from. Other marks, like mastheads, brands, and command-line names aren't an explicit part of XPInstall. XPInstall does not support a graphical representation of the application, such as an icon.

The text name is a Unicode string that appears in the dialog boxes that XPInstall presents to the user. Because it is Unicode, it can contain © or ® or ™ symbols, among others. XPInstall presents this string in Latin left-to-right order, which is restrictive for some languages. An example text name is

```
Frederick's Amazing Shopping Spree System, Gold Version
```

The package name is the name of the chrome package that the application will be installed under (assuming that it is to go into the chrome). It is an 8-bit extended ASCII name and, for absolute portability, should be 8 characters or less and alphabetic (UNIX: 14 characters or less). Because it is used as a folder (file system directory) name, it shouldn't contain any punctuation, except perhaps the underscore. There isn't any need to include a version number (e.g., `netscape7`) in the package name, unless two different versions are to be installed at the same time. If the application is not installed under the chrome, then an install directory name has much the same rules as a package name. An example package name is

```
fredshop
```

The registry application name is a name that the Mozilla Platform uses to manage the application on the local host. It is used for version management, installation, and uninstallation. Mozilla's registries are described later in "Install Technologies." A registry name looks like a UNIX path, except that it can include Unicode characters. It is encoded in UTF8 inside the registry. Most registry application names follow a syntax convention like this:

```
/Application Publisher/short-name/subproduct
```

The `Application Publisher` part might be corporate or technical. A corporate version might just state `Alpha Trading Company`. A technical version might be a domain name in a similar convention to Java packages, like "`mozilla.org`". The short-name part is the application's name and usually is similar to the package name (e.g., `Navigator`). The subproduct part is optional and is used where the application is a suite of tools—one such tool might be a subpart of the larger application. An example from Mozilla is

```
/mozilla.org/Mozilla/JavaScript Debugger/Venkman Chrome
```

This example shows subproducts nested two levels deep. `Venkman Chrome` is a subproduct of the `JavaScript Debugger`, which is a subpart of Mozilla-the-application.

If the leading / is left off, then the path is considered to be a subpath and will be prefixed by `/mozilla.org/Mozilla/` (or by `/Netscape/` for Netscape versions of the platform).

In fact, the registry application name can be any path delimited by forward slash characters; there isn't any implicit meaning to the first or subsequent parts. The name does not need to match a directory name; it is just a hierarchical key in the style of Windows Registry keys. It has a length limit of about 2,000 characters. An example is

```
/Fred's Pyramid Company/Amazing Shopping Spree System/Gold Version
```

Mozilla application versions have a fixed format, which is a four-part number. A version can be expressed as four 32-bit integers or as a string of

period-separated integers. The string should be easily convertible to four integers—it should not have a "beta" suffix or other junk. The string version has the format:

```
"{major}.{minor}.{revision}.{build}"
```

- The major number starts from 0 and should change only when the application's designer significantly changes the application's design.
- The minor number starts from 0 and indicates feature additions to the base application. Applications based on earlier minor numbers should be able to interoperate with this version.
- The revision number starts from 0 and indicates bug fixes and trivial changes at most. Except for these fixes, earlier versions should operate identically.
- The build number is used to track a specific attempt at generating the application from its source. It comes from the build process and is a unique key of some sort.

An example version string is

```
"1.0.2.20021018"
```

Many unenforced conventions apply to these numbers. Some of these conventions follow:

- The build number uniquely identifies a single compilation or packaging pass. It might be a sequence number or a date. A 32-bit integer is big enough to hold a decimal number consisting of digits from the sequence full year-month-day-hour, until at least year 2200. An example for 9 A.M., 28 February 2003 is 2003022809. This is the system that Mozilla uses.
- The Linux kernel and some other products use even minor numbers to indicate a stable release, and odd minor numbers to indicate an in-progress work. Mozilla does not use this system. Minor numbers always indicate user releases.
- If the major number is zero, the product is considered to be under fundamental construction. No user should expect such an application to work with any previous version, no matter how minor. Any such compatibility is just a lucky convenience.

17.2.1.2 Organizing the World

Naming an application is easy. Organizing one is harder.

It is in the application developer's best interest to ensure that when the user deploys the application, that user has a good experience. This is because when users choose to deploy, they also choose to trust the Web site that offers the application. That trust must be maintained, or a bad reputation will be the only result.

Therefore, the second step for deployment is a release review. For the deployment strategy to work cleanly, the release engineer must have a clear understanding of what is being deployed and where. That means capturing some configuration information. The README document supplied with the Mozilla browser suite is an example of this information, but such thinking should go further. Three sets of information need to be captured: a *baseline*, a *footprint*, and a *target*.

A *baseline* is a reference point for the origin of an application bundle. It can be as simple as a CD burn of all the source files in the application or as organized as a CVS tag that includes a fully automated build and bundling system. If you can't re-create an application's bundle reliably, you can't test the deployment system properly or offer patches later. If your application is based on an Open Source license, then you are required by that license to produce a baseline ("the source") and to make it available to everyone.

For simple applications, just copy the source to a backup before each release.

A *footprint* represents the impact an installed application has on the end user's computer. It is a list of all the files on the destination computer that are affected by the application installed. The purpose of a footprint is to nail down every place on the user's computer that might be modified by the installing or running application. When user number 52,345 rings up for help with a messed-up PC, the footprint describes the boundaries of the problem space.

Examples of common footprint items are the Windows Registry, desktop shortcuts, `.ini` or `.rc` files, environment variables, global MIME types, and boot scripts. Inside Mozilla, running applications might also affect preferences files, security settings, and local MIME types. Applications might add components to the Mozilla components directory, add special-purpose binary utilities, or modify shell scripts used to start the platform.

If a bundle deploys files outside the platform install area, then those files should not automatically be put under `C:\Program Files` (Windows) or under `/usr` (UNIX). That is a very irritating practice for IT people who must configuration-manage their systems. Instead, ensure that the application can be entirely installed under a single folder of the user's choosing. Never put files in `/etc` or `C:\Windows` or `C:\Winnt` unless it is impossible to avoid doing so.

For simple applications, the footprint should go no further than the chrome directory and Mozilla registries. If users elect to save application-generated files elsewhere, that is their business.

Finally, a *target* is a description of the computing environment for which the application is designed. Such a description includes hardware, operating system, existing applications, specific files or configurations, required disk space—everything. A target description is the basis of a README file that the user might see. It is also used to identify checks that installation scripts need to do.

For simple, portable applications, a target consists of no more than a minimum version of the Mozilla Platform.

These three documents keep the application deployment process orderly and sane. You don't want the wrong software on the wrong computer complicating the wrong operating system files.

17.2.1.3 Scripts To automate application deployment, you must write up to two scripts.

The first script runs in an ordinary HTML Web page or in a XUL document. No special security arrangements are necessary. This script works only when the page is viewed with Mozilla technology. The rest of the page holds an invitation to the user to grab the application. The second script, called `install.js`, runs inside the XPInstall part of the platform, where it is isolated from the Web and from XPCOM.

Both scripts benefit from JavaScript host objects. The objects noted in the following overview are covered in full in "Install Technologies."

The first script is a so-called trigger script; it starts the application deployment. Two objects are available, and a third must be created from pure JavaScript. The two objects available are the `window.InstallTrigger` object and the `InstallVersion` object, which is also available as `window.InstallVersion`.

The `InstallTrigger` object contains diagnostic methods, plus the `install()` method, which starts the download of XPI files and eventually calls one or more copies of the second script, `install.js`. The diagnostic methods can be used to do some basic version checks and to tell whether `XPInstall` is enabled.

The `InstallVersion` object is a convenience object that can compare two application versions and report which is greater and which of the four version numbers are different.

The third object, which the application programmer must make, has this form:

```
var xpi_container = {
    "Test app part 1" : "URL1",
    "Test app part 2" : "URL2",
    ...
}
```

This object represents all the XPI files that together make up an application and is passed to the `InstallTrigger.install()` method. The preceding example object contains two properties, and so represents two XPI files. Any number of properties greater than zero is allowed. Because both properties have names that are literal strings, they can only be read using array notation like this:

```
var url = xpi_container["Test app part 1"];
```

Each property name is a text name for that application component, and the user will see it. Each property value is a URL (relative or absolute), which must be an XPI file. The URL may have a parameter string appended. That parameter string starts with ?, which is the same as parameters in an HTTP GET request. The remainder of the parameter string can follow HTTP GET syntax, or it can be any string (although that is a poor design choice). An example URL is

```
/downloads/apps/mozilla/shopcart/main.xpi?java=yes;flash=no
```

This URL is a relative URL, so Mozilla will add `http:` and a domain. In this example, the trigger code has detected the presence of Java and the absence of the Flash plugin and has passed that information to the second script `install.js` via parameters `java` and `flash`. The parameter portion of an XPI URL is passed directly to the `install.js` script without further processing.

All these objects are combined into a function that is usually called from an `onclick` handler on a link or a button. Listing 17.1 is a skeleton of such a function that shows most of the functionality it might contain.

Listing 17.1 Skeleton for a full-featured XPInstall trigger script.

```
function deploy()
{
  if ( !is_moz_browser() ) { return false; }
  if ( !window.InstallTrigger.enabled() { return false; }
  if ( !is_target() ) { return false; }
  if ( !is_app_version_ok() } { return false; }

  var      error_flag = false;
  function error_handler(url, err) { error_flag = true; };

  calculate_params();
  var      xpi_container = { ... };

  with (window.InstallTrigger)
    install(xpi_container, error_handler);

  return !error_flag;
}
```

Most of the functions used in this script need to be filled out for each application. The initial series of tests aborts the deployment if anything is wrong, including problems with the user's computer and problems with applications already installed. The `error_handler()` function is simplistic and can be made more complex if necessary. The `calculate_params()` function prepares whatever information needs to be passed to the second script. That information is used in the creation of the `xpi_container` object. Finally, `install()` is called to make the whole thing go. Of course, nothing happens if JavaScript is disabled, or if the preference `xpinstall.enabled` is set to `false`.

Testing the user's computer to see if it matches the target is a challenging task. The browser environment is limited to Web Safe scripts, and the XPInstall environment cannot access XPCOM components. Tests may need to be split over both places. If complex testing is needed, then write a separate application devoted to platform testing and ask the user to install that first. That application can then be used in trigger scripts from then on.

The second script to be written is always called `install.js`. Each XPI archive must contain one of these scripts. This second script is responsible for putting each file in the XPI archive in the correct spot in the local file system. The file-copying work is not done directly in the script. Instead, the script provides the XPInstall system with a series of file placement instructions. When all the instructions are scheduled, XPInstall is told to go ahead and do them all. When that happens, XPInstall automatically executes, records, and logs its actions; handles errors; and saves information for a future uninstall. At any point before the go-ahead is given, the `install.js` script can abort the installation. `install.js` can hack on the operating system a little as well.

The environment that `install.js` runs in is very restrictive. It runs in a separate JavaScript interpreter context and has its own global object, which is not an HTML or XUL window object. Only a few objects exist with which the script can work. Their names are

```
Install InstallVersion File FileSpecObject WinProfile WinReg
```

The `Install` object is the global object for the JavaScript context, so its methods can be called directly as though they were functions. It is the central object and has factory methods that can be used to create objects of the other types.

The `Install` object has useful properties. The `platform` property states the operating system. The `arguments` property contains any parameters, and the `url` property contains the full XPI URL.

The `Install` object also has useful methods: `initInstall()`, which primes XPInstall so that it is ready to accept instructions; `cancelInstall()`, which aborts everything; `performInstall()`, which runs the install according to instructions; and `uninstall()`, which removes applications. There are also useful diagnostic features.

The `File` and `FileSpecObject` objects are separate from any XPCOM file concepts—they are separate and different implementations with similar features. They can be used to manipulate files and directories anywhere on the local computer. Some special names that allow this to be done portably are available. The `File` object can also perform a few tests on the operating system and other miscellany.

The `WinProfile` and `WinReg` objects are Microsoft Windows specific. `WinProfile` provides read/write access to an `.INI` configuration file, and `WinReg` provides read/write access to the Windows Registry.

Depending on its contents, the `install.js` script might require that the platform be restarted; it also might require that the platform reassess the

chrome or plugins when that restart happens. None of these side effects affects the processing of the script itself. As for the trigger script, `install.js` has a standard pattern of use. Listing 17.2 shows this pattern.

Listing 17.2 Skeleton for a full-featured XPInstall `install.js` script.

```
var TEXT_NAME = "Test Application Release 3.2";
var REG_NAME  = "/Test Company/Test Application";
var VERSION   = "3.2.0.1999";
var params;
var rv = SUCCESS;

function prepare()
{
  if ( !(params = parse_args())) return INVALID_ARGUMENTS;
  if ( is_target() != SUCCESS )   return getLastError();

  initInstall(TEXT_NAME, REG_NAME, VERSION);

  /* -- as many functions like this as required -- */
  if ( schedule_folders()!= SUCCESS) return getLastError();
  if ( schedule_files() != SUCCESS ) return getLastError();
  if ( modify_os() != SUCCESS )      return getLastError();
  if ( run_any_programs() != SUCCESS)return getLastError();
  if ( register_chrome() != SUCCESS) return getLastError();

  return SUCCESS;
}

rv = prepare();
(rv == SUCCESS) ? performInstall() : cancelInstall(rv);
```

The script relies on error codes that the `Install` object maintains and returns when something goes wrong. The first step is to ensure that any arguments passed from the trigger script are in good order, and that the user's computer is suitable for install. If that is all okay, then `initInstall()` primes XPInstall to receive install instructions. Each function that follows does part of the work required to set up the installation. When these functions are called, few errors result because each installation instruction is only recorded, not performed. Finally, if all goes well, everything is run at once with `performInstall()`. Not shown are logging messages that can be recorded to a file with the `logComment()` method, or progress reports that can be sent to the user with `alert()` or `confirm()`. `prompt()` is not available as a method.

The simplest version of this skeleton, useful for testing, is shown in Listing 17.3.

Listing 17.3 Complete `install.js` script for a simple chrome application.

```
var TEXT_NAME = "Test Application Release 3.2";
var REG_NAME  = "/Test Company/Test Application";
var VERSION   = "3.2.0.1999";
```

```
var rv = SUCCESS;

function schedule_folders()
{
  var tree = getFolder("Chrome"); // Special keyword
  setPackageFolder(tree);
  addDirectory("chrome");         // topmost directory
}

function prepare()
{
  initInstall(TEXT_NAME, REG_NAME, VERSION);
  if ( schedule_folders()!= SUCCESS) return getLastError();
  return SUCCESS;
}

rv = prepare();
(rv == SUCCESS) ? performInstall() : cancelInstall(rv);
```

This example cuts down the `prepare()` function so that it is nearly trivial and adds an implementation of the `schedule_folders()` function. That implementation contains the critical step of matching a directory hierarchy in the XPI file against a directory hierarchy on the local file system. The hierarchy in the file will be copied to that file system.

The mechanics of this process are as follows. The special keyword `Chrome`, one of only a few such keywords, is used to pick out the chrome folder in the platform's install area. This keyword is independent of operating system, but other keywords exist that are operating system specific. That folder is made of the target directory (effectively the current directory). Finally, the folder named "`chrome`" in the XPI file is singled out to be copied. In fact, only the children of that XPI folder (and all their descendants) will be copied.

In Listing 17.3, all files in the XPI bundle (except `install.js`) have as their topmost directory the name `chrome`. That part of the path could be replaced with `X` or `Part1` or any other text string because it is just a placeholder. Everything will still work, provided that the `addDirectory()` call is changed to match.

If the XPI file is layed out using "`chrome`" as the placeholder, then good choices of file names within it are

```
chrome/content/TestApp/TestApp.xul
chrome/locale/en-US/TestApp/master.dtd
chrome/skin/classic/TestApp/global.css
```

Folders in the file system are matched against folders in the XPI bundle, and this can be repeated several times in the one bundle. An XPI bundle can contain several trees with different topmost directories. For example, if an XPI contained three installable subtrees, each subtree can be matched to a different location on the local disk. Such an XPI file might contain

```
install.js
subtree1/file1
subtree1/file2
subtree2/file3
subtree2/file4
subtree3/file5
subtree3/file6
```

Each subtree of two files can be placed in a different location by the one install script.

Finally, if an application is to be uninstalled, then the `uninstall()` method can also be used between `initInstall()` and `performInstall()`. It schedules an instruction that will uninstall an application based on historical information in the Mozilla registry. That uninstall will also occur when the other scheduled instructions are executed.

17.2.1.4 XPI Files An XPI file has the format of an ordinary ZIP file. Use WinZip, `pkzip`, or similar programs on Microsoft Windows; use `zip(1)`, not `gzip(1)`, on UNIX. Path names in ZIP files are always relative paths.

Such a file has one content requirement: It must contain at the top level an `install.js` file. It is common practice that the rest of the content directory structure matches the directory structure of the platform's installation area. If the XPI file directory structure doesn't match that area, then the `install.js` script will be more complicated.

It is possible to sign an XPI file digitally. Digital signing is done with the Netscape SignTool tool, as for all digitally signed files in Mozilla, but there is one restriction. The digital signature must be the first item in the XPI ZIP file. The digital signature is a file with path `META-INF/{signature}`, where `{signature}` is a file name indicating an encrypted signature of a type that Mozilla supports. Be aware that tools like WinZip use sorted views that can confuse the apparent order—display the ZIP file in "original order" to be sure or use `unzip(1)` or `pkunzip`. For more about SignTool and digital signing, consult *http://devedge.netscape.com*.

Figure 17.1 shows the contents of the XPI bundle for the Chatzilla instant messaging client. It is a ZIP file. This bundle is typically installed when the Classic Mozilla application suite is installed. That installation is done locally by Mozilla's native install system. The file could equally be installed from a remote location using the remote install part of the platform.

The topmost directory, named `bin` in this case, collects together all the files into one subtree. There are versions of this XPI for each operating system platform; for example, the UNIX version replaces `.ICO` files with `.XBM` files. On all platforms the `chatzilla.jar` file inside the bundle contains all the chrome files for the Chatzilla application. This JAR file can be seen in the `chrome` area of any installed copy of the Mozilla application suite. The `chatzilla-service.js` file is a new XPCOM component that is delivered with the application. That component registers command-line options

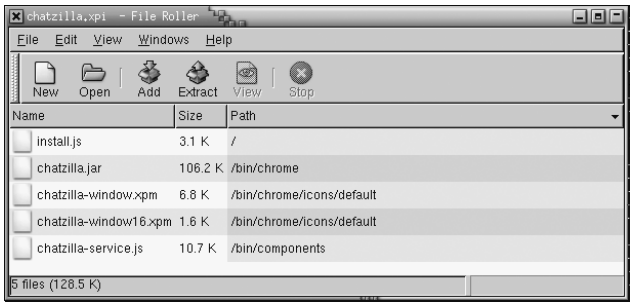


Fig. 17.1 Chatzilla XPI file for Microsoft Windows.

(`-chat`) and a URL scheme (`irc://`) with the platform that integrates Chatzilla with the rest of the platform.

17.2.1.5 Shorthand for No-Content XPI If the XPI file only implements a skin or a locale, then the scripting process can be shortened. No `install.js` script is needed in that case. In the trigger script, call `installChrome()` instead of `install()`. Because installing skins or locales cannot fail (unless disk space is very low), the trigger script is reduced to a single line.

In this case, files inside the XPI file are copied directly into the chrome directory. Those inside files should be JAR files.

17.2.1.6 Shorthand for MIME Types If the XPI file is served up by a Web server so that it has MIME type

```
application/x-xpinstall
```

then the XPInstall system will handle that file automatically. In that case, there is no need for a script placed on an event handler in the application code. Any `install.js` script provided inside the XPI file will still be run.

The application programmer also has the option of avoiding the XPInstall system entirely. The XUL tags described in “Install Technologies” allow a dialog box that acts like an installation wizard to be created easily.

17.2.2 What the User Does

The user chooses whether to deploy the application. The experience he has with the install process is shown in the following sequence of figures. The first thing he sees is a document separate from the to-be-installed application, as shown in Figure 17.2.

This is an HTML page, but it could be a XUL document, in which case the trigger script might be less obvious than a button. In this example, one application consisting of three XPI bundles is specified by the trigger script. Next, the user sees a dialog box as in Figure 17.3.

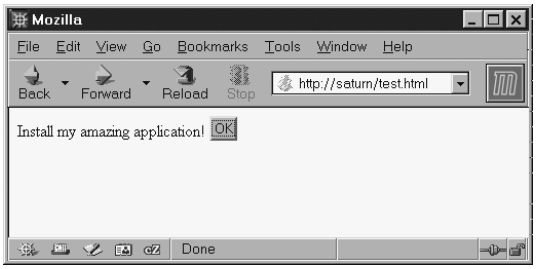


Fig. 17.2 Step 1 of remote deployment: an HTML page.

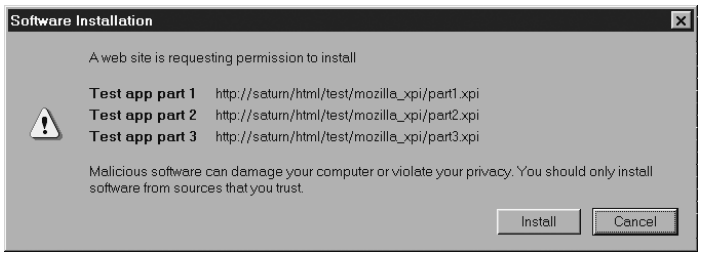


Fig. 17.3 Step 2 of remote deployment: picking list display.

This dialog box reports the goods to be received. If the user chooses Cancel, everything is aborted. If the user agrees, the installation starts immediately and may complete with no further opportunities to back out. After this picking list, a progress dialog box is displayed, as shown in Figure 17.4.

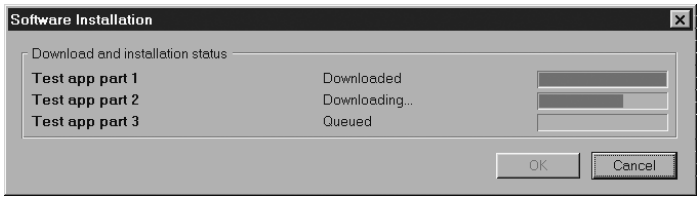


Fig. 17.4 Step 3 of remote deployment: application bundle download.

The listed bundles are downloaded in order. The contained `install.js` files are run as soon as their XPI bundle is available. Unless they contain specific code, these scripts finish without user interaction. If prompts are displayed by the code in the script, they look just like ordinary JavaScript prompts, as Figure 17.5 shows.

After the `install.js` scripts have run, the outcome is either a canceled, a failed, or a completed installation. The user next sees a summary of the processing, as in Figure 17.6.

If the installation process requires that the application be restarted, then a final dialog box will advise the user of this.

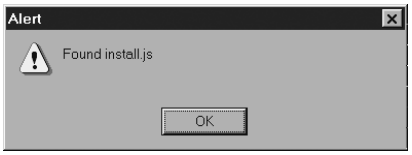


Fig. 17.5 Step 4 of remote deployment: optional user interaction.

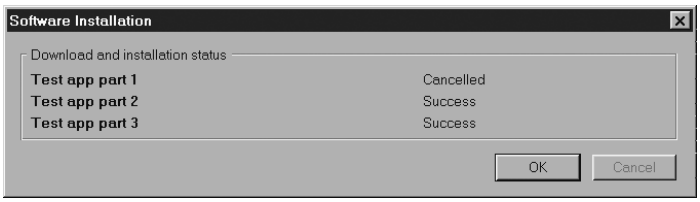


Fig. 17.6 Step 5 of remote deployment: results.

17.2.3 What the Platform Does

During remote deployment, the platform manages the process, receives instructions from the `install.js` script, does all the actions specified, and keeps a record of everything done. The following steps are performed:

1. Start the XPInstall system when `InstallTrigger.install()` or `InstallTrigger.installChrome()` is called. Present and manage the windows that the user sees.
2. Maintain a list of XPI archives to download. Download them to the operating system's temporary files directory. Don't use the Mozilla cache.
3. If the first item in a downloaded archive is a digital signature (detected by path name), then verify the signature. Fail only if verification against the right certificate is possible and fails; in that case, cease the download. Otherwise, continue.
4. Run each `install.js` file one at a time. There is no coordination between different `install.js` files other than their initial order. If one XPI file needs to know if an earlier one finished successfully, then application code must be written that separately tracks progress. That code might create a "touch file" or use Microsoft Windows registry keys as counters to track progress.
5. During the execution of an `install.js` file, record the information in steps 6 to 11.
6. Record any logged messages, plus some automatically produced text, in the file `chrome/install.log`.
7. Record in a Mozilla registry all the things that would need to be undone if the application were uninstalled.

8. Record in a Mozilla registry that this application and its version now exist.
9. Record in `chrome/installed-chrome.txt` the results of all calls to `registerChrome()`.
10. Record if the chrome or components need to be reassessed by the platform.
11. Record if a reboot is required.
12. Next, proceed to the scheduled instructions for the install.
13. Unpack whatever files are required. If file names or path names in the XPI bundle don't match those supplied in the script, do nothing and move to the next instruction.
14. Perform other operating system manipulations as instructed. If the instructions aren't possible or aren't sensible, report an error and do nothing, and then move on to the next instruction.
15. When the user restarts the platform, go through normal post-installation initialization: recalculate chrome overlays, XPCOM components, and available locales and skins.

That is the whole of the remote installation process. Version checking is entirely up to the application programmer and the `install.js` file.

The XPInstall system occasionally must reach out to the rest of the platform to get its job done. An example is the use of alert dialog boxes. If this is necessary, then scripts running elsewhere in the platform will be blocked until XPInstall has finished with the resources it needed. Usually any such blockage is brief and of little consequence.

17.3 INSTALL TECHNOLOGIES

The remainder of this chapter describes the pieces of technology used or usable for remote installation.

17.3.1 File Uses and Formats

The platform's installation uses several different kinds of data files. There are three main uses for these files, and they are written in several different file formats. The three uses are registries, manifests, and logs.

- *Registries* are read/write files that the platform uses as simple databases that can be updated. Configuration, application, and version information is stored in registries.
- *Manifests* are read-only files. They act as bills-of-lading or picking lists; in other words, they describe what's provided. Mozilla uses manifests to list available XPCOM components, plugins, chrome files, overlays, and

some aspects of the build process. Manifests can sometimes be generated from other information.

☞ *Logs* are write-only files that can be independently examined. Mozilla creates logs for the initial platform install and for subsequent remote application installs. Extensive extra logging is possible if the platform used was compiled with `--enable-debug`.

Formats used for these files include the Mozilla registry format (described next), Microsoft Windows `.ini` format, and RDF and plain text. Table 17.1 lists all these formats and their uses.

17.3.2 Mozilla Registries

A Mozilla registry is a file that is similar in format to the Microsoft Windows registry. There is very little direct access to the registry, but a peek inside clears up some mysteries about the platform.

A Mozilla registry consists of a hierarchy of keys, each of which can have a set of attribute-value pairs. The keys can be named using a hierarchical description, which is effectively a path. At the top of the hierarchy is a root key named `/`, and several special names point to important parts of the hierarchy. These are equivalent to Microsoft's `HKEY_` names. Mozilla uses UTF8-encoded Unicode strings for the names in a registry path.

Unlike the Windows registry, the Mozilla registry uses forward slashes (`/`) to delimit steps in a key's fully qualified path. There are other differences as well.

Table 17.1 Installation documents used by the platform

File type	File names using this type	Use
Mozilla registry	mozver.dat, mozregistry.dat, registry, registry.dat, appreg, global.regs, versions.regs, "Mozilla Registry," "Mozilla Versions"	Registry of platform and application names and versions, uninstall information, plugins, and created user profiles
Microsoft Windows .ini	pluginreg.dat, pluginreg, xpti.dat, compreg.dat	Manifest of current plugins, current components, and XPConnect type libraries
	manifest.ini, master.ini, talkback.ini	Configuration files for Talkback-enabled releases
XML RDF	overlays.rdf, contents.rdf, various other per-profile files	Manifests of overlays and of JAR archive content
Line-formatted plain text	installed-chrome.txt, chromelist.txt	Manifests of chrome files that need to be considered for overlays, themes, and locales
Free format text	install.log, install_status.log	Log files for native and remote installation

- Mozilla does not, as yet, have a tool equivalent to `regedit` or `regedit32`.
- Interfaces to registries are not yet fully exposed to application programmers.
- Registries are cross-platform and appear on UNIX, MacOS, and other platforms.
- The platform maintains more than one registry per platform installation.

All Mozilla registries have the same top-level structure. Each registry consists of exactly one root (named /) with four immediate children:

```
"/Users/"
"/Common/"
"/Version Registry/"
"/Private Arenas/"
```

If special compilation options are turned on, then a fifth child exists: "`/Current User/`". This fifth child is not present in the default builds. A key held under one of the four names might have a full path like so:

```
"/Version Registry/mozilla.org/Mozilla/XPCom/bin"
```

This key does not look like an application registry name by accident—that is exactly what the string after "`/Version Registry`" is.

A Mozilla registry is effectively the boot information for the platform. A running platform instance seeks information from the registry after the operating system launches that instance. The most confusing aspect of registries is that there are several, and each one holds a different subset of the information. Table 17.2 attempts to clarify this.

The "chrome registry" is not a file in Mozilla registry format. It is a term that covers the RDF-driven configuration of the chrome, including the overlays database and supporting text files like `installed-chrome.txt`. See Chapter 12, Overlays and Chrome, for more on it.

The Mozilla registry cannot be accessed from an `install.js` script, unless a separate executable is run. It can be accessed from an ordinary application script using this XPCOM pair:

```
@mozilla.org/registry;1 nsIRegistry
```

This interface provides read/write methods and methods that can be used to traverse the tree of registry keys. The interface provides no obvious way to dump out the whole hierarchy from the root. A dirty trick, which has worked in the past, is to use the magic number 32 (hex 0x20) as an `nsRegistryKey` argument. That is the number of the root key. Be aware that neither the root nor its immediate children (the four well-known children) have any attribute-value pairs stored against them. Do not look for such pairs; only look for pairs further down the tree.

Table 17.2 Mozilla registries maintained by the platform

Everyday name	Single user O/S has	Multiuser O/S has	File names used	Keys populated	Use
Versions registry	One	One per O/S user	mozver.dat, "Mozilla Versions", Versions.regs	Versions Private arenas	Global registry of version information for all platform installations, including Netscape Global registry of uninstall information for all platform installations, including Netscape
Global registry	One	One per O/S user	mozregis-try.dat, "Mozilla Registry", registry, Global.regs	None	No current use
Application registry	One	One per O/S user	appreg, registry.dat	Common	Registry of all user profiles and of all available plugins and Java support
Component registry	One per platform installation	One per platform installation	compon-ent.reg , "Component Registry"	Common	Registry of all XPCOM components available

17.3.3 XUL Wizards

Occasionally an application needs to guide the user through a complicated procedure. Deploying software is one such procedure. A traditional way to manage the complexity is to provide a window or dialog box that assists the user step by step. Such a dialog box is sometimes called a *wizard*.

XUL supplies a `<wizard>` tag to assist with complex processes like installation. `<wizard>` is like a fancy combination of a `<deck>` and a `<dialog>` tag. Each `<wizard>` tag holds one or more `<wizardpage>` tags. Each of the `<wizardpage>` tags holds any XUL content.

Like the `<dialog>` tag, the `<wizard>` tag represents a whole window and is used in place of a `<window>` tag. Like cards in a deck, the set of `<wizardpage>` tags are laid on top of each other. The `<wizard>` tag supplies Next, Back, Cancel, and Finish buttons that let the user navigate between the pages, just as the tab labels in a `<tabbox>` provide navigation between tabs. Both `<wizard>` and `<wizardpage>` tags are based on XBL bindings stored in the file `wizard.xml` in `toolkit.jar` in the chrome.

In `toolkit.jar` in the chrome, there are also a number of files prefixed with "`wizard`." These files add value to the basic `<wizard>` tag, particularly in the form of the WizardManager JavaScript object. If your application needs several wizards, it is worth examining this code for its value as a time-saver. It allows you to create a central object that holds all the scripting logic that ties the wizard GUI to your application. It is therefore an orderly way to proceed.

Figure 17.7 shows a window based on the `<wizard>` tag. This window is used in the Mozilla Email client to create a new News or Email account.

Fig. 17.7 Email account creation system based on the `<wizard>` tag.

None of this markup is used or usable from the normal remote install system discussed in this chapter. It can be used only for custom installs, in the chrome, or in remotely hosted applications.

17.3.3.1 <wizard> The `<wizard>` tag provides some content, navigation logic, and event processing for the window it manages. The minimum the application programmer needs to do is supply the remaining content as a set of `<wizardpage>` tags (which usually consists of form elements and explanatory text) and scripts to validate and act on the choices the user makes. The `<wizard>` tag supports the following special attributes:

```
title pagestep firstpage lastpage onwizardnext onwizardback
     onwizardcancel onwizardfinish
```

- `title` specifies the string that will appear in the top part of the window, after the words "Welcome to the."
- `pagestep` specifies how many pages to jump when the Back or Next buttons are pressed. The default is 1 (one).
- `firstpage` is set to `true` by the wizard when the first page is being displayed.
- `lastpage` is set to `true` by the wizard when the last page is being displayed.

The remaining attributes are event handlers and fire when the Next, Back, Cancel, and Finish buttons are clicked. These event handlers all have sensible default actions. Like <dialog> and <window>, the width, height, screenX, and screenY attributes do nothing for the <wizard> tag. <wizard> automatically sets width="500px", height="380px".

In addition to attributes, the XBL definition for <wizard> has methods and properties that allow scripts to mimic the user's actions. Figure 17.8 shows the content that the <wizard> tag provides for free.

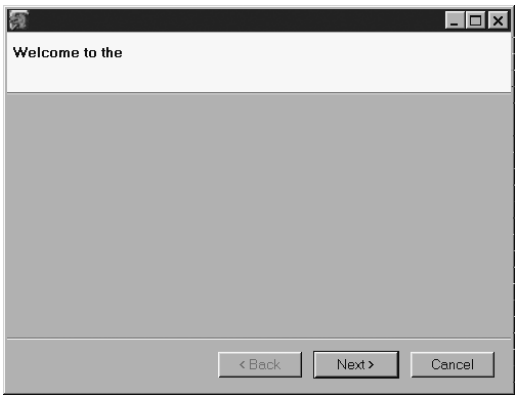


Fig. 17.8 Bare-bones wizard based on the <wizard> tag.

17.3.3.2 <wizardpage> The <wizardpage> tag is a simple boxlike tag. It is operated on extensively by its parent <wizard> tag and is of little use by itself. Put any XUL content inside it, but beware of messing up the simple navigation strategy of the wizard—it is better to add more pages than complicate an existing page. <wizardpage> supports the following special attributes:

```
pageid next onpagehide onpageshow onpagerewound onpageadvanced
```

- pageid is an identifier for the page separate from id. It is used by the logic internal to the <wizard> tag and should always be supplied.
- next provides a way of disrupting the normal page order of the wizard. It holds a pageid identifier. If it is set, the wizard will abandon its simple strategy of stepping forward and backward through the pages. Instead, it will rely on the next attribute for all navigation. If this is to work, the attribute must be set by assigning to the next property of the DOM object for the <wizardpage> tag, not by directly adding it to the <wizardpage> tag. After this is done, all wizard navigation relies on all <wizardpage> tags having a next attribute.

The remaining four attributes are event handlers. They fire when the page appears and disappears, and when the user goes forward a page and

backward a page. They have no default implementations. `<wizardpage>` is trivial at best.

17.3.4 Web-Side Objects

The XPInstall remote install system provides the application programmer with two objects usable in ordinary HTML and XUL documents: `InstallTrigger` and `InstallVersion`. `InstallTrigger` and `InstallVersion` are properties of the global (`window`) object; `InstallVersion` objects can also be made from the `InstallTrigger.getVersion()` method.

17.3.4.1 `InstallTrigger` Table 17.3 describes the `InstallTrigger` object.

17.3.4.2 `InstallVersion` Table 17.4 describes the `InstallVersion` object. Note that extra constants are available but that the meaning is the same.

Table 17.3 The XPInstall `InstallTrigger` Object

Constant, property or method signature	Use
SKIN (1), LOCALE (2), CONTENT (4), PACKAGE (7)	bitmask flags for installChrome()
MAJOR_DIFF(4), MINOR_DIFF(3), REL_DIFF(2), BLD_DIFF(1), EQUAL(0), NOT_FOUND(-5)	Constants returned by compareVersion() indicating where two versions differ; values of opposite sign are also possible, except for 5
Boolean enabled()	True if XPInstall is enabled in preferences
Boolean install(Object xpi_list, function(url, err))	True if a list of XPI bundles installs correctly; the function argument will be called for each XPI URL that fails to install; see “Scripts”
Boolean installChrome(Number flags, String url, String name)	Same as install(), except flags is a bitwise OR that says what the content is, and install.js is not run; name is the application’s text name
Number compareVersion(String name, String version) Number compareVersion(String name, InstallVersion version) Number compareVersion(String name, Number major, Number minor, Number release, Number build)	Compare the version of the application with registry name “name” against the supplied version; returns positive constants if the supplied version is greater
InstallVersion getVersion(String name)	Return the version of the supplied application registry name, or null

Table 17.4 The XPInstall installversion object

Constant, property, or method signature	Use
MAJOR_DIFF(4), MINOR_DIFF(3), REL_DIFF(2), BLD_DIFF(1), EQUAL(0), BLD_DIFF_MINUS(-1), REL_DIFF_MINUS(-2), MINOR_DIFF_MINUS(-3), MAJOR_DIFF_MINUS(-4), NOT_FOUND(-5)	Constants returned by compareTo() indicating where two versions differ
major	Holds the major version
minor	Holds the minor version
release	Holds the release version
build	Holds the build version
void init()	Initializes this object to version "0.0.0.0"
void init(String version)	Initializes this object to the version number supplied
String toString()	Returns a string representation of the version held, or null
Number compareTo(String version) Number compareTo(InstallVersion version) Number compareTo(Number major, Number minor, Number release, Number build)	Compares the held version against the supplied version; returns positive constants if the supplied version is greater

The `InstallVersion` object is also available to the `install.js` scripting environment.

17.3.5 XPInstall-Side Objects

The following objects are available to `install.js` scripts:

```
Install InstallVersion FileSpecObject File WinProfile WinReg
```

The `InstallVersion` object is described in "Web-Side Objects"; the others are described here.

17.3.5.1 `Install` The `Install` object is the global object within the `install.js` scripting environment. That object's methods may be called directly or prefixed with `Install`. `Install` is equivalent to the window property in a Web page.

The `Install` object is a factory object and can produce all the other objects that exist. It holds all the arguments passed in from the install trigger script. It provides access to a global error number similar to `errno` in C/C++

and has a concept of the current directory during the installation. It can do very basic logging, user interaction, and execution of native programs.

Tables 17.5,17.6, and 17.7 describe this object. All the properties of the `Install` object are read-only.

Table 17.5 The `XPInstall` Install object

Constant, property, or method signature	Action deferred until install?	Use
SKIN (1), LOCALE (2), CONTENT (4), PACKAGE (7), DELAYED_CHROME(16)		Bitmask values for flags property and for registerChrome(); use of DELAYED_CHROME delays registration until the platform next starts up
Number buildID		The build version of this installation of the platform (e.g., 2002060411)
Error constants (see Table 17.6)		
String platform		Holds the operating system type and version, which closely follow the style of window.navigator.userAgent
String jarfile		Full path name of the copy of the XPI file on the local computer
String archive		Same as jarfile
String arguments		Holds any string after ? in the XPI file's URL, or null
String url		The full XPI URL passed to install() or installChrome()
Number flags		Flags passed to InstallTrigger's installChrome(), or zero
Number _finalStatus		Value returned back to the remote install progress dialog box
Boolean _installedFiles		False after cancelInstall() is called
File File		Reference to a File object
Object Install		Self-reference to the global Install object

Table 17.5 The `XPInstall` Install object (Continued)

Constant, property, or method signature	Action deferred until install?	Use
Number addDirectory(String XPItree) Number addDirectory(String name, String XPItree, FileSpecObject OSpath, String localPath) Number addDirectory(String name, String version, String XPItree, FileSpecObject OSpath, String localPath) Number addDirectory(String name, InstallVersion version, String XPItree, FileSpecObject OSpath, String localPath)	✓	Copy the supplied XPItree path to the local file system; install under the current application or the one with registry name name, if supplied; always install under the latest version, or, if a version is supplied, use it to check if any existing application is newer; don't install if this check says it is newer; if no destination is supplied, install the contents of XPItree under the current directory; if OSpath and localPath are supplied, concatenate them and store XPItree under the result; return any errors
Number addFile(String XPIfile) Number addFile(String name, String version, String XPIfile, FileSpecObject Ospath, String localPath, [Boolean force]) Number addFile(String name, InstallVersion version, String XPIfile, FileSpecObject OSpath, String localPath, [Boolean force])	✓	Copy the XPIfile in the XPI archive to the local file system; install under the current directory, current application, and current version if no other details are supplied; if an application registry name is supplied in name, use that instead of the current application; if a version is supplied, don't install if an existing application is more recent than the version; if OSpath and localPath are supplied, concatenate them and install the file under the resulting folder; if force is supplied and set to true, don't do version tests—always install in that case; return any errors
Null alert(String value)		Display a modal dialog window showing value until the user acknowledges it
void cancelInstall() void cancelInstall(Number reason)		Don't perform any of the scheduled instructions; if a reason is suppled, set it as the error code, otherwise, set to INSTALL_CANCELLED
Boolean confirm(String value)		Display a modal dialog window showing value until the user accepts or rejects it; Return false if it is rejected

Table 17.5 The `XPInstall` Install object (Continued)

Constant, property, or method signature	Action deferred until install?	Use
Number execute(String XPIpath, String args, Boolean blocking) Number execute(String XPIpath, String args); Number execute(String XPIpath);	✓	Execute the program at path XPIpath in the XPI archive; Optionally supply it with operating system–specific arguments; Optionally supply it with blocking set to true, which pauses the install until the executed program is finished; The default for blocking is false
Number gestalt(String selector)		On the Macintosh only, reports the value of selector according to the Gestalt Manager; Otherwise, return null; Also see text accompanying below this table
FileSpecObject getComponentFolder(String name) FileSpecObject getComponentFolder(String name, String subpath)		Return the folder for the application with registry name name; return the folder of the subpart of that application if subpath is present; otherwise, return null
FileSpecObject getFolder(String keyword) FileSpecObject getFolder(String keyword, String subpath) FileSpecObject getFolder(FileSpecObject folder, String subpath)		Return the folder matching the supplied keyword or a subfolder of that folder if subpart is supplied: if subpart is a JAR file name rather than a folder, then step into the root virtual folder of the JAR file; alternately, specialize an existing folder to one of its subfolders; return null on failure
Number getLastError()		Return the last error status code encountered, which could also be SUCCESS
WinProfile getWinProfile(FileSpecObject folder, String filename)		Return a WinProfile object for the supplied .INI file; returns null if the operating system is not Microsoft Windows
WinReg getWinRegistry()		Return a WinReg object
Number initInstall(String text_name, String reg_name, String version) Number initInstall(String text_name, String reg_name, InstallVersion version);		Begin the scheduling process for this install; set the current application to text name text_name, registry name reg_name, and current version to version; return any errors

Table 17.5 The `XPInstall` Install object (Continued)

Constant, property, or method signature	Action deferred until install?	Use
Object loadResources(String XPIpath)		Return a JavaScript object modeled on a properties (stringbundle) file in the XPI archive; that properties file has XPI relative path name XPIpath; each property in the file appears as a property on the JavaScript object; returns null on failure
Null logComment(String text)	✓	Write the text, plus some formatting, to the install.log file
patch()	✓	This method allows a single file to be updated based on a byte-by-byte delta (a series or diff of changes); not recommended for applications; use addFile() instead
Number performInstall()		Execute all the scheduled install tasks and return an error status
Number registerChrome(Number flags, FileSpecObject folder) Number registerChrome(Number flags, FileSpecObject folder, String rdfpath)	✓	Tell the platform that these chrome files should be reexamined for overlays, locales, and skins; flags says what kind of thing is being registered (see below), folder is the location of the files to consider, rdfpath is an optional subpath (including file name) from folder that says where to find the contents.rdf file for overlays
Number refreshPlugins() Number refreshPlugins(Boolean reloadPages)	✓	Make the platform recalculate the available plugins and then reload all windows depending on them; if reloadPages is false, reload is skipped
void resetError() void resetError(Number error)		Set the last error encountered to zero or to error if it is supplied
Number setPackageFolder(FileSpecObject folder)	✓	Change the current directory to the supplied folder
Number uninstall(String name)	✓	Schedule the given application registry name for uninstall; returns an error code

The following notes expand on aspects of Table 17.5.

It is important to realize that error values returned from scheduling methods only report problems with the scheduling process. They do not report problems with the execution of the scheduled instruction. Even if no error is received during scheduling, the instruction can still fail when it is executed during installation.

Error values are usually negative integers. Values between -200 and -299 are reserved for the platform; values smaller than -5550 are Macintosh specific; 0 is SUCCESS, and 999 is REBOOT_NEEDED. All values produced by the platform have matching property names that hold constants. Table 17.6 lists these names.

A list of valid selectors for the `gestalt()` method and their meanings and values can be viewed at *www.rgaros.nl/gestalt/index.html*.

The special folder keywords submitted as arguments to `getFolder()` are listed in Table 17.7.

The "`Program`" keyword matches the top of the platform installation area. The "`file:///`" keyword matches the top of the local file system. To see the value of a specific keyword on a specific computer, use this line of code:

```
alert(getFolder(keyword).toString());
```

17.3.5.2 `FileSpecObject` The `FileSpecObject` is a value-like object that is passed between the methods of other objects in the XPInstall system. It is rare that this object type is manipulated directly. It is never created with `new` from JavaScript. The `FileSpecObject` has only one useful property:

```
String toString()
```

This method returns a nonportable path for the folder that the `FileSpecObject` represents.

17.3.5.3 `File` The `Install` object is in part responsible for the overall matching and installing of files, folders, and subtrees of files. By comparison, the `File` object is responsible for inspecting and manipulating one file or folder closely. Some of the `Install` object's methods schedule tasks to be done when the install gets going. All of the `File` object's methods schedule tasks for later execution.

Only one `File` object is ever needed, and that object is available as a property named `File` on the global object. All methods on this object are therefore accessible just by calling:

```
File.method_name(args);
```

Table 17.8 describes the `File` object.

Table 17.6 Mozilla platform property names

Name	Name	Name
ACCESS_DENIED	INSUFFICIENT_DISK_SPACE	READ_ONLY
ALREADY_EXISTS	INVALID_ARGUMENTS	REBOOT_NEEDED
APPLE_SINGLE_ERR	IS_DIRECTORY	SCRIPT_ERROR
BAD_PACKAGE_NAME	IS_FILE	SOURCE_DOES_NOT_EXIST
CANT_READ_ARCHIVE	KEY_ACCESS_DENIED	SOURCE_IS_DIRECTORY
CHROME_REGISTRY_ERROR	KEY_DOES_NOT_EXIST	SOURCE_IS_FILE
DOES_NOT_EXIST	MALFORMED_INSTALL	SUCCESS
DOWNLOAD_ERROR	NETWORK_FILE_IS_IN_USE	UNABLE_TO_LOAD_LIBRARY
EXTRACTION_FAILED	NO_INSTALL_SCRIPT	UNABLE_TO_LOCATE_LIB_FUNCTION
FILENAME_ALREADY_USED	NO_SUCH_COMPONENT	UNEXPECTED_ERROR
GESTALT_INVALID_ARGUMENT	PACKAGE_FOLDER_NOT_SET	UNINSTALL_FAILED
GESTALT_UNKNOWN_ERR	PATCH_BAD_CHECKSUM_RESULT	USER_CANCELLED
INSTALL_CANCELLED	PATCH_BAD_CHECKSUM_TARGET	VALUE_DOES_NOT_EXIST
INSTALL_NOT_STARTED	PATCH_BAD_DIFF	

Table 17.7 Keyword identifiers for Install.`getFolder()`

Cross-platform	Microsoft Windows	Macintosh	UNIX
“Plugins”	“Win System”	“Mac System”	“Unix Local”
“Program”	“Windows”	“Mac Desktop”	“Unix Lib”
“Temporary”		“Mac Trash”	
“Profile”		“Mac Startup”	
“Preferences”		“Mac Shutdown”	
“OS Drive”		“Mac Apple Menu”	
“file:///”		“Mac Control Panel”	
“Components”		“Mac Extension”	
“Chrome”		“Mac Fonts”	
		“Mac Preferences”	
		“Mac Documents”	

Table 17.8 The XPInstall File object

Constant, property, or method signature	Action deferred until install starts?	Use
Number copy(FileSpecObject src, FileSpecObject target)	✓	Copies a file or folder to a new destination.
Number dirCreate(FileSpecObject local)	✓	Creates the local directory given by local.
FileSpecObject dirGetParent(FileSpecObject dir)		Returns the parent directory of dir, or null.
Number dirRemove(FileSpecObject local)	✓	Removes the local directory given by local.
Number dirRename(FileSpecObject local)	✓	Removes the local directory given by local.
Number diskSpaceAvailable(FileSpecObject local)		Returns the disk space available on the volume/drive holding the file or folder local. Returns bytes.
Number execute(FileSpecObject file [, String args [, Boolean blocking]])	✓	Runs the executable given by file, with optional arguments args. If blocking is also supplied and set to true, the install will halt until the program finishes. blocking is false by default.

Table 17.8 The XPInstall File object (Continued)

Constant, property, or method signature	Action deferred until install starts?	Use
Boolean exists(FileSpecObject local)		Returns true if the local file or folder named local exists.
Boolean isDirectory(FileSpecObject local)		Returns true if the thing named local is a local folder (file system directory).
Boolean isFile(FileSpecObject local)		Returns true if the thing named local is a local file, and not a folder.
Boolean isWritable(FileSpecObject local)		Returns true if the local folder or file named local is writable.
Number macAlias(FileSpecObject src, String filename, FileSpecObject target) Number macAlias(FileSpecObject src, String filename, FileSpecObject target, String alias)	✓	Creates a Macintosh Alias in the folder target, based on the file file name that resides in the folder src. If alias is supplied as an argument, make that the text of the new alias.
Number modDate(FileSpecObject local)		Returns when local was last changed in milliseconds. This time is calculated differently for each platform.
Boolean modDateChanged(FileSpecObject local, number modDate)		Returns true if the file local has changed since the date modDate (from modDate()).
Number move(FileSpecObject src, FileSpecObject target)	✓	Moves the file src to the folder target. Cannot move Microsoft Windows directories.
String nativeVersion(FileSpecObject local)		Gets Microsoft Windows version information about the file local (e.g., DLL version), or return null.
Number remove(FileSpecObject local)	✓	Removes the file or folder local.
Number rename(FileSpecObject local, String name)	✓	Renames the file or folder local to name.
Number size(FileSpecObject local)		Returns the size in bytes of the file local.
String windowsGetShortName(FileSpecObject local)		For Microsoft Windows only, gets the 8.3 (non-LFN) file name for the supplied file; otherwise, returns null.

Table 17.8 The XPInstall File object (Continued)

Constant, property, or method signature	Action deferred until install starts?	Use
Number windowsRegisterServer(FileSpecObject local)	✓	For Microsoft Windows only, registers the file local as a server.
Number windowsShortcut(FileSpecObject local, FileSpecObject target, String linkname, FileSpecObject dir, String params, FileSpecObject icondb, Number index)	✓	For Microsoft Windows only, creates a shortcut for file local. Puts the shortcut in directory target. Gives the shortcut the name linkname, plus an .lnk extension. Makes the working directory of the shortcut dir. Gives the shortcut parameters params. Uses the indexth icon in the file at path icondb for the shortcut's desktop icon.
String windowsVersion(FileSpecObject local)		Gets Microsoft Windows version information about the file local, or return null.

17.3.5.4 `WinProfile` The `WinProfile` object, manufactured by the `Install.getWinProfile()` method, can perform operations on a Microsoft Windows specific `.INI` file, such as `C:\WINDOWS\WIN.INI`. It contains two methods only:

```
String getString(String section, String key)
String writeString(String section, String key, String value)
```

Since an `.INI` file is in extended ASCII format, Unicode information cannot be put in such a file.

17.3.5.5 `WinReg` The `WinReg` object, manufactured by the `Install.GetWinRegistry()` method, provides access to the Windows registry. Operations on the registry happen immediately; they are not deferred.

The `WinReg` object holds the current registry root key. By default the current root key is `HKEY_CLASSES_ROOT`.

Path names for Windows registry keys are delimited by backslashes (\). Backslashes in JavaScript strings must be stated doubly (\\) if they are to be treated as normal characters.

Table 17.9 describes the `WinReg` object. Some of the methods listed return data, but many of the methods merely return a status code. When a status code is returned, `null` means that the Mozilla Platform couldn't assemble the registry change correctly. This usually means problems were encountered with the arguments supplied. If a non-null-value is returned, that value comes from the

actual registry operation. Even when the return value is ordinary data, a `null` value means failure of the method, again most likely the result of argument problems. In summary, always check return values for `null`.

Table 17.9 The XPInstall WinReg object

Constant, property, or method signature	Use
HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS	Predefined constants for well-known root keys.
Number createKey(String subkey, String name)	Creates the subkey subkey with the class name name. name may be a zero-length string.
Number deleteKey(String subkey)	Deletes the subkey named subkey.
Number deleteValue(String subkey, String name)	Deletes the attribute-value pair of subkey whose name is name.
String enumKeys(String subkey, Number index)	Returns the indexth subkey for the key named subkey. Returns a zero-length string for non-existent subkeys.
String enumValueNames(String subkey, Number index)	Returns the indexth attribute name for the key named subkey.
Number getValueNumber(String subkey, String attr)	Returns the DWORD value of the attr attribute of the subkey key.
String getValueString(String subkey, String attr)	Returns the String value of the attr attribute of the subkey key.
Number getValue(String subkey, String attr)	Returns the DWORD value of the attr attribute of the subkey key.
Boolean isKeyWritable(String subkey)	Returns true if the subkey key is writable.
Boolean keyExists(String subkey)	Returns true if the subkey key exists.
void setRootKey(String key)	Sets the root key to one of the predefined roots listed in the first row of this table. Returns null on failure.
Number setValueNumber(String subkey, String attr, Number val)	Sets the attr attribute of the subkey key to the val value. Returns null on failure.
Number setValueString(String subkey, String attr, String str)	Sets the attr attribute of the subkey key to the str value. Returns null on failure.
Boolean valueExists(String subkey, String attr)	Returns true if the subkey key has an attr attribute.

17.3.6 XPCOM Objects

The XPInstall system does not make XPCOM interfaces available to `install.js` scripts.

There are, however, a number of components and interfaces available outside of the XPInstall environment that duplicate the XPInstall functionality. If a custom deployment system is required, or deployment-like design is needed, then these components and objects are perhaps worth exploring.

This XPCOM pair provides access to the Mozilla registry, although it is not complete or completely flexible:

```
@mozilla.org/registry;1 nsIRegistry
```

This interface treats the different registry files as though they were all one file. The registry is also expressed as an RDF data source. At the time of this writing, that RDF data source is not available for use. Its XPCOM pair is forecast to be

```
@mozilla.org/registry-viewer;1 nsIRDFDataSource
```

A different registry interface is provided by the so-called chrome registry. This interface allows the platform to recalculate its view of the chrome. That includes refreshing overlay, skin, and locale information. It can also install chrome packages, skins, and locales. The XPCOM pair providing this functionality is

```
@mozilla.org/chrome/chrome-registry;1 nsIXULChromeRegistry;1
```

Of particular interest to Microsoft Windows developers are two pairs of XPCOM items:

```
@mozilla.org/winhooks;1 nsIWindowsRegistry
@mozilla.org/winhooks;1 nsIWindowsHooks
```

These interfaces provide much of the low-level access to Microsoft Windows that the platform-specific operations need.

Separate from these registration-like interfaces are the many file and network manipulation interfaces that are covered in Chapter 16, XPCOM Objects, including interfaces that can unpack a ZIP archive.

All that remains is some observations about XPInstall's own objects. The Web-side objects `InstallTrigger` and `InstallVersion` have no useful XPIDL interface definitions. They do happen to have Contract IDs registered with XPCOM, along with the XPInstall infrastructure itself:

```
@mozilla.org/xpinstall/installtrigger;1
@mozilla.org/xpinstall/installversion;1
@mozilla.rg/xpinstall;1
```

There is no point in using these Contract IDs from a script since no useful interfaces are available. The objects available in the `install.js` scripting environment are not exposed to XPCOM either. In fact, in that second case, not even Contract IDs exist for those objects.

17.4 HANDS ON: BUNDLING UP NOTETAKER

This "Hands On" session bundles up the already working NoteTaker tool into an XPI file that can be installed using XPInstall. In theory, the tool could run directly from a remote Web server, but merging local content with server-based overlays is a messy approach that defies common sense. In practical terms, it may not even work. We'll stick with a downloadable installation.

Although the running code for the tool is complete, several small bits and pieces must be added. Our strategy for bringing them all together is the sum of the issues brought to light in this chapter. That means

- Determine any names for the tool.
- Determine the content of release documents.
- Determine the content of the final tool.
- Create download pages, installation scripts, and support files.
- Create a final XPI file.

Let's run through these tasks.

17.4.1 Release Preparation

First we pick suitable names for our tool:

- **Text name**. We choose "`NoteTaker Web Notes`". We save the advertising and promotional hype for the Web page that will offer the tool.
- **Package name**. Throughout this book we've been using "`notetaker`", which we'll stick to. That's more than the eight characters that very old Microsoft Windows computers support, but perhaps that's a small loss only.
- **Registry application name**. That's "`/Nigel McFarlane/NoteTaker`". If this tool were absorbed into the main Mozilla Browser development stream, then the name might be merely "`NoteTaker`", which would be appended to something like "`/mozilla.org/Browser/`". In that case, the full name would be "`/mozilla.org/Browser/NoteTaker`". There's no such affiliation at this time.
- **Version number**. There's a shortage of real-world testing for the tool so far, but it appears to work. Call it version 0.9. That version is 0.9.0.0 when stated in full. Many enhancements and modifications are possible, but they would bump the version over 1.0, probably.

We also review the software to be delivered.

- **Baseline**. Whatever is discussed in this book is the basis of this release of the tool. I keep a copy of all code relevant to each chapter in a directory

under that chapter and do incremental backups daily and full backups weekly and monthly. Since I'm finished on the 30th, tonight's monthly backup will freeze everything. My baseline will be source files for this book (edition 1, author's final draft), plus the final installable XPI file plus a backup date. That backup will also contain any test files and test data, which is handy.

- **Footprint**. The footprint for the NoteTaker tool is very small. It has only three items: the chrome directory hierarchy; the Mozilla registries; and the `notetaker.rdf` file in the current user profile.
- **Target**. The target for the software has several parts. Because of recent changes to XPCOM objects (file-based streams), this release requires platform version 1.4 final, minimum. It is tested on the Classic Browser only, not on the Mozilla Browser. The application is portable, so the operating system doesn't matter that much; however, there are bound to be a few constraints we haven't identified yet. The platform version and Classic Mozilla application suite will be the whole target, and we note that future versions are not automatically supported.

That's all the logistics required.

17.4.2 Creating Support Files and Scripts

The NoteTaker 0.9 release requires files above and beyond the application code.

We'll include a `README.txt` file for developers exploring the source code. We'll make one up based on the information in the last topic.

We'll need a `contents.rdf` file for the `notetaker/contents` directory and to register the NoteTaker package. We'll use the one specified in Chapter 2, XUL Layout, and include the enhancements made in Chapter 12, Overlays and Chrome.

We want to show a trivial example of locale support. For that we'll need a `contents.rdf` file and a DTD file for a sample locale. We'll use the `contents.rdf` file from Chapter 3, Static Content, and make up a trivial DTD file—one that has no effect.

We also want to show a trivial example of skin installation. For that we'll need a `contents.rdf` file and a CSS file for a sample skin (theme). We'll use the `contents.rdf` file from Chapter 4, First Widgets and Themes, and make up a trivial style sheet—one that has no effect.

Finally we'll need the installation support. That amounts to an HTML file and two scripts. The NoteTaker tool is small and almost entirely reliant on the platform and the Classic Browser application. We expect the install scripts to be lightweight rather then complex.

Because the HTML file might be displayed in any Web browser, it had better be highly portable as shown in Listing 17.4.

Listing 17.4 Download Web page for the NoteTaker tool.

```
<html>
  <head>
    <script src="deploy.js"/>
  <body>
    <h1>NoteTaker Download</h1>
    <p>The NoteTaker tool adds Web Notes to your Mozilla-based Web
          browser. Web Notes are placed on top of displayed Web pages.
          They hold information that you record for your own purposes.
    </p>
    <p>Only the Classic Browser, version 1.4, is supported. It is part of
          the established Mozilla Web application suite. The standalone
          Mozilla Browser is not yet supported.
    </p>
    <p>Download here:
      <a href="notetaker.xpi" onclick="download(event)">NoteTaker tool
          0.9</a>
    </p>
  </body>
</html>
```

The `deploy()` function follows the outline of Listing 17.1. In this case, the full script is shown in Listing 17.5. Bullet-proof browser detection is a lengthy matter; this code covers most common alternatives only.

Listing 17.5 XPInstall trigger script for the NoteTaker bundle.

```
function download(e) {
  if ( ! deploy() )
    alert("NoteTaker 0.9 requires Classic Mozilla 1.4");
  e.preventDefault();
}

function is_moz_browser() {
  return (
    window.navigator &&
    window.navigator.userAgent &&
    window.navigator.userAgent.search(/^Mozilla\/5\.0/) != -1
  );
}

function is_target() {
  var agent = window.navigator.userAgent;
  return (
    agent.search(/rv:1\.4/) != -1 &&  // matches
    agent.search(/Phoenix/) == -1 &&  // no match
    agent.search(/Firebird/) == -1    // no match
  );
}

function is_app_version_ok() {
  var it = window.InstallTrigger;
```

```
  var result = it.compareVersion(
    "Nigel McFarlane/NoteTaker", "0.9.0.0");

  return ( result == it.NOT_FOUND ||
           Math.abs(result) <= it.REL_DIFF );
}

function deploy()
{
  if ( !is_moz_browser() ) { return false; }
  if ( !window.InstallTrigger.enabled() { return false; }
  if ( !is_target() ) { return false; }
  if ( !is_app_version_ok() } { return false; }

  var xpi_container =
    { "NoteTaker Web Notes" : "notetaker.xpi" };

  with (window.InstallTrigger)
    install(xpi_container, null);
  return true;
}
```

There are no parameters required, so that part of the skeleton in Listing 17.1 is gone. If the user attempts to install to the wrong platform or the wrong application, we complain. There is no error handler for the `install()` function because there is nothing we could do to rectify a failure. In a large organization, we might display an HTML page that allows a problem report to be filed. We'll rely instead on the `install.js` file issuing a useful user-oriented complaint.

The three detection functions are simple. Any browser whose `userAgent` starts with "`Mozilla/5.0`" is likely to be a mozilla.org browser. If the `userAgent` contains "`Firebird`" or "`Phoenix`", then it's the Mozilla Browser, not the Classic Browser, which we don't support. The `is_app_version_ok()` test is lenient; it allows, for example, the installation of version `0.9.1.0` on top of `0.9.0.0`. This is a guarantee from the developer that going backward in a release version is minor enough to be safe. That might be required if a new release proves more defective than anticipated.

The final piece of the install system is the script `install.js`. Listing 17.6 is most of that script, based on Listing 17.2.

Listing 17.6 `install.js` script for the NoteTaker tool.

```
var TEXT_NAME = "NoteTaker Web Notes";
var REG_NAME  = "/Nigel McFarlane/NoteTaker";
var VERSION   = "0.9.0.0";
var rv = SUCCESS;

function prepare()
{
  initInstall(TEXT_NAME, REG_NAME, VERSION);
```

```
  if ( schedule_files() != SUCCESS ) return getLastError();
  if ( register_chrome() != SUCCESS) return getLastError();

  return SUCCESS;
}

rv = prepare();
if (rv == SUCCESS) {
  performInstall();
}
else {
  alert("Installation failed. (Error = " + rv + ")" );
  cancelInstall(rv);
}
```

As for the install trigger script, the `install.js` script is simplified from the skeleton of Listing 17.2. There are no parameters to check. We assume that the install trigger script proceeds with the download only if platform conditions are right. Because the NoteTaker tool is a browser enhancement, we don't have desktop menu items or icons or shortcuts to add. NoteTaker is such a small tool that checking available disk space is pointless. Because the tool is written entirely in JavaScript, we don't have any executables or binary libraries to manage either. All we need to do is place the contents of the `.XPI` file correctly and to advise the platform that new chrome content exists.

To do those few steps, we need to know the contents of the XPI. Looking ahead to "Final Bundling," we see that the XPI content is

```
install.js
notetaker.jar
extras/README.txt
extras/notetaker.rdf
```

The whole application resides in the `notetaker.jar` archive. Files in the extras virtual directory won't ever be used at run time. The `README.txt` file is there for curious programmers to find and read; the `notetaker.rdf` file is the initial copy of the user's note database. It needs to be copied into the current user profile.

Listing 17.7 shows the two missing functions from Listing 17.6. They perform the required manipulation of the XPI file and of the chrome registry.

Listing 17.7 Deploying files and registering chrome from `install.js`.

```
function schedule_files()
{
  addFile(TEXT_NAME, VERSION, "notetaker.jar",
          getFolder("Chrome"), "notetaker.jar", true);

  addFile(TEXT_NAME, VERSION, "extras/notetaker.rdf",
          getFolder("Profile"), "notetaker.rdf", true);
```

```
  return SUCCESS;
}

function register_chrome()
{
  var jar_root = getFolder("Chrome", "notetaker.jar");

  registerChrome(PACKAGE | DELAYED_CHROME,
                 jar_root, "content/notetaker/");
  registerChrome(SKIN | DELAYED_CHROME,
                 jar_root, "skin/modern/notetaker/");
  registerChrome(LOCALE | DELAYED_CHROME,
                 jar_root, "locale/en-US/notetaker/");
  return SUCCESS;
}
```

The second `addFile()` function call shows how any file path in an XPI file can be matched to any path on the local file system. The `getFolder()` call in the function `register_chrome()` shows how the top of the folder hierarchy inside the JAR file can be returned as an object. Both `schedule_files()` and `register_chrome()` complete without touching the `notetaker.xpi` file. Their `addFile()` and `registerChrome()` calls are scheduled for later execution when `performInstall()` is called.

That is the whole of the `install.js` script.

17.4.3 Final Bundling

Having located or created all the required files, the `notetaker` XPI file can finally be assembled. It must contain at least the `install.js` file (because it is more than a skin or a locale), so a start is to create a ZIP archive containing that one file.

The NoteTaker tool should be the main contents of the XPI file. It is a neat and efficient arrangement to have the NoteTaker package in a single JAR archive in the chrome. Such a file is faster to read from disk because it is small. It is also easy to manage if the number of installed packages is high. We'll do that, and we'll put that JAR archive inside the XPI install archive for delivery. Unfortunately, JAR archives are very fiddly to set up for two reasons.

In all chapters to date, the NoteTaker tool has resided under the folder `resource:/chrome/notetaker` as a set of discrete files and subfolders. If we delivered the package arranged this way, we could create the required XPI file simply by zipping up a working NoteTaker folder, with an `install.js` file added. A JAR archive, however, has its directory structure arranged differently to support fast access. Our installed NoteTaker packages look nothing like the hierarchy that a JAR file requires:

```
Chrome test package          JAR archive
notetaker/content      <---> content/notetaker
```

```
notetaker/locale/en-US <---> locale/en-US/notetaker
notetaker/skin/modern  <---> skin/moder/notetaker
```

The only solution to this set of differences is to make a copy of the test files from the chrome and rearrange them in a temporary folder hierarchy that reflects the JAR convention. A systematic solution is to write a Perl, WSH, or shell script that automates the rearrangement process. That second hierarchy can then be zipped up; the JAR file results.

A second wrinkle with JAR files is that the order of the files in the archive matters if the archive is large. The most time-critical files should be near the start of the archive, where less effort is required to find them. To achieve this, put the package files in a hierarchy matching the JAR convention as before. Create a text file that lists the files in the order required, and pass that list to a suitable command-line zip tool like `pkzip` (Microsoft Windows) or `zip(1)` (UNIX). It's possible to create the JAR file correctly by adding files and folders to it a piece at a time from the desktop, but that is quite a tedious process.

Figure 17.9 shows a JAR file for the NoteTaker package that has some trivial ordering. The content part of the package is put first because it is used the most. In fact, the skin and locale files in this package are just placeholder files that illustrate where such things might be put. For such a small application, this ordering is probably of no benefit.

We now have two files for the final XPI: `install.js` and `notetaker.jar`. To these we can add a README file and the beginning `notetaker.rdf` file. The final XPI file (in which ordering is not important unless digital signatures are present) appears in Figure 17.10.

Nothing more is required except to upload the HTML page with the download link and the XPI file to a Web site. That concludes the NoteTaker running example in this book.

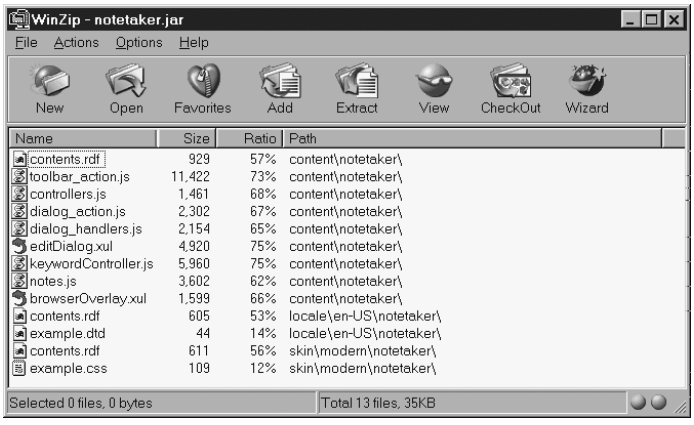


Fig. 17.9 JAR archive holding the NoteTaker package.

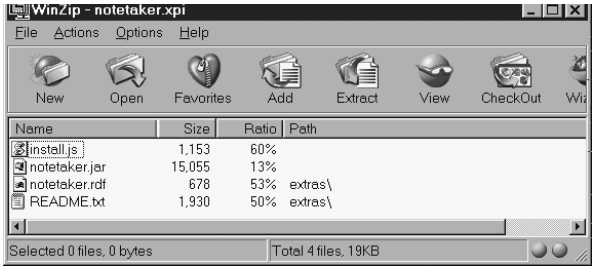


Fig. 17.10 XPI archive holding the full NoteTaker distribution.

17.5 DEBUG CORNER: LOGGING AND TESTING

The `logComment()` method of the `Install` object is the only way to produce diagnostic messages from the `install.js` file, unless `alert()` is considered. Messages are logged to the `install.log` file, which is stored in the top directory of the platform install area.

A common source of problems in the install process is poor matching between hierarchical path names in the XPI file and hierarchical path names on the local file system. If the two are not stated correctly, or do not match correctly, then zero files might be copied by that step of the installation process. This can easily be detected by examining the `install.log` installation log and by checking the file system.

If nothing (or the wrong thing) is happening, a good debugging strategy is to simplify the installation process. Remove any digital signatures, and then remove any code that performs version checks and balances. Start with an XPI file whose contents need only to be copied into the chrome. Concentrate on ensuring that the archive's subtrees of files are installing to their correct locations. If there is no fancy version checking done, then the same XPI file can be installed over and over again harmlessly—there is no need to uninstall between installs. In simple cases, it is also safe to delete custom portions of the chrome by hand. Don't ever delete any of the standard JAR files that come with the platform.

If overlays are included in the application, beware of corrupting the overlay database. If any of the overlays have syntax error or incompatibility problems, the platform or an initial window may fail to start, and the installation will be useless. To fix this, delete the `overlayinfo` database, the installed files, the `chrome.rdf` file, and the line items added to `installed-chrome.txt`. Then restart.

In general, it is recommended that any install testing be done on a separate installation of the platform to that used for everyday purposes. That separate installation can be a "crash and burn" area where experiments can be freely undertaken. On Microsoft Windows, a separate computer is even better because there is only one central Mozilla and Window registry per host. It is

also possible to corrupt the Windows registry from an XPInstall script if you try hard enough.

17.6 SUMMARY

The XPInstall subsystem of the Mozilla Platform has many faces and can be exploited in many ways. Application programmers who don't want to learn how to build and package the platform itself use only a few of these faces.

The most attractive of the XPInstall technologies is remote install. Remote install holds the promise of delivering to users and customers service-provision software that has a low cost of distribution and the possibility of ongoing updates.

The mostly portable XPInstall system highly complements the other portable aspects of Mozilla applications. Rather than dividing the world into Visual Basic, AppleScript, and Perl, Mozilla-based applications can be uniformly deployed and used across most popular operating system platforms.

Application programmers can veer toward custom installs or native installs once their applications gain some maturity and a user base. The XPCOM system provides enough features that a separate installation system can be written on top of the platform if required. The only part of the platform that is difficult to reproduce without C/C++ code is the native installer stubs that first boot the platform into being.

With XPInstall covered, this book's overview of Mozilla technologies is complete. Today the Mozilla Platform is a feature-rich and practical development environment. It is a highly visible and substantial Open Source project, and its future is certainly bright. Good luck with your Mozilla work.