sitepoint®

BUILD YOUR OWN FIREFOX EXTENSION

BY JAMES EDWARDS

Summary of Contents

sitepoint®

BUILD YOUR OWN FIREFOX EXTENSION

BY **JAMES EDWARDS**

Build Your Own Firefox Extension

by James Edwards

Managing Editor: Chris Wyness
Technical Editor: Andrew Tetlaw
Editor: Kelly Steele
Cover Design: Alex Walker
Latest Update: July 2009

Notice of Rights

Notice of Liability

Trademark Notice

sitepoint®

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066.
Web: www.sitepoint.com
Email: business@sitepoint.com
.

About the Author

James (aka brothercake) is a front-end developer based in the UK, specializing in advanced JavaScript programming and accessible web site development. He's an outspoken advocate of standards-based development, and contributing author to SitePoint's *The Art & Science of JavaScript* (http://www.sitepoint.com/books/jsdesign1/). His web site can be found at http://www.brothercake.com/.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums.

Table of Contents

Preface

One of the main reasons for the explosive and continued popularity of Firefox is its framework for custom extensions. What many web developers don't realise is that they're so easy to create and implement — anyone with a minimal understanding of JavaScript and XML can build a Firefox add-on. After reading this book, you'll be able create simple Firefox extensions with extremely powerful functionality.

Who Should Read This Book

If you want to build custom extensions for Firefox, then this is the book for you! With a little JavaScript know-how, author James Edwards will show you just how straightforward it is to build your own Firefox add-ons.

Where to Find Help

The definitive reference source for making Firefox extensions, XUL, and related technologies can be found at the MDC—the Mozilla Developer Center[1]. There are also two google groups available for help:

- For topics relating to extension development, the `mozilla.dev.extensions` group: http://groups.google.com/group/mozilla.dev.extensions/topics.
- For topics relating to XUL, the `mozilla.dev.tech.xul` group: http://groups.google.com/group/mozilla.dev.tech.xul/topics.

The SitePoint Forums

The SitePoint Forums[2] are discussion forums where you can ask questions about anything related to web development. You may, of course, answer questions, too. That's how a discussion forum site works—some people ask, some people answer and most people do a bit of both. Sharing your knowledge benefits others and strengthens the community. A lot of fun and experienced web designers and developers hang out there. It's a good way to learn new stuff, have questions answered in a hurry, and just have fun.

[1] https://developer.mozilla.org/en/Extensions/
[2] http://www.sitepoint.com/forums/

There's even a dedicated JavaScript forum:
http://www.sitepoint.com/forums/forumdisplay.php?f=15.

The Book's Web Site

Located at http://www.sitepoint.com/books/byofirefoxpdf1/, the web site that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains each and every line of example source code that's printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the archive.[3]

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page on the book's web site will always have the latest information about known typographical and code errors.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters, such as *SitePoint Tech Times*, *SitePoint Tribune*, and *SitePoint Design View*, to name a few. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of web development. Sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/.

The SitePoint Podcast

Join the SitePoint Podcast team for news, interviews, opinion, and fresh thinking for web developers and designers. They discuss the latest web industry topics, present guest speakers, and interview some of the best minds in the industry. You can catch up on the latest and previous podcasts at
http://www.sitepoint.com/podcast/ or subscribe via iTunes.

[3] http://www.sitepoint.com/books/byofirefoxpdf1/code.php

Your Feedback

If you're unable to find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is `books@sitepoint.com`. We have a well-staffed email support system set up to track your inquiries, and if our support team members are unable to answer your question, they'll send it straight to us. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Conventions Used in This Book

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css ***(excerpt)***

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Also, where existing code is required for context, rather than repeat all the code, a vertical ellipsis will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➥ets-come-of-age/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me …

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always …

… pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Chapter 1

Making a Start

One of the most sophisticated features in Firefox, and arguably the most significant reason for its continued popularity among developers, is its framework for custom extensions. This goes beyond that they're possible at all, but rather that they're so easy and straightforward to build. Anyone with knowledge of JavaScript and XML can build a Firefox extension, and given the extensive range of components that are provided by the environment, it takes very little to create extremely powerful functionality.

The core technology that extensions are built with is an XML language called **XUL** (eXtensible User-interface Language—sometimes pronounced *zool*), which defines interface components from simple tags like `<label>` and `<menuitem>`, through to more complex structures like `<listbox>`, `<tree>`, and `<toolbar>`. XUL also features tags for elements that are purely visual, such as `<spacer>`, `<splitter>`, and the primary layout-box elements, `<hbox>` and `<vbox>`.

So in a sense, working with XUL is more akin to visual markup like **SVG** (Scalable Vector Graphics), than content markup like XHTML. If you've never worked with a visual markup language before, you might find it a bit of a culture shock, or that it somehow feels *wrong*. Be assured, you'll soon start to enjoy it!

We'll see many examples of XUL elements as we go through this book. The definitive reference source for XUL and related technologies can be found at the MDC—the Mozilla Developer Center;[1] for example, a complete list of XUL elements and attributes can be found at https://developer.mozilla.org/en/XUL_reference/. The documentation is generally of a high quality, even if it's rather patchy in places and difficult to follow in others.

The details in this book apply to Firefox 1.5 or later. Building for earlier versions of Firefox or for the Mozilla application suite is different in detail and approach, and is beyond the scope of this book. However the techniques here are applicable to recent versions of Flock,[2] with only minor changes to the install file, as we'll see later.

Firefox Configuration

To build a Firefox extension we'll be creating and working with many different file types, some of which your computer may be unable to recognize, so to do this effectively you'll need to make sure that your working environment is set up to handle it:

- First, make sure file extensions are visible. If you're using Windows, in Windows Explorer go to **Tools** > **Folder Options** > **View** and uncheck the option **Hide extensions for known file types**. For Mac users, the equivalent option is called **Show all file extensions** and is in the Finder's **Advanced Preferences**. Doing this will ensure that you can see all file extensions, and therefore avoid confusion when working with a mixture of known and unknown files.
- Also, for Windows users, and in the same dialog, check the option **Show hidden files and folders**. Later on we're going to need to access Firefox's Application Data folder, which ordinarily is hidden. (For Mac users the folder we want will already be visible, so no preference change is necessary there.)

For Firefox itself, there are also a couple of tasks you should do to create a suitable development platform:

[1] https://developer.mozilla.org/en/XUL/
[2] http://www.flock.com/

- Go to the URL about:config to view the Firefox configuration page and enable two options: `javascript.options.strict` and `javascript.options.showInConsole`. This will give us the most comprehensive reporting of JavaScript errors and warnings, which we'll need to build our extension properly. Once you do this, you'll start seeing errors and warnings appear that have nothing to do with what you are working on. They can be reported from other extensions, or from Firefox's core. Every error report shows the location, so look at that to see if it's coming from a file you're working on, otherwise ignore it.

- Grab a hold of the Quick Restart extension.[3] For 90% of the development we'll be doing it's necessary to restart the browser between changes, and having a nice big button on the toolbar—like the one in Figure 1.1—to do this will save you massive time and hassles. Believe me, you'll be very glad you did!

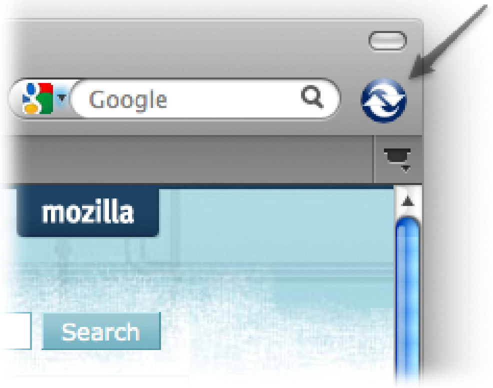


Figure 1.1. The Quick Restart button

There's also another option; you can disable the XUL cache while you're developing your extension—see Chapter 2 for details.

Now that we have our environment sorted, we can create the basic extension template. But before we do that, let me digress for just a moment to give you some background information. This is necessary to understand the inner workings of an extension, and of Firefox itself.

[3] https://addons.mozilla.org/en-US/firefox/addon/3559

Understanding Chrome

Firefox uses a pseudo-protocol called `chrome://` to load XUL files; a chrome address points internally to Firefox's application files. To see what I mean, type or paste the address `chrome://browser/content/browser.xul` into Firefox's address bar and take a look at what happens. Seeing double? What you're viewing is the file that defines Firefox's main interface, and you can poke around it using Firebug[4] or the DOM Inspector.[5] You may recognize some of the more obvious tags, like the `<toolbarbutton>` tags for back, forward, and home buttons, and the `<popup>` tags that define the main document context menu.

If you want to see it in your text editor, go to **C:\Program Files\Mozilla Firefox\chrome\** on a Windows machine, or **/Applications/Firefox/Contents/MacOS/chrome/** in Mac OS X (you'll need to right-click the application file and select **Show Package Contents** from the context menu to look inside the **Firefox** directory). In there you'll find a file called **browser.jar**; copy that and rename the copy to **browser.zip**, then expand the ZIP file and you'll have a directory called **content**. Look inside that, then within the **browser** directory, and there you'll find **browser.xul**.

When we make a Firefox extension, what we're doing is inserting new XUL components into that XUL file, using what's called an **overlay**. An overlay defines one or more insertion points into which new XUL elements can be inserted, usually (though not always) into **browser.xul**. We'll learn more about overlays shortly.

And that's as much as you need to know about chrome files and addresses for now. Let's begin to create our extension. I'm going to talk you through the steps necessary to create a new template, but you can also download the code archive for this book,[6] that has one ready made that has everything necessary to begin working. Unzip the **myextension_template1.zip** file and move the entire **myextension** folder it contains to a convenient location for you; then we'll see in the section called "Step 4: Point Firefox at Your Development Folder" how to point Firefox at it.

[4] http://getfirebug.com/

[5] https://developer.mozilla.org/En/DOM_Inspector/

[6] http://www.sitepoint.com/books/byofirefoxpdf1/code.php

Step 1: Create a Development Folder

The structure of a Firefox extension is a set of folders and files with a specific hierarchy, and in many cases, specific names. In Figure 1.2 below, you can see blue folders, green files (all of which are plain text), and one red file (which is an image). The names in black are all fixed—you must use that exact name—while the names in gray can be anything you choose (with the correct file extensions as a given). You'll notice that the root folder name (**myextension**) and one of the JavaScript files have a star next to their name. This means that you can choose any name for these, as long as they share the exact same name; that is, the JavaScript filename must match the root folder name.

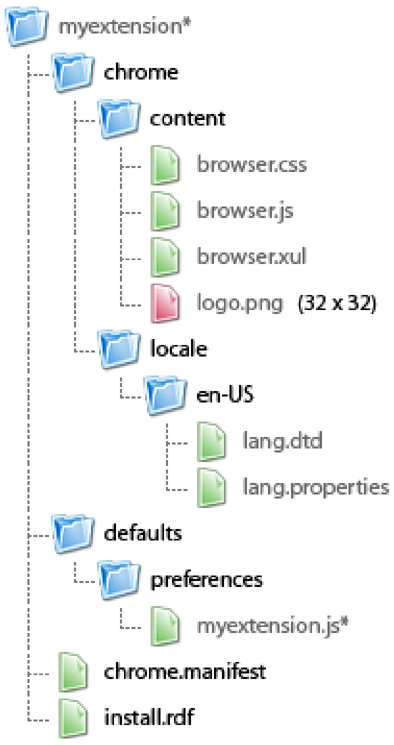


Figure 1.2. The directory structure of a Firefox Extension

The root folder name is the name of your extension, and this will be used in lots of different places—so decide on a name *now*, and stick to it! (Or you could just leave it as “myextension” for now, while you’re setting everything up to work—it’s one less task to think about, and one less item to go wrong!)

There’s no need to worry about what all those files and folders actually do for now; we’ll arrive at each one in turn and I’ll explain it as we use it. For now, you just need to create that hierarchy with empty text files and some kind of image to use as a logo for the **Add-ons** dialog. You can easily change it later, so anything will do for now. The image *must* be 32x32 pixels, and although there’s no need to make it a PNG, it may as well be, to take advantage of the lovely alpha-channel transparency PNG provides.

There are a few other files and folders that an extension may optionally use, but which are excluded here. Principally among them is a **skin** directory, which is unnecessary really, unless you plan to support more than one skin (over and above the default, classic skin that Firefox ships with), or indeed, to create a new skin. Firefox also supports a mechanism for adding platform-specific style sheets, as well as style sheets for different skins, but all of that is beyond the scope of this book.

What we’ve included here is a single style sheet, just for any bits of styling we may need to do to our interface; since that’s all it is, it can quite happily live in the main content directory along with everything else. There may be no need for it at all, since all XUL interface components have default styling, defined by the default skin.

If you’re developing on a Mac then every file and folder name is case-sensitive, while on a Windows machine they’re case-insensitive. However, since JavaScript is case-sensitive, and that’s the primary language we’ll be programming in, it’s best to assume that everything else is case-sensitive too, thereby avoiding any cross-platform issues. Furthermore, Firefox 2 will be unable to find the chrome if the extension name is mixed case, resulting in a “no chrome registered” error. Therefore, you should always choose a name that’s purely lowercase, which is why our extension is called “`myextension`” rather than “`MyExtension`”.

You may feel that it’s impractical supporting Firefox 2 at all, given that it’s such an old version, and no longer officially supported. But in my view it’s a question of accessibility, as some users may not be able to run more recent versions. My sister-in-law, for example, is still running Mac OS X 10.3, and the most recent version she can run is Firefox 2.

Step 2: Define Key Meta Files

Now we need to add content to the two key meta files that the extension uses to identify its components and installation settings; these are **chrome.manifest** and **install.rdf**.

Let's begin with **chrome.manifest**, and this is a *tab-delimited* text file that tells the extension where to find its overlay, content directory, and language directories. Our **chrome.manifest** file should look like this:

```
overlay  chrome://browser/content/browser.xul
➥  chrome://myextension/content/browser.xul
content  myextension  chrome/content/
locale   myextension  en-US  chrome/locale/en-US/
```

As I explained before, an overlay is the primary method by which new components are inserted into Firefox's interface. We define the file which will be inserted into, and the file which contains the overlay. It's unnecessary for our overlay file to be strictly called **browser.xul**, but it may as well be for simplicity.

Next, we define where the content of our extension is and, as we saw in the diagram, this is the **chrome/content/** directory, so that's what we specify.

Finally, we define where our language files will be kept. You have to have one definition for each language you support, defining the language code and the directory where the files are. Since we're only supporting one language in this example, we only need one definition. But if you want to support additional languages you would add them underneath, for example:

```
locale  myextension  es  chrome/locale/es/
locale  myextension  de  chrome/locale/de/
```

Each of those directories in turn would contain another **lang.dtd** and **lang.properties** file.

Now we need to define **install.rdf**, and this is an XML file with the following basic contents:

```
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:em="http://www.mozilla.org/2004/em-rdf#">

  <Description about="urn:mozilla:install-manifest">

    <em:id>myextension@mysite.com</em:id>

    <em:version>1.0</em:version>
    <em:iconURL>chrome://myextension/content/logo.png</em:iconURL>

    <em:name>My Extension</em:name>

    <em:description>
      A brief description for the install dialog and add-ons list
    </em:description>

    <em:homepageURL>
      http://www.mysite.com/extension/details/
    </em:homepageURL>

    <em:creator>My Name; http://www.mysite.com/</em:creator>

    <!-- firefox -->
    <em:type>2</em:type>
    <em:targetApplication>
      <Description>
        <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
        <em:minVersion>2.0</em:minVersion>
        <em:maxVersion>3.5.*</em:maxVersion>
      </Description>
    </em:targetApplication>

  </Description>

</RDF>
```

The most important point here is the first `em:id` element, which defines a *unique* identifier for our extension. You could use a **GUID**[7] (Globally Unique Identifier)

[7] http://www.guidgenerator.com/

wrapped in braces, as Firefox itself and many extensions do, or you could use a namespace-like value as we have in this example: `myextension@mysite.com`. The important point is that the value must be *absolutely unique*.

Most of the other elements should be fairly self-explanatory, and are all values used in the install or **Add-ons** dialog to give users more information. You can also include one or more `em:contributor` elements below `em:creator` to credit any other contributors, and there's an `em:aboutURL` element if you have an *About* page for your extension in addition to its home page. There's also an `em:updateURL` element for defining the URL of an updates meta file, but we'll avoid dealing with updates here; we can leave that to Mozilla when we submit our extension to its directory (which we'll talk about later).

The section at the bottom of the file, beginning with the `<!-- firefox -->` comment, defines the target application for our extension. That code must be *exactly* as shown if it's to target Firefox properly. As well as defining the add-on type (the value `2` means it's an extension) and Firefox's GUID, it also includes two elements that define the minimum and maximum versions we support, using wildcards; hence, a value of `3.5.*` means any version in the 3.5 branch.

You'll notice that even though we could technically support Firefox 1.5, the lower limit specified in the install file above is Firefox 2. The reason for this is simply to avoid having to do legacy testing; you'll find quite a few CSS differences in this older version, and it's arguably pointless spending the time correcting glitches for a version that nobody will be using anyway.

That previous example shows the install details for Firefox. If you want your extension to support Flock as well, then add the following code to declare it in addition:

```
<!-- flock -->
<em:type>2</em:type>
<em:targetApplication>
  <Description>
    <em:id>{a463f10c-3994-11da-9945-000d60ca027b}</em:id>
    <em:minVersion>1.0</em:minVersion>
    <em:maxVersion>2.5.*</em:maxVersion>
  </Description>
</em:targetApplication>
```

Step 3: Create a Basic Overlay

To create an overlay we use the `overlay` element, and inside that we *redefine* each of the elements we want to insert new content into; then we define the new content inside that, with an optional attribute that specifies a relative insertion point. We use the file **browser.xul** to specify our master overlays, so let's do that now:

```
<?xml version="1.0"?>
<?xml-stylesheet
    href="chrome://myextension/content/browser.css"
    type="text/css"?>
<!DOCTYPE overlay SYSTEM "chrome://myextension/locale/lang.dtd">
<overlay xmlns=
    "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
⋮
</overlay>
```

You can see that in this file we have a reference to our main style sheet, and a reference to the document type definition (`lang.dtd`) that defines our language. Language here is defined as custom entities which can then be referenced in the XUL code; more about that later. Then the root element in this file is the `overlay` element, which (as with all XUL documents) must define the root namespace.

Inside the `overlay` element we define each of the overlays we want. For example, if we wanted to add to the main status bar, we would redefine the status-bar element, and then define content inside it:

```
<statusbar id="status-bar">

  <statusbarpanel
      id="myextension-statusoverlay"
      insertafter="statusbar-progresspanel">
    <label>Hello World!</label>
  </statusbarpanel>

</statusbar>
```

So in the above example we're inserting a new `statusbarpanel` element inside the main status bar, and telling it to insert after the progress bar (the element with the ID of `statusbar-progresspanel`). We're unable to obtain absolute control over the position of our `statusbarpanel`—it would depend on whether any other extensions are using the same insertion reference—but it's as much control as we genuinely need. In Figure 1.3 we can see other extensions using the same insertion point as our Hello World example.

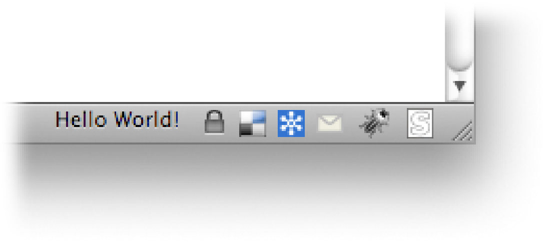


Figure 1.3. Multiple extensions using the same insertion point

Neither the insertion reference nor the `ID` of the status bar can be worked out programmatically, we just have to know what they are. This can be established by viewing Firefox's source code, inspecting **browser.xul** using Firebug or the DOM Inspector, or by looking at other extensions to see what they do (we'll see in the next step how to find the source of other extensions).

There are lots of places where you can insert new content; basically any part of the visible interface is game. To give another example, we could add a new item to Firefox's Tools menu, like this:

```
<menupopup id="menu_ToolsPopup">
  <menuitem id="myextension-toolsmenuitem"
      label="My Extension"
      insertbefore="sanitizeSeparator"/>
</menupopup>
```

Again, we've redefined the `menupop` element that is the **Tools** menu, defined a new menu to go inside it, and specified an insertion reference to determine where it goes, as Figure 1.4 demonstrates.

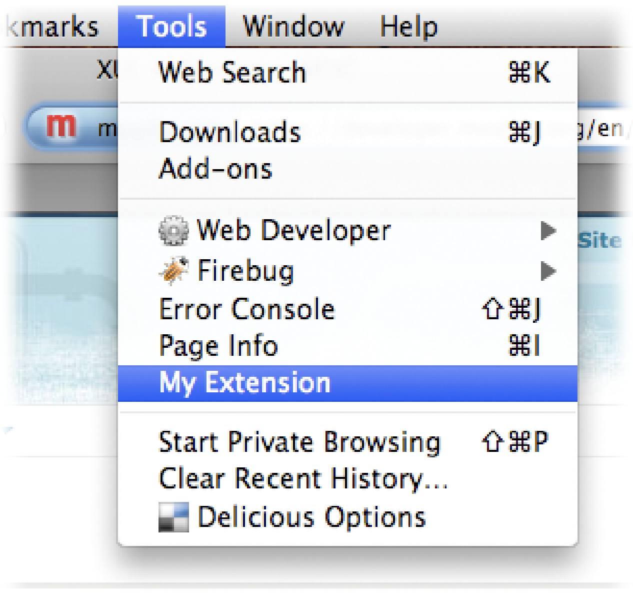


Figure 1.4. Inserting a new menu into the Tools menu

Step 4: Point Firefox at Your Development Folder

Now we have everything ready, the final step is to point Firefox at your development folder. To do this, you create a file with the same name as your extension's ID, enter the path to your development folder, and put it in Firefox's **extensions** directory.

If your extension ID has a value ending in `.com`—like the ID `myextension@mysite.com` from our example—this may cause you a slight problem on Windows, because it will think this is a binary COM file. You'll be without an **Open with…** option in the right-click menu, and if you open it in an editor like TextPad—which is binary-aware—it will also think it's a binary file. The simplest way to bypass this annoyance is to start up Notepad, then open the file from its **File** menu (or if you're creating it for the first time, save it from Notepad in the same way). Alternatively, if your ID is a GUID value in braces such as `{3c6e1eed-a07e-4c80-9cf3-66ea0bf40b37}`, then that should be okay. Make sure, though, that no extension is added to the file name, like `.txt`, which many editors will try to do unless you explicitly remove it. This is partly why it's important that you can view all file extensions, as we discussed at the very start.

Inside that file you simply add the path to your development folder. So if you're on Windows XP and you put the **myextension** folder directly inside **My Documents**, then the path will be as follows:

```
C:\Documents and Settings\User\My Documents\myextension\
```

On Windows Vista you put the **myextension** folder directly inside your **Documents** folder, then the path will be as follows:

```
C:\Users\User\Documents\myextension\
```

On a Mac, with the development folder directly inside **Documents**, it would be:

```
/Users/User/Documents/myextension/
```

You'll need to substitute *User* for your username, and the drive letter *C* if necessary; and add the trailing slash too, of course!

Now you should put that file inside your profile **extensions** folder; this is where all the extensions you have installed are kept, each with a folder containing all its source files. But we'll avoid putting our extension folder here for now; we're simply going to point to the development folder elsewhere.

This is where you need to be able to view hidden files on Windows; the path to your extensions folder on a Windows XP machine will be: **C:\Documents and Settings\*User*\Application Data\Mozilla\Firefox\Profiles\g5arepve.default\extensions** (the **Application Data** folder is the one that's hidden). On Windows Vista the path is **C:\Users\User\AppData\Roaming\Mozilla\Firefox\Profiles\g5arepve.default\extensions** (the **AppData** folder is the one that's hidden).

On a Mac the extensions folder will be at **/Users/*User*/Library/Application Support/Firefox/Profiles/*g5arepve.default*/extensions.**

Again, change *User* for your username, the drive letter *C* if necessary, and *g5arepve.default* for the name of your profile. (Chances are you only have one profile, so it will be easy to work out which folder it is!)

With the pointer file in place, restart Firefox—and if all is well it will automatically open the **Add-ons** dialog to indicate that a new add-on has been installed. You'll see the information in the dialog from **install.rdf**, showing the title, version, and description of your extension, as well as the 32x32 pixel icon.

You should also see the first bit of content we put into the extension itself: `Hello World!` written into the status bar, and **My Extension** in the Tools menu.

If it has failed to work, make sure of the following: the ID of your extension and the pointer filename match perfectly; the path inside the pointer file is correct and includes a trailing slash; and that your **chrome.manifest** file is correct, with all the right names and paths.

What's Next?

Now that we're all set up and our extension is up and running, the fun really starts! In the next chapter we begin building our extension using XUL and JavaScript.

Chapter 2

Developing and Testing Your Extension

There's so much you can do with a Firefox extension, whether you're looking to build tools that make other developers' lives easier, or usability widgets to improve the everyday surfing experience, or a more unusual task that meets a particular need.

A useful maxim when it comes to application development is to *solve your own problems*. By that I mean, make something that *you* need; be it application, widget, tool, or gadget. This philosophy gives rise to applications that solve genuine needs and perform useful tasks, as well as being built with genuine passion and care.

With Firefox development you're also able to experience the joy of developing for a single browser! No more cross-browser testing—or cross-platform either, unless you've done a great deal of custom styling with your interface, rather than relying on the default styling. You *probably will* have to test between older and newer versions (for example, between 2.0 and 3.5), but there again, only if you've done a lot of custom CSS, or used components that are only supported in later versions. Here again, the Mozilla Developer Center[1] (MDC) will steer you right, but if you've

[1] https://developer.mozilla.org/en/XUL/

used such components, then arguably you lack support for the earlier versions anyway.

In the next two sections of this book we're going to look at the two main aspects of extension development: XUL and JavaScript. It would be impossible for me to cover everything there is to say on the subject—even the MDC is unable to do that, and it goes on for ages. What I can do is introduce some of the most useful bits and pieces, and point out any gotchas you might encounter. For the most part however, it's up to you to experiment and figure out what's what—that's what I did!

The Development and Testing Cycle

The cycle of extension development follows a predictable routine of edit, restart, test. For most edits you would need to restart Firefox to see the changes—anything in scripting or style sheets, and anything in XUL inside the main overlay. However, if you disable the XUL cache it's possible to see your changes by opening a new window. To do this, you'll need to create a new boolean preference in your Firefox configuration. Go to the URL about:config to view the Firefox configuration page, right-click on the page, and choose **New** > **Boolean** from the context menu. Enter the preference name `nglayout.debug.disable_xul_cache` and give it the value `true`.

If you prefer to restart Firefox than constantly open new windows, then you'll need to get into the habit of reloading every time which is why it's so helpful to have the QuickRestart extension that I mentioned at the beginning of the previous chapter; it places a nice big button right on the toolbar to make restarting a cinch. There's also an extension called the "Extension Developers' Extension" which purports to offer useful tools, such as the ability to reload the chrome without restarting the browser, but it failed to work for me—I think it's just out of date.

XUL

XUL is an interface language, and its elements create structures that are used as interface components. Some XUL elements are standalone, while others are used in combination to make a particular structure. For example, an XUL menu is comprised of several elements that combine to make the overall structure, such as this simplified context menu from Dust-Me Selectors:[2]

[2] http://www.sitepoint.com/dustmeselectors/

```
<popup id="dms-statuspopup">
  <menuitem label="Find unused selectors"/>
  <menuitem label="Stop" disabled="true"/>
  <menuseparator/>
  <menu label="Automation">
    <menupopup>
      <menuitem label="Run automatically" type="checkbox">
      <menuitem label="Spider Sitemap...">
    </menupopup>
  </menu>
</popup>
```

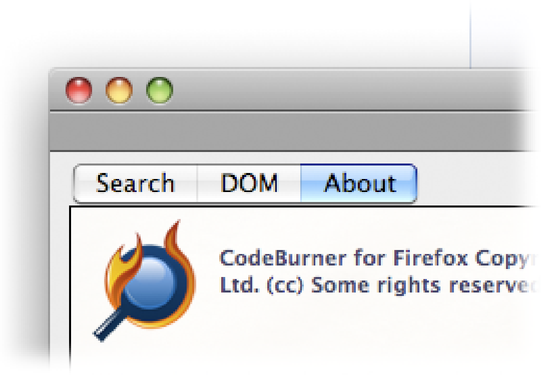
As you can see, the master element here is the `popup` element; it contains multiple `menuitem` elements and one `menu` element, which declares a nested submenu and has its parent label. The submenu in turn contains a master `menupopup` element, and this in turn contains multiple `menuitem` elements (`popup` always represents an independent menu like a context menu, while `menupopup` is always attached to another XUL interface in some way). That structure could continue indefinitely with further nested menus if necessary, as long as the correct structure is maintained.

One of the most important elements in XUL is `label`, which is both an element and an attribute. Most XUL elements cannot directly contain text content, so the `label` element is added as a wrapper to that text. Similarly, many XUL elements contain text as part of their layout—the text next to a `checkbox`, in a `menuitem` or inside a `tooltip`—and in all these examples the `label` attribute is used to contain the text.

The most common structural elements in XUL are `hbox` and `vbox`, and these elements simply control the orientation of their children: elements inside an `hbox` are laid out horizontally, while elements inside a `vbox` are laid out vertically. (I'd be lying if I didn't sometimes wish that HTML had similar elements, but then that would be contrary to the function of HTML, whose elements define the *meaning* of their content, rather than their structure or appearance. This to my mind is a reason why HTML is completely the wrong choice for building web applications … but I digress!)

You'll never just see a `label` or `hbox` out there on its own, because what would be the point?

Another good example of the interdependence of XUL elements is a tabbed structure, such as you'd find in a properties dialog, or like Figure 2.1 from CodeBurner.[3]



CodeBurner for Firefox Copy
Ltd. (cc) Some rights reserve

Figure 2.1. The tabs in CodeBurner

The basic code looks like this:

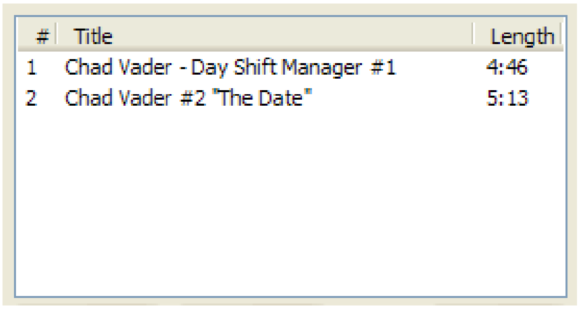
```
<tabbox flex="1">
  <tabs>
    <tab label="Search"/>
    <tab label="DOM"/>
    <tab label="About"/>
  </tabs>
  <tabpanels flex="1">
    <tabpanel flex="1">
      ⋮ panel content …
    </tabpanel>
    <tabpanel flex="1">
      ⋮ panel content …
    </tabpanel>
    <tabpanel flex="1">
      ⋮ panel content …
    </tabpanel>
  </tabpanels>
</tabbox>
```

In that structure, each `tab` corresponds with a `tabpanel`, and the contents of the panel are visible when its tab is selected. Note the `flex` attribute used on many elements there: that's a visual attribute that tells the element to take up as much

[3] http://tools.sitepoint.com/codeburner/

room as it can inside its parent—to be flexible—and is analogous to the behavior of a `td` element in HTML. Elements with a larger flex value take up proportionately more space than those with a smaller value; the actual value is insignificant. For example, if two elements in a row had flex values of `1` and `3`, then the second element would take up three times as much space as the first.

I could spend all day showing you different XUL structures, but we have other things to talk about! So let me give you one more structure to play with, and that's a simple `listbox`, used to build a multi-column layout as in Figure 2.2.



#	Title	Length
1	Chad Vader - Day Shift Manager #1	4:46
2	Chad Vader #2 "The Date"	5:13

Figure 2.2. A listbox interface

This comes from an as-yet unreleased extension I wrote for personal use, which is a simple playlist manager for YouTube.

The code looks like this:

```
<listbox id="tube-playlistbox" flex="1" seltype="multiple">

  <listcols>
    <listcol/>
    <listcol flex="1"/>
    <listcol/>
  </listcols>

  <listhead>
    <listheader label="#"/>
    <listheader label="Title"/>
    <listheader label="Length"/>
  </listhead>

  <listitem>
    <listcell label="1"/>
    <listcell label="Chad Vader - Day Shift Manager #1"/>
    <listcell label="4:46"/>
  </listitem>

  <listitem>
    <listcell label="1"/>
    <listcell label="Chad Vader #2 "The Date""/>
    <listcell label="5:13"/>
  </listitem>
</listbox>
```

You can see that the XML structure of a `listbox` is very similar to an HTML `table`, but it behaves more like an HTML `select` element, with items you can select from. In this case the `seltype` attribute declares that you can select more than one item simultaneously, just like the `multiple` attribute of a `select` element.

In the actual extension the example comes from, those `listitem` groups are generated on the fly. You can create XUL structures using DOM methods like `createElement` and `appendChild`, just as you would with HTML (there's more on this in the section called "JavaScript and the DOM").

There are so many different kinds of structures that can be built with XUL—some very simple, some quite complicated. The best way to learn is to look at other extensions and to browse around the MDC for pointers; there's really no substitute for just playing around to see what stuff does!

Styling XUL Elements with CSS

Just as with HTML, elements in XUL have default styling (provided by a global style sheet) and can be additionally styled using CSS. You can add a new style sheet to an XUL file using an `<?xml-stylesheet?>` instruction, as we did in our template:

```
<?xml-stylesheet
    href="chrome://myextension/content/browser.css"
    type="text/css"?>
```

Note the use of a `chrome://` URL in the style sheet address; that URL points to the extension's chrome folder, and should be used whenever external files are used, including scripts and images.

Then we can add rules to the style sheet just as you'd expect, using XUL element selectors, ID and other attribute selectors, and so on. And since we're working in a Firefox-only environment, we have the freedom to use the most advanced CSS.

For example, if we wanted the status bar text in our original template to have bold text, we could style it like this:

```
#myextension-statusoverlay > label
{
  font-weight:bold;
}
```

Another good example of using CSS in XUL is to add icons to menu items. These are implemented as a `list-style-image` property of the `<menu>` or `<menuitem>` element:

```
#myextension-toolsmenuitem
{
  list-style-image:url(chrome://myextension/content/logo.png);
}
```

Unfortunately, menu icons only show up on a Mac, they're unsupported on the Windows version. But they're still worth adding, I reckon, because they do look cool for those who can see them, as Figure 2.3 attests.

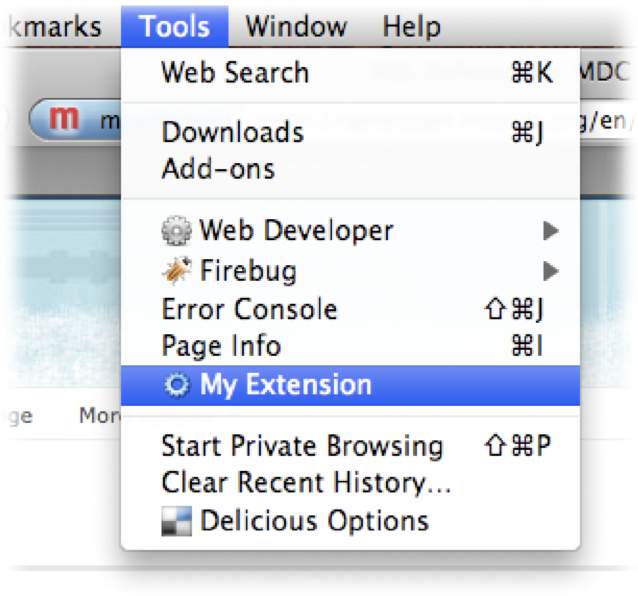


Figure 2.3. Adding an icon to a menu item

JavaScript and the DOM

No amount of XUL will be of tremendous use unless it hooks into scripted behaviors, but before we learn how to implement those hooks, let's take a step back and look more generally at the structure of an extension script.

Before we can do any scripting we obviously have to include the script, and XUL has a `<script/>` tag for this, just like HTML. But since we're in XML we can use a self-closing tag, and since Firefox understands it we can use the correct mime-type:

```
<script src="chrome://myextension/content/browser.js"
    type="application/javascript"/>
```

Now, the best way to structure the code for a Firefox extension is to use **object-based scripting**—place all your custom code within a single JavaScript object. This gives us a single point of reference to the scripting object, both internally from our methods and externally from XUL event handlers.

So inside our template **browser.js** file, I would suggest a basic framework like this:

```
var myextension =
{
  init: function()
  {
  }

};

window.addEventListener('load', myextension.init, false);
```

Any further properties and methods we need can be extended from the `myextension` object, remembering that each member must be separated with a comma (apart from the last one—trailing commas are not allowed in object literals as they are in arrays):

```
var myextension =
{
  init: function()
  {
  },

  anotherMethod: function()
  {
  },

  aStringProperty: 'foo'

};
```

We'll need the `init` function to take care of anything that needs to be deferred until the window `load` event. The XUL DOM is just like HTML or any other DOM in the sense that you're unable to access DOM objects until they've loaded. When you look at a Firefox window, what you're looking at is the top-level `window` object of the XUL DOM, and everything else descends from that. And so, just as we do in HTML scripting, we have a window `load` method to contain any such code.

`this` Might be Unavailable

One important point worth mentioning here is that the `this` keyword will sometimes be unavailable in this context, as predictably as you might think at first glance. If a method of an object is called from *any event-handling attribute*, then the `this` reference doesn't point to the object. The reason is that event-handling attributes are out of the object's scope. If you find that awkward to remember, it's easier just to avoid using `this` altogether, and use the single object reference instead (in this case, `myextension`).

Another important reason for using object-based scripting is **encapsulation**. Just as multiple scripts on an HTML page can conflict with each other, so multiple scripts in an XUL document can also conflict; in our case this principally means *other extensions*, although it also means parts of the Firefox codebase itself, of course. All of that code lives inside the same XUL DOM, so by keeping our code neatly encapsulated inside a uniquely-named object, we ensure that there are no conflicts.

It's also possible for extension scripting to interfere with scripting in the HTML document a user is viewing; if you add prototypes to built-in objects, they'll pollute that scope, so leave well alone!

Event Handling in XUL

The simplest way of associating interface actions with scripted methods is to use the event-handling attributes, the most common of which is `oncommand`, which is valid in virtually all XUL elements. For example, let's add one to our Tools menu item (shown in bold):

```
<menupopup id="menu_ToolsPopup">
  <menuitem id="myextension-toolsmenuitem"
      label="My Extension"
      insertbefore="sanitizeSeparator"
      oncommand="myextension.greet(this)"/>
</menupopup>
```

We then define the method that it calls in our script:

```
greet: function(menuitem)
{
  alert('Welcome to ' + menuitem.getAttribute('label'));
}
```

The `command` event will be fired by clicking on the menu item with a mouse or activating its keyboard trigger (see the section called "Adding Keyboard Accessibility" in Chapter 3 for how to add keyboard triggers).

These events can also be added in the DOM, using the same `addEventListener` syntax you'll already be familiar with:

```
var menuitem = document.getElementById('myextension-toolsmenuitem');

menuitem.addEventListener('command', function(e)
{
  alert('Welcome to ' + e.target.getAttribute('label'));
}, false);
```

This would have to be called from inside our `init` method, of course, because it refers to aspects of the XUL DOM that are unavailable at the point when the script is parsed.

Now in HTML, separation of content and logic is desirable, and so event handlers added in this way are preferable—they're *better* than the use of event-handling attributes. XUL is different, however. In XUL the content and logic are far more inextricably linked, with greater co-dependence and more closely associated semantics. XUL elements, in many cases, exist only as a hook to scripted behaviors, and XUL documents are virtually useless without them.

So a certain shift of mindset is necessary: abstracted event handling in XUL is not *better* or in any way intrinsically preferable, and there is no advantage in separating content and logic, nor is it particularly meaningful.

However, abstracted event listening can still have a useful role in XUL. Personally, I find it most useful for binding behaviors to elements that are created on the fly; for instance, if we'd built our **Tools** menu item in the DOM instead of in static XUL:

```
var menuitem = document.createElement('menuitem');
menuitem.setAttribute('label', 'My Extension');

var separator = document.getElementById('sanitizeSeparator');
var toolsmenu = document.getElementById('menu_ToolsPopup');

toolsmenu.insertBefore(menuitem, separator);
```

The Relationship between Properties and Attributes

Just as with HTML, XUL elements have scriptable properties that relate to specific attributes. A `checkbox` element, for example, has a `checked` attribute (which takes the value `true` or `false`), and a `textbox` has a `value` attribute, which can take any string value. However, when the user modifies the state of such an element the *attribute value may not change*, but the corresponding *property value always will change*. So if you were to produce code like this:

```
<textbox onchange="alert(this.getAttribute('value'))"/>
```

The `alert` dialog in this example will *always* produce an *empty string*. If you want to retrieve the actual value, you'd need to do this:

```
<textbox onchange="alert(this.value)"/>
```

It may seem obvious, but it's an annoying gotcha if you're unaware of it (he says, speaking from tedious experience!). The same caveat applies to all attributes that have a user-modifiable state: the attribute value *might* change, so it's unreliable—use the JavaScript property instead. These properties always have intuitive names, like `value` and `checked`, but if in doubt you can look them up in the MDC XUL Reference.[4]

Opening Windows and Dialogs

XUL has several types of window-level elements, the most general of which is `window`. This element should be used as a general container for interfaces that are outside of the main browser window. It's also the root element for the main browser window itself.

[4] https://developer.mozilla.org/en/XUL_reference/

But for producing dialog boxes, the element you want is `dialog`, which has a range of semantics in XUL and JavaScript specifically designed for such purposes. Here's the XUL for one of the dialogs from Dust-Me Selectors:[5]

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<dialog xmlns=
    "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
  id="dms-clearconfirm"
  buttons="accept,cancel"
  defaultButton="cancel"
  buttonlabelaccept="Yes"
  buttonlabelcancel="No"
  title="Confirm"
  ondialogaccept=
    "opener.DustMeSelectors_browser
        .doClearSavedSelectors(false, window);">

  <description style="margin-bottom:10px;max-width:300px;">
    Are you sure you want to clear data for the current site?
  </description>

  <checkbox id="dms-clearconfirm-clearconfirm" checked="true"
      label="Ask me every time"/>
</dialog>
```

Note that the dialog manually includes the global skin style sheet; you have to include that in all windows and dialogs, otherwise they'll have little or no default styling, and a completely transparent background!

What we have here is basically a slightly more sophisticated `confirm`, which looks like Figure 2.4.

[5] http://www.sitepoint.com/dustmeselectors/

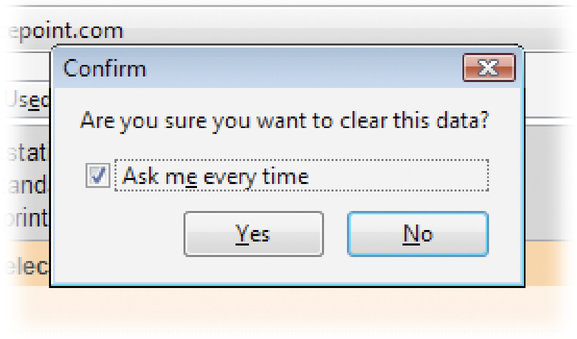


Figure 2.4. A modal confirm dialog in Windows Vista

You can see from the code that the choice of buttons, their layout, and their actions are all controlled by attributes of the `<dialog>` tag—such as `buttonlabelaccept`, which is the text on the accept button, and `ondialogaccept`, which is the event handler invoked when the accept button is clicked. You might extrapolate from this that the behavior of the cancel button would be specified by an event handler called `ondialogcancel`—and you'd be right, that's exactly what it's called!

The reason for defining the buttons using those attributes, rather than leaving it to us to add individual button elements, is so that the layout can vary according to the user's platform: on Mac computers the convention is that the accept button is to the right and the cancel button to the left, whereas on Windows machines it's the other way round. So that same dialog on a Mac looks like Figure 2.5.

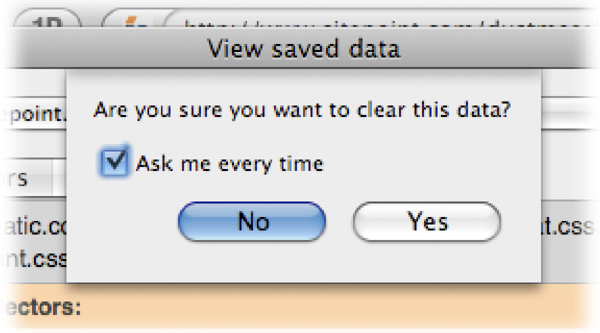


Figure 2.5. The same dialog, on a Mac

Dialogs are opened using `window.openDialog`. This is very similar to `window.open`, but it gives more control over the modality, placement, and decoration of the resulting dialog, courtesy of a wider range of definable features (as well as having different default features, such as no address bar). The dialog above, for example, was opened with this code:

```
window.openDialog(
    'chrome://dustmeselectors/content/clearconfirm.xul',
    'clearconfirm',
    'modal,centerscreen,resizable=no'
);
```

The first argument is the URL of the XUL document to be opened, and the second argument defines the window name, in case we need it for later reference. The third argument defines the window features, and is a comma-separated list of values that control aspects of the dialog's layout. All the same values are available as with `window.open`, plus there are several additional features that only work for chrome windows opened using `openDialog`.

In this example, **modal** means that the dialog is, er, modal: it restricts user focus to the dialog box, unless closed or acted on (just like a standard JavaScript `alert` or `confirm` dialog). The `centerscreen` option means to place the dialog directly in the center of the window (but for modal dialogs on a Mac this has no effect, as all such dialogs appear out of a *slot* beneath the title bar). The final option, `resizable=no`, should be fairly self-explanatory!

Another value that can be used is `dialog=no`, which, for a non-modal dialog, adds minimize and maximize/restore controls … although the latter will be disabled if "`resizable=no`" is also specified. (Conversely, if the dialog being opened is a `window` rather than a `dialog` element, setting `dialog=yes` will *remove* those controls that would otherwise be present.)

See https://developer.mozilla.org/en/DOM/window.open for a complete list of window features (in among lots of other information!).

Working with the Content Document

At some point in the development of your extension, it's very possible that you'll need to refer to the actual document the browser is currently viewing. You may want complete data about its DOM for an extension like Firebug[6] or the DOM Inspector;[7] you may want to scan its elements and style sheets to look for unused rules, like Dust-Me Selectors;[8] or you may just want to add a new item to the document context menu, like my playlist manager for YouTube.

Fortunately, Firefox provides a convenient `window.content` object that always refers to the `window` object of the currently visible document. Whenever the user changes this reference—by selecting a different tab, for example—the `window.content` reference updates automatically.

However, if you want to load a new page you should avoid overriding a user's main document—that would be rude. But with a reference to the embedded browser that contains it (available using the `getBrowser` method), you can use the `addTab` method to create a new tab and load the document into that:

```
getBrowser().addTab('http://www.sitepoint.com/', null);
```

The new tab opens the foreground or background, according to the user's tab preferences.

Using the Preferences System

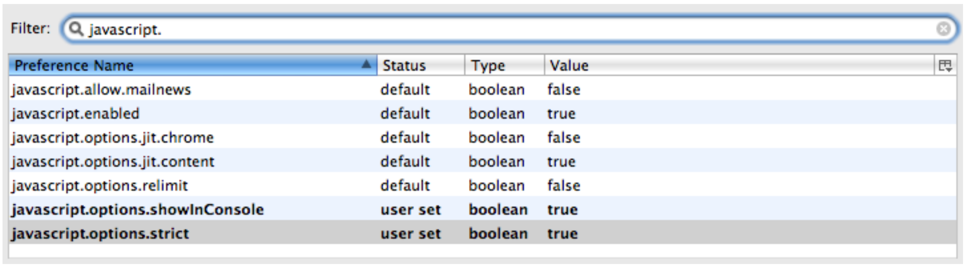
Firefox has a built-in preferences system that you can use to store all kinds of user data. If you're used to the limitations of working with cookies, you'll appreciate this feature. The basic types of data it can work with are strings, integers, and boolean values.

The first step in using this system is to create a reference to it, so let's create it as a property of our `myextension` object:

[6] http://getfirebug.com/
[7] https://developer.mozilla.org/En/DOM_Inspector
[8] http://www.sitepoint.com/dustmeselectors/

```
var myextension =
{
  'prefservice' : Components
      .classes["@mozilla.org/preferences-service;1"]
      .getService(Components.interfaces['nsIPrefService']),
  ⋮
}
```

We can then refer to it as `myextension.prefservice`, however there's still a bit more to do before we're ready to use it. The preference service is made up of **branches**, each of which contains a number of preferences grouped together, or another branch. You'll remember that we modified some of these preferences at the beginning to improve our development environment, and these were contained in the `javascript` branch. This is illustrated in Figure 2.6.


Filter: javascript.

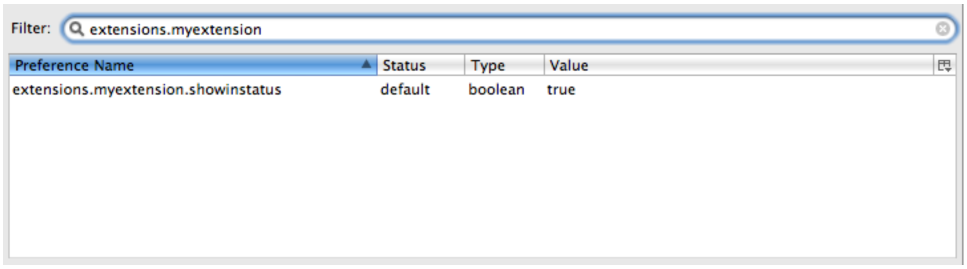
Preference Name	Status	Type	Value
javascript.allow.mailnews	default	boolean	false
javascript.enabled	default	boolean	true
javascript.options.jit.chrome	default	boolean	false
javascript.options.jit.content	default	boolean	true
javascript.options.relimit	default	boolean	false
javascript.options.showInConsole	**user set**	**boolean**	**true**
javascript.options.strict	**user set**	**boolean**	**true**

Figure 2.6. The javascript branch of the preferences system

We want to create a new branch for our extension, and good practice here is to keep it within the `extensions` branch. If you type "`extensions.`" into the search box you'll see all the different preferences that exist for the extensions you've installed. We can create a new branch simply by referring to it, so let's update our original definition like this:

```
var myextension =
{
  'prefservice' : Components
      .classes["@mozilla.org/preferences-service;1"]
      .getService(Components.interfaces['nsIPrefService'])
      .getBranch('extensions.myextension.'),
  ⋮
}
```

Now, whenever we define any new preferences they'll be in the `extensions.myextension` branch, as shown in Figure 2.7.



Filter: extensions.myextension

Preference Name	Status	Type	Value
extensions.myextension.showinstatus	default	boolean	true

Figure 2.7. Showing a custom preference in our branch

To use the preferences service then, there are getter and setter methods. For example, to set the boolean preference called "`showinstatus`" with the value `false` we would do this:

```
myextension.prefservice.setBoolPref('showinstatus', false);
```

Then we could read it back like this:

```
myextension.prefservice.getBoolPref('showinstatus');
```

The other typed methods are `setIntPref` and `getIntPref` for integers, and `setCharPref` and `getCharPref` for strings. Other useful methods are `clearUserPref` to clear a value, and `prefHasUserValue` to test if a user preference exists. If you try to refer to a preference that doesn't exist you'll receive an error.

However, you can be sure that the preferences you want do exist and define defaults for them by using the default preferences script. You remember how we created a `defaults` folder at the start, and inside that we created a script with the same name as our extension? Well that's the default preferences script, and inside that we use the `pref` method to define them. So for example, if we want our `showinstatus` preference to have the default value of `true`, we would do this:

```
pref('extensions.myextension.showinstatus', true);
```

As you can see, we define the complete branch and name, and the value of our preference. There's no need to worry about data types when using the `pref` method;

it will work them out automatically based on the values. Now when we read that value for the first time it will come back `true`. We might then want to use it for a task like whether or not to show the message in the status bar:

```
init: function()
{
  if(!myextension.prefservice.getBoolPref('showinstatus'))
  {
    document.getElementById('myextension-statusoverlay')
        .setAttribute('hidden', 'true');
  }
},
```

Another way to use the preference service is inside a `prefwindow` element. This is designed specifically for building preferences dialogs, and elements inside it can take attributes that link their state or value to a preference. However, in my experience, the implementation of `prefwindow` is littered with bugs, often incorrectly sized, badly laid out, or even completely blank! If you want to learn more about it and you have time to wrestle with its issues, visit https://developer.mozilla.org/en/XUL/prefwindow/. My recommendation would be to implement preferences dialogs manually using a regular `dialog` element, and you can find an example of this in Dust-Me Selectors.[9] For example, in a preferences window, the state of a `checkbox` can be automatically linked to a boolean preference. To implement it manually you simply need to read the `checkbox` state and modify the preference yourself, using the `dialog`'s `ondialogaccept` event as the trigger. Too easy!

Further Reading

There's so much more I could talk about here, but this chapter is way too long already! There are so many features to explore like using native drag and drop, and reading and writing to files.

You can find all of this stuff at the MDC, but a lot of it is quite intense and theoretical. The best way to learn at this stage is to *look at other extensions*. Find extensions that do similar tasks to what you want, and poke around inside them to see how

[9] http://www.sitepoint.com/dustmeselectors/

they work. For example, both drag and drop and file manipulation are implemented in GreaseMonkey,[10] so there's a place to look to find out more about them.

In fact, I learned quite a lot of what I know now from that extension in particular, as well as from other popular extensions like Firebug[11] and the Web Developer Toolbar.[12] And it never does anyone any harm to feel humbled from time to time, and reading through such accomplished programmers' code is a good way of doing so!

[10] https://addons.mozilla.org/en-US/firefox/addon/748
[11] http://getfirebug.com/
[12] https://addons.mozilla.org/en-US/firefox/addon/60

Chapter 3

Adding Accessibility and Internationalization

In this section we're going to look at two important topics. Firstly, we'll talk about improving access for people with disabilities by adding keyboard triggers to our interface. Secondly, we'll look at a framework for internationalization, using two different methods of externalizing language: one for static XUL and one we can use from JavaScript.

Adding Keyboard Accessibility

XUL has excellent semantics for keyboard accessibility, although they're less useful on a Mac as they are on Windows.

The first and most straightforward option is the `accesskey` attribute that can be added to certain elements, primarily menu items. On Windows, access keys are indicated with an underline on the access character, as you'd expect. On a Mac there's no indication at all, nor do they work in the same way.

For example, our **Tools** menu item currently looks like this:

```
<menuitem id="myextension-toolsmenuitem"
    label="My Extension"
    insertbefore="sanitizeSeparator"
    oncommand="myextension.greet(this)"/>
```

If we add an `accesskey` to that item, whenever the Tools menu is open, pressing that key will trigger the command action:

```
<menuitem id="myextension-toolsmenuitem"
    label="My Extension"
    insertbefore="sanitizeSeparator"
    oncommand="myextension.greet(this)"
    accesskey="M"/>
```

So now if you open the **Tools** menu (on Windows that would be **Alt**+**T**), then press **M** (no modifier), you'll receive the greeting just as if you'd clicked it with the mouse. On a Mac, pressing the key merely sets focus on the menu item; you still have to press **Enter** to fire its command event.

A more flexible and robust way to implement keyboard actions is to use `<keyset>` and `<key>` tags, and these have much better cross-platform support. For example, let's use the key combination **Alt**+**Ctrl**+**G** for Windows, or **⌥**+**⌘**+**G** (**Option**+**Command**+**G**) on a Mac, to trigger that greeting. Both of those modifier keys can be expressed in a platform-independent way, so there's no need to worry about the difference:

```
<keyset>
  <key id="myextension-toolsmenukey"
      key="G"
      modifiers="alt accel"
      oncommand="myextension.greet
          (document.getElementById('myextension-toolsmenuitem'))"/>
</keyset>
```

In the above example, `alt` means **Alt** on Windows and **⌥** on a Mac, while `accel` means **Ctrl** on Windows or **⌘** on a Mac.

Now, since we have an interface element which represents this action (as we should—just as it would be bad practice to have a mouse action with no key equivalent, it's bad practice to have a keyboard action with no mouse equivalent!), we can associate that `key` element with the `menuitem` in question, using its `key` attribute:

```
<menuitem id="myextension-toolsmenuitem"
    label="My Extension"
    insertbefore="sanitizeSeparator"
    oncommand="myextension.greet(this)"
    accesskey="M"
    key="myextension-toolsmenukey"/>
```

That association will cause a platform-specific key label to be drawn next to the menu text, as Figure 3.1 demonstrates.

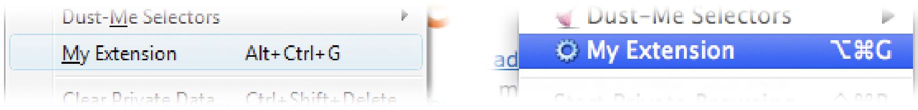


Figure 3.1. Platform-specific key labels

And now that we've made that association, we can press the key combination at *any time* to trigger the command action, even if the menu is closed! The `accesskey` assignment will still work too, when the menu is open. This is a fantastic way of creating shortcuts to commonly used actions, and benefits power users just as much as it benefits people with disabilities.

Note that even though we've now called the `greet` method twice, it only fires once; the associative logic is clever enough to understand what we want.

One final point to note is the `keyset` elements *must be deeper than the first-child* of the root element (such as `window` or `overlay`), or they may fail to work. In this case I've put them inside the `<menupopup>` tags that defined our Tools menu overlay.

Language Data in XUL

The language for Firefox extensions is defined in two different ways, and ultimately *all the language* you use in an extension should be implemented using one of these methods.

Language for XUL files is defined as entities in a Document Type Definition (DTD) file, kept inside the **locale** directory. We already have a blank **lang.dtd** file that we created at the beginning, so let's go ahead and add some language to it.

To define a language fragment we create a named entity, for example, the text in our menu item:

```
<!ENTITY myextension.title "My Extension">
```

Entity names are allowed to contain dots, which can help you to group them meaningfully, but more importantly it *ensures encapsulation*; that is, there will be no name conflicts with other entities in the same namespace (such as those defined by other extensions). Best practice is to begin all our entity names with the name of our extension, effectively namespacing them.

Then to add that to the XUL code we simply use the entity in place of the text:

```
<menuitem label="&myextension.title;"
```

We can also use the same method to define `accesskey` assignments, so that if a particular letter is no longer appropriate for a phrase in a different language, it can be easily changed to a more suitable one, so:

```
<!ENTITY myextension.accesskey.menuitem "M">
```

And to make use of it in the XUL code:

```
accesskey="&myextension.accesskey.menuitem;"
```

In our extension, all the language is defined in English and contained in the **en-US** directory. If you or anyone wanted to translate the extension, all they'd have to do is translate the text in each of the entities, then put the new DTD (still called **lang.dtd**) into a new locale directory that's named with the appropriate language-code. It would also need to be declared in **chrome.manifest**, as we discussed at the start.

This means that it's easy for a person to translate an extension without having to route through all the files; there's no need for them to even understand how any of this works—if they can type between two quotation marks, they can make the translation!

This elegant method of internationalization is so simple to use that there's really no reason to use anything else.

Language Data in JavaScript

DTDs are all very well for static language in XUL, but what if you need data you can refer to from JavaScript, such as the text in our greeting? That's what the second language file, **lang.properties**, is for.

The language in a properties file is defined in `name=value` pairs, for example:

```
greeting=Welcome to My Extension
```

Just as with the DTD, names may also contain dots to help you group them meaningfully; however, there is no issue with encapsulation here because the data will be private to our `myextension` object.

To make the data available to JavaScript, we firstly use the `<stringbundleset>` and `<stringbundle>` tags to include it in the XUL code. The URL must be an absolute `chrome://` address:

```
<stringbundleset>
  <stringbundle id="myextension-langbundle"
      src="chrome://myextension/locale/en-US/lang.properties"/>
</stringbundleset>
```

The `stringbundleset` element is a container for one or more `stringbundle` elements; in our case we only have one, but if you're using a great deal of language data you may want to split it into multiple files, included using multiple `stringbundle` elements as required.

Once we have that element, we need to save a reference to it, and since we're referring to the existing DOM we must put this inside our `init` wrapper:

```
init: function()
{
  myextension.lang =
    document.getElementById('myextension-langbundle');
```

And finally, to retrieve individual values from the properties file, we use the `getString` method:

```
myextension.lang.getString('greeting');
```

Now, that enables us to retrieve static strings, but what is often needed is a template string that we can insert dynamic data into. The way to do this is to have variable tokens in the properties strings, which can then be parsed using `String.replace`.

There's no particular convention here, but my preference is to use named tokens that begin with a percentage sign, such as `%name`. You could use the same convention as PHP's string parsing methods, like `%s` for a string and `%f` for a floating-point number, but personally there's no point to using such restrictive tokens when there's no other advantage to knowing their intrinsic type.

Let's look back at the `greet` method we wrote earlier, where we saw how part of that message came from the XUL element that triggered it:

```
alert('Welcome to ' + menuitem.getAttribute('label'));
```

We'll implement it using language from our properties file. First, we modify the greeting string so it has a token in place of the name:

```
greeting=Welcome to %name
```

And then we modify the output to use that string instead of hard-coded text:

```
alert(myextension.lang.getString('greeting')
    .replace('%name', menuitem.getAttribute('label')));
```

To make best use of replacement tokens you should keep in mind that the word order of phrases differs across languages. Take, for example, a phrase like "No, Mr. Bond, I expect you to die," where "die" is also a language variable. This could be expressed in string fragments like this:

```
die=die
escape=escape
sing=start singing

threat=No Mr. Bond, I expect you to %action
```

And then parsed in JavaScript like this:

```
var action = myextension.lang.getString('die');
myextension.lang.getString('threat').replace('%action', action);
```

Anyone doing a translation of that can then move the `%action` token around within the overall string to achieve the appropriate grammar. This approach is therefore much preferable to having the action separate from the rest of the string, as in:

```
die=die
escape=escape
sing=start singing

threat=No Mr. Bond, I expect you to
```

This is followed by:

```
var action = myextension.lang.getString('die');
myextension.lang.getString('threat') + ' ' + action;
```

We do this because the grammar of a different language may place the word “die” elsewhere within the sentence.

One final note here is that, just like `<keyset>` tags, `<stringbundleset>` tags *must be deeper than the first-child* of the root element (such as `<window>` or `<overlay>`), or they may fail to work. In our template I put them inside the first overlay as a child of the `statusbar` element.

The Big Moment Has Arrived!

So now you’ve built your extension and made sure it’s accessible and ready for translation, how do you let the world see what you’ve made? The next chapter will show you how to publish your extension, so that others may enjoy your fine work.

Chapter 4

Publishing Your Extension

So how do you let the world see what you've made? It's easier than you think!

Creating an XPI File

Well, the first step is to package it into an XPI file, and this is incredibly easy. All you have to do is create a ZIP file of the contents of your working directory; that's *everything inside but excluding* the **myextension** folder. Then simply rename the file with a `.xpi` extension, and there you have it. Another person can then install it just by opening the file in Firefox, or by dragging and dropping it onto a Firefox window.

But that lacks the required amount of user-friendliness. If you want to make it easy for others to install your extension, you need to create an install trigger.

Making an Install Trigger

An **install trigger** is a specially constructed link, placed on your web site, that will trigger the Firefox add-on install process.

First, create a link to the XPI file in HTML with an `onclick` event handler defined. This link uses the `onclick` attribute:

```
<a href="myextension.xpi" onclick="return install(event)">
  Install My Extension!
</a>
```

Clicking on the link above will call the following JavaScript code:

```
function install(e)
{
  if(typeof InstallTrigger != 'undefined')
  {
    try
    {
      var target = e.target;
      InstallTrigger.install({
        'My Extension' : {
          URL: target.href,
          IconURL: 'logo.png'
        }
      });
      return false;
    }
    catch(err)
    {
      return true;
    }
  }
  return true;
}
```

The code refers to an icon which will be used in the install dialog, so you'll need a copy of your original logo image to use there; you'll also need to define the title of the extension, which is also used in the installation dialog. The URL of the extension itself is extracted from the link's `href` attribute.

When the link is clicked in a browser other than Firefox, it will simply try to download the XPI file, but in Firefox activates the installation process. You'll see two visible elements. First, a security warning appears, like the one in Figure 4.1, that requires you to explicitly allow the installation—although this will only occur if the HTML page with the install link is viewed across a network, rather than locally.

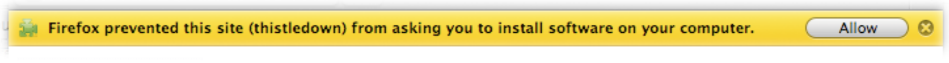
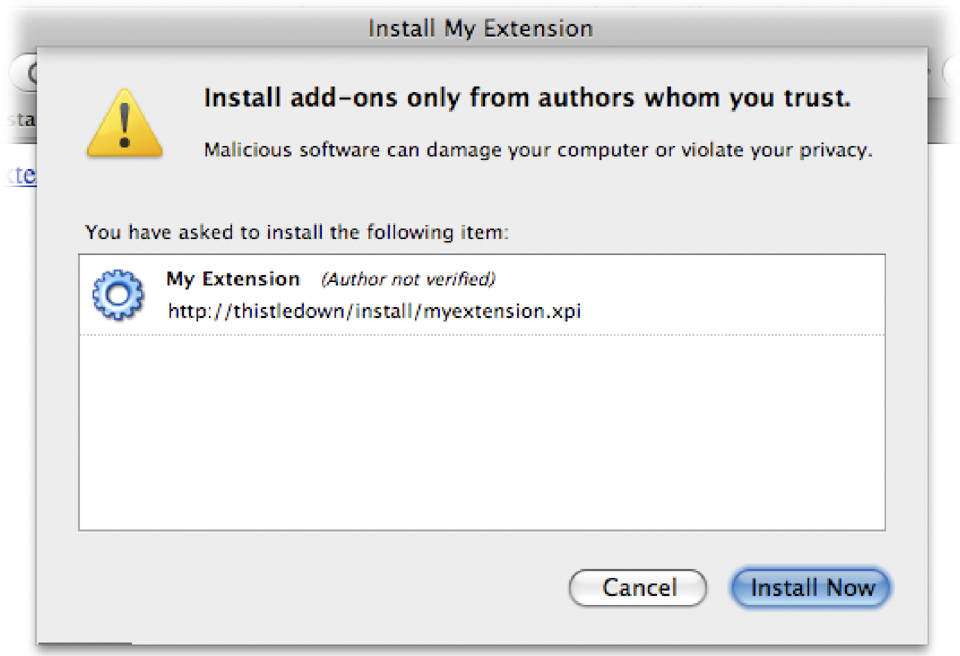

Firefox prevented this site (thistledown) from asking you to install software on your computer.

Figure 4.1. Pre-installation warning

Then if you allow that, you'll receive the installation dialog as illustrated in Figure 4.2.



Install add-ons only from authors whom you trust.

Malicious software can damage your computer or violate your privacy.

You have asked to install the following item:

My Extension (Author not verified)
http://thistledown/install/myextension.xpi

Figure 4.2. Installation dialog

Of course you should do this in a different profile to the one you're developing with, otherwise you'll overwrite your installation pointer file, and then you'll no longer see the changes from your working directory.

There's a mechanism for automatic updates, using a special meta file referred to from **install.rdf**, but you can make your life easier by just forgetting about that, and leaving it to Mozilla to deal with when you submit your extension to their directory.

Submitting Your Extension to the Add-ons Directory

Submitting an extension is simply a case of registering for an account, uploading your extension, completing some details about it, and then waiting! New extensions are immediately available but are always **sandboxed** to begin with, which means they're labeled as experimental, and only registered, logged-in users can download and install them. This is why you'll probably want to make it available from your own site as well, at first.

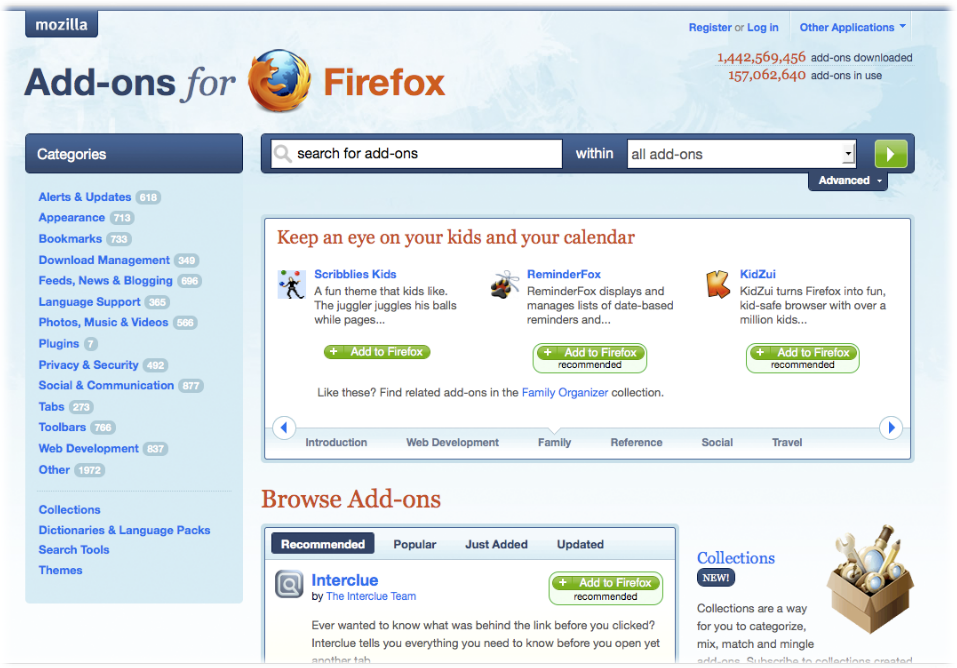

mozilla
Register or Log in
Other Applications
Add-ons for Firefox
1,442,569,456 add-ons downloaded
157,062,640 add-ons in use
Categories
search for add-ons
within
all add-ons
Advanced
Alerts & Updates 616
Appearance 713
Bookmarks 733
Download Management 349
Feeds, News & Blogging 696
Language Support 365
Photos, Music & Videos 566
Plugins 7
Privacy & Security 492
Social & Communication 877
Tabs 273
Toolbars 766
Web Development 837
Other 1972
Collections
Dictionaries & Language Packs
Search Tools
Themes
Keep an eye on your kids and your calendar
Scribblies Kids
A fun theme that kids like. The juggler juggles his balls while pages...
ReminderFox
ReminderFox displays and manages lists of date-based reminders and...
KidZui
KidZui turns Firefox into fun, kid-safe browser with over a million kids...
Add to Firefox
Add to Firefox recommended
Add to Firefox recommended
Like these? Find related add-ons in the Family Organizer collection.
Introduction
Web Development
Family
Reference
Social
Travel
Browse Add-ons
Recommended
Popular
Just Added
Updated
Interclue
by The Interclue Team
Add to Firefox recommended
Ever wanted to know what was behind the link before you clicked? Interclue tells you everything you need to know before you open yet another tab
Collections
NEW!
Collections are a way for you to categorize, mix, match and mingle add-ons. Subscribe to collections created

Figure 4.3. The Mozilla Add-ons for Firefox site

Once your extension has been on the site for a while you can nominate it for promotion out of the sandbox; this enables it to be generally available for installation without users being logged in, and automatic updates are enabled. Nominated add-ons are all reviewed manually, so expect to wait a few weeks for this to happen.

Go to https://addons.mozilla.org/en-US/firefox/ (shown in Figure 4.3) to find out more and register for an account, then you can submit your extension through the Developer Tools at https://addons.mozilla.org/en-US/developers.

Now It's All Up to You!

I hope you've enjoyed this book and that you'll find it useful. Building Firefox extensions is great fun, and if you make a really useful extension, the community feedback can be very encouraging. There's no real money in it, but as with everything in this industry, good work breeds good reputation, and that's what brings in business.

Also, make sure you download the code archive for this book.[1] Within archive you'll find two files:

- **myextension_template1.zip** is a basic template extension that includes everything we covered in Chapter 1.
- **myextension_template2.zip** is the same basic template plus all the stuff we implemented in examples, including a sample installer.

[1] http://www.sitepoint.com/books/byofirefoxpdf1/code.php

Appendix A: Introducing Jetpack

Jetpack is essentially a collection of APIs—to aspects of chrome functionality, to bundled libraries such as jQuery,[1] and to bundled web APIs (currently limited to the Twitter[2] API). Anyone with knowledge of HTML, CSS, and JavaScript (that is, the technologies you already know and use to build web sites and applications) can develop a Jetpack extension by making use of these APIs.

Well, I say that, but in fact the kind of extension you can develop using Jetpack is unable to compare with what's possible using XUL. Just consider some of the limitations:

- Jetpack provides *only one* insertion point into the existing Firefox chrome, namely, the status bar (no good for users who have the status bar disabled!). Future versions of Jetpack will allow insertion of menu items, but only single items, not submenus.

- Jetpack provides a limited range of scripting components. All the most important stuff is there, such as access to tabs and the content document, a persistent storage API, JSON utilities, and cross-domain `XMLHttpRequest`—but limited to what's provided through the Jetpack API.[3]

Having said that, Jetpack does have some handy features that take away some of the hassle of extension development, for example:

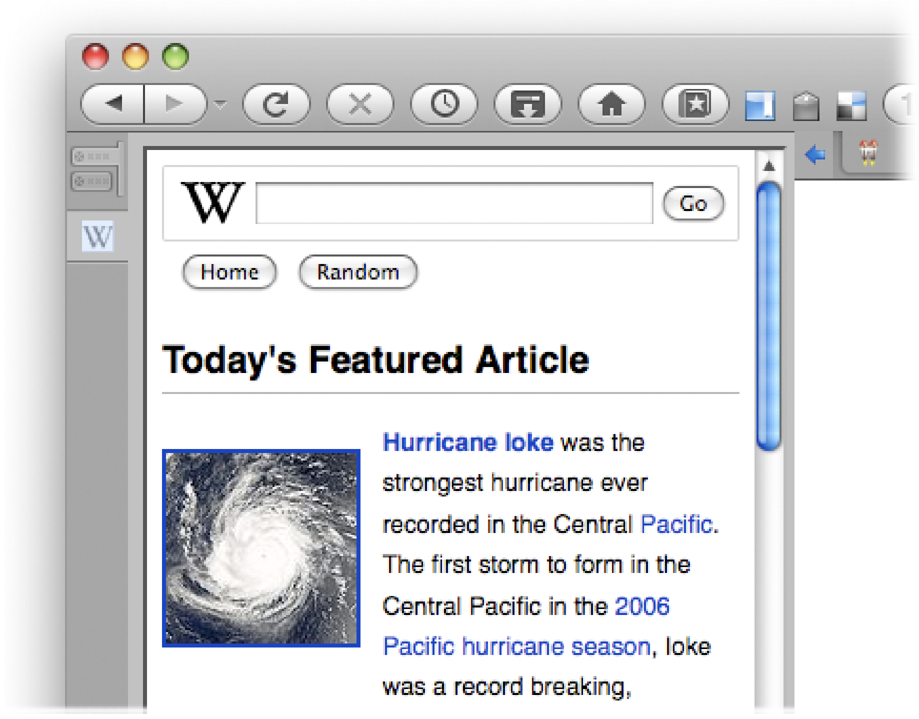
- Programming with the Jetpack API means there's no need to track Firefox versions and update your extension when each new one comes out; Jetpack takes care of that for you.

- Jetpack's tab access mechanism takes the pain out of monitoring the user's browsing, as it always gives you access to the currently focused tab.

It also provides some unique UI components which, for regular extension development, you'd have to roll yourself. For example:

[1] http://jquery.com/
[2] http://apiwiki.twitter.com/
[3] https://jetpack.mozillalabs.com/api.html

- The Jetpack *SlideBar*, which is a retractable sidebar, seen expanded in Figure A.1. It's unique to Jetpack, and can be used to display information.



Today's Featured Article

Hurricane Ioke was the strongest hurricane ever recorded in the Central Pacific. The first storm to form in the Central Pacific in the 2006 Pacific hurricane season, Ioke was a record breaking,

Figure A.1. The Jetpack SlideBar

- Easy access to operating system level notifications.

How Are Jetpack Extensions Programmed?

Jetpack extensions are programmed in JavaScript using the aforementioned Jetpack API. All the demo examples[4] use jQuery, but other libraries are planned for inclusion (and of course, using a library at all is totally up to you).

Interface components are coded in HTML, and the mechanism allows you to insert entire HTML pages—rather than just fragments of HTML—into Firefox's chrome! Well … into the status bar anyway which, as I mentioned before, is the only place you can currently insert UI content (apart from the SlideBar, but that's for displaying

[4] https://jetpack.mozillalabs.com/#demos

information, not interface widgets like buttons and menus). Figure A.2 features the status bar display from the Email Notifier demo,[5] and a few other Jetpack extensions.

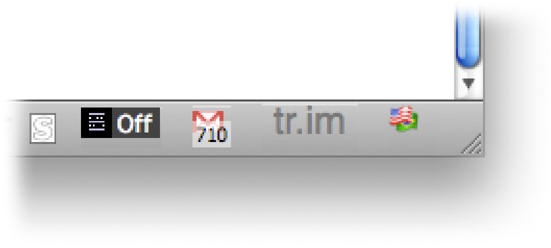


Figure A.2. Status bar information for the Email Notifier demo

The Jetpack documentation makes a song and dance about its use of HTML, and the fact that it gives you access to open web-based technologies such as the `canvas` element. But in fact this is *already* the case: you can currently integrate HTML into Firefox add-ons, either by using `browser` elements to load or generate HTML pages (which is what Firebug does for its panels), or by inserting XHTML directly into XUL using namespaced methods (such as I did in places in Dust-Me Selectors[6]).

But what will make it attractive to a certain niche of developers is that it's all so much easier than programming with XUL. Not *inherently* easier (as I've said, neither are difficult to learn), it's just that a) there are no new languages to learn, and b) the Jetpack API provides much friendlier wrappers around the scripted components.

[5] https://jetpack.mozillalabs.com/demos/gmail-checker-install.html
[6] http://www.sitepoint.com/dustmeselectors/

Here's a short example of a Jetpack script (paraphrased from the Jetpack tutorial at https://jetpack.mozillalabs.com/tutorial.html):

```
jetpack.statusBar.append({
  html: "<b>Tabs Info</b>",
  width: 100,
  onReady: function(widget){
    $(widget).click(function(){
      console.log( jetpack.tabs );
    });
  }
});
```

What that does is add the bold text "`Tabs Info`" to the status bar which, when clicked, writes information about all the currently open tabs to the Firebug console. (Jetpack uses Firebug for its debugging and logging output, to save re-inventing that particular wheel.)

I mentioned previously that Jetpack allows easy access to operating system level notifications. These are displayed using Growl for Mac OS X and toaster notifications for Windows and Linux, as demonstrated by Figure A.3.

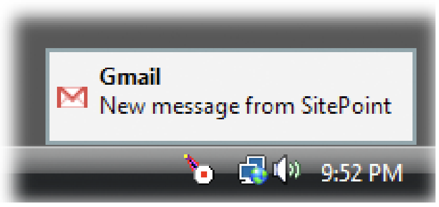

Gmail
New message from SitePoint
9:52 PM

Figure A.3. A notification message in Windows

To create a notification message, you simply do this:

```
jetpack.notifications.show({
  title: 'Gmail',
  body: 'New message from SitePoint',
  icon: 'http://mail.google.com/mail/images/favicon.ico'
});
```

Using the Notification System in Your Extension

On a side note, I have to say thanks to the Jetpack developers for showing me how to use the notification system. If you want to generate notifications like those above in your XUL-based extension, it turns out to be *only two lines of code* (exception handling notwithstanding):

```
var alertService = Components.classes
  ["@mozilla.org/alerts-service;1"]
  .getService(Components.interfaces['nsIAlertsService']);

alertService.showAlertNotification(
  'http://mail.google.com/mail/images/favicon.ico',
  'Gmail',
  'New message from SitePoint'
);
```

How Does Jetpack Compare to XUL?

If you're anything like me, you may still need convincing. I guess I'm a bit of an old-fashioned guy, in the sense that, when I write a JavaScript application I want to feel like I *really* wrote it. *Completionism* is all very well—after all, it's the end result that's the point—but I enjoy programming, that's why I do it; if so much of the *hard work* is done for me in APIs and wrappers, I don't feel like I'm really programming anymore. Instead, it feels as if I'm painting by numbers, and I fail to enjoy myself.

That's why I give libraries a miss, and why I rarely use APIs if I could write their functionality myself (even if I'm re-inventing the wheel—maybe my wheel will be better!). Still, I also accept that *all* high-level coding is a level of abstraction, and it's perhaps an arbitrary distinction to draw the line in one place rather than another.

But even if you're nothing like as masochistic as me, let's face it (as I hope I've demonstrated through the rest of this article): writing Firefox add-ons using XUL really is quite easy.

One of the beautiful factors about XUL is that its elements only describe the *function* and *physical layout* of UI components, instead of attempting to describe those components' appearance. The appearance—as in the fonts, colors, backgrounds, borders, and so on—is all determined by the user's skin and operating system. This gives rise to superb visual accessibility with little or no effort on the developer's part, because everything you define just naturally slots in with the pre-existing appearance of the user's interface.

Components written with Jetpack, on the other hand, will have exactly the same accessibility issue that all web applications have: namely, that each and every one has its own, unique appearance, with little or no deference possible to the user's chosen scheme—be it high-contrast colors, large fonts, or otherwise.

Every interface developed in HTML must be learned by the user from scratch. This is an inherent problem with developing interfaces in HTML, and therefore a problem that *every* Jetpack extension will create; that wouldn't be the case if that extension were developed with XUL.

Here's What Jetpack is *Really* Good for

There's one aspect of Jetpack development which I've yet to mention which, in my opinion, makes *all the difference*, and fundamentally changes my attitude towards the whole project:

Jetpack extensions can be developed and installed without the need to restart the browser.

From a developer's point of view this makes development faster. But far more importantly, from a site owner's point of view, it suddenly turns Jetpack into a mechanism for giving users more control over the behavior of their site. So Jetpack, in this sense, is like *Greasemonkey on steroids*.

Consider that you're using a favorite site or web application that you trust; that site can now make scripts available, which modify the browser's behavior to suit the application. These scripts can provide privileged functionality to enhance the application; they can provide modifications that control the appearance of certain kinds of content (in much the same way that Greasemonkey[7] or content-filtering extensions like AdBlock[8] do now).

[7] https://addons.mozilla.org/en-US/firefox/addon/748
[8] https://addons.mozilla.org/en-US/firefox/addon/1865

In short, Jetpack scripts can enhance the behavior of the browser *on a per application basis* (providing you trust that application), and they can do this in a completely seamless way. All one has to do to offer a Jetpack script for installation on one's web site is this:

```
<link rel="jetpack" href="my-script.js">
```

And all the user has to do is click a button to install it, as Figure A.4 demonstrates, and then carry on browsing. No browser restart is required.

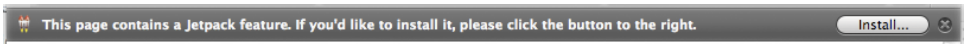

This page contains a Jetpack feature. If you'd like to install it, please click the button to the right.

Figure A.4. A Jetpack install notice

The seamlessness of it all may seem like just a small aspect, but it makes a very big difference. And it also changes Jetpack's functional limitations *from a weakness to a strength*.

Users may be unprepared to install a fully-fledged Firefox add-on just to suit one application. The vast range of tasks an add-on can do means a vast range of potential security concerns, and if the installation is outside Mozilla's official add-ons site, there's no independent verification that what you're installing is safe. But a Jetpack script, with its limited range of privileged access, is a far less risky—and therefore easier—proposition.

This, in my opinion, is what Jetpack is *really* good for.

So Where's it Heading?

Jetpack will never take over as the primary mechanism for extending Firefox: its niche is entirely different. And I think it will fail to attract developers who want to build extensions but lack the motivation to learn XUL.

I think Jetpack will attract an entirely different niche of developers, and only time will tell what weird and wonderful stuff they'll find to do with it! As William Gibson[9] famously said, "the street finds its own uses for things."

[9] http://en.wikipedia.org/wiki/Hackers_(anthology)#.22Burning_Chrome.22

Inevitably, of course, Jetpack will add more fuel to the fiery debate over user modification versus old-media content control—but I hardly think that's a bad outcome!