Developing for Firefox

- The Mozilla Mission
- What can you do with Firefox?
- Anatomy of an add-on
- The grand old way to do it – XUL
- A boilerplate XUL add-on
- A new way to develop – the Jetpack API
- A boilerplate Jetpack add-on
- The case for choosing Jetpack over XUL
- An exercise in comprehensive development
- The new frontiers
Join the forum to express your views and resolve your differences in a more civilised way.

Post your queries and get instant answers to all your technology related questions.

One of the most active online technology forums not only in India but worldwide.

JOIN NOW www.thinkdigit.com
FAST TRACK to DEVELOPING FOR FIREFOX powered by digit
The Mozilla Mission
The tech world is in great shape right now, thanks in no small part to a small project that was born in 1998 for this very purpose.

What can you do with Firefox?
You can create a wide array of extensions such as add-ons, plug-ins and themes.

Anatomy of an add-on
It's time to look under the hood to know how exactly an add-on works.

The grand old way to do it – XUL
To know our future, we must get to know our past!

A boilerplate XUL add-on
Here we put our knowledge to some use and create a template.
CONTENTS

38
PAGE

A new way to develop – The
Jetpack API
Learn from the past, look to the future!

50
PAGE

A boilerplate Jetpack add-on
This is where we’ll teach you to create a boilerplate add-on with the help of Jetpack

55
PAGE

The case for choosing
Jetpack over XUL
When you have to pick sides, make sure you pick the right one

60
PAGE

An exercise in comprehensive
development
It’s time to take those training wheels off!

80
PAGE

The new frontiers
The Mozilla project has grown way beyond the ‘friendly neighbourhood browser’ stage. The team is working on creating a gamut of technologies and, of course, all of them could use your help!
Introduction

We at Digit have always had a soft spot in our hearts for Open Source, like most other people with a deep interest in having worldwide harmony prevail – in the digital form at least, because the real thing isn’t anywhere on the horizon. But yes, open source has grown by leaps and bounds, and it fills our hearts with joy and renewed hope. A few years ago, you’d know you’ve run into a power user when you found them all up in arms about the kind of proprietary practices rampant in the industry.

Fast forward to today, and you see that more people are looking up to free and open source software than ever before. It seems like something good did come out of the whole NSA fiasco. Don’t get us wrong, every human being ever loves free stuff, but it’s encouraging to see the move from “free as in free beer” to “free as in free speech”. If you’ve been with us for long (tech hipsters, y’know), you’d be aware of our agenda to reach out and lead some converts to the too-good-to-be-true-but-let’s-try-it-anyway land of open source, which we embarked on about a decade ago when a new Linux distro was made a staple of every month’s DVD. In fact, the seeds for this month’s Fast Track were planted years ago, when the writer of this month’s booklet had his first brush with Linux through Digit’s DVDs.

It is said that if you can positively affect a life, then yours is a life worth living. So we’ve gone ahead and tried to touch all our readers’ lives (with at least one guaranteed success), and give you a peek into what a flagship open source organisation looks like, what it does and how you can help. This one’s to Mozilla!

Disclaimer: Digit does not claim ownership of any piece of code released as a part of this book, which continue to be unaffectedly governed by the associated (citation to the) respective licenses.
INTRODUCTION: THE MOZILLA MISSION

The tech world is in great shape right now, thanks in no small part to a small project that was born in 1998 for this very purpose.

To a regular person from the 90s, our current world would seem really strange. Although there would be a million reasons to account for that, the Internet would take up a huge chunk of those reasons. The speed at which we’ve moved forward in that regard is, for lack of a better word, surprising. This is because today, a large faction of the community is an active participant in the medium, as opposed to being silent spectators. Web 2.0, ‘the web the world made’ is so mainstream, it’s almost old. It’s marvellous when you think about the kind of journey we’ve had to take to arrive at our present reality.

It all started in the great browser wars of the 90s. Netscape Navigator – as loved as it was – was falling behind both, Internet Explorer (which believe it or not, was at the forefront of innovation at the time), and the open web standards in general, due to its minimal support of new-age features such
as dynamic HTML. Therefore, it was decided that the rendering engine at the core of the browser needed a complete overhaul. Work began on a new engine from scratch, which was supposed to be the fifth iteration of the Netscape browser. Mozilla started out as an entity that oversaw the development of the codebase which was made public when Netscape Navigator (and its new rendering engine that was designed to better implement the web standards as defined by the W3C) was declared as Open Source and released to the public in 1998, around the time that Netscape itself was bought out by AOL. In 2003, AOL backed out of the project and Mozilla took on a larger role in the development of the browser.

One of the defining features of Mozilla as an organisation is the degree to which it engages with its community. In fact, it’s widely known that volunteers in every part of the organisation are the most important cogs in the wheel. Of course, positioning itself as a non-profit organisation endears it to those disillusioned by the pseudo-dictatorships in the tech scene. Working together with the people who share the Foundation’s common values of openness – to paraphrase what’s perhaps the most fitting description of its aims by a Mozilla employee – it’s making the web what the world needs it to be.

A very significant development that we take for granted today, simply because we’ve grown so used to it being available that its absence would be considered blasphemy, is the abundance of free software around us. In an era that’s so acutely money oriented, it almost seems counter-intuitive that so much of the internet is free. But it is, as championed by a huge Open Source community, of which Mozilla forms an integral part.

Despite being among the foremost web products of this day and age, one thing that strikes you about Firefox, and Mozilla in general, is that
it’s never actually been a part of a ‘feature race’. In fact, it has followed the W3C guidelines to the tee.

The Mozilla mission, as understood by those involved, can be reduced to three very important points:
1. The web should be open
2. The web should be interoperable
3. The web should be ours

As straightforward as this may sound, it takes a great deal of hard work to conform to these rules. Let us throw some more light on these rules.

1. ‘The web should be open’
The goal is very simple – the internet should not be a place for restrictions – anything that’s put out there should be within the reach of any user accessing the internet (short of key personal and sensitive data, obviously).

2. ‘The web should be interoperable’
This is one of the most important goals across the organisation, and is the idea that drives a significant amount of work that gets done here – the thought that at no point should the compatibility of services on the web be compromised. Mozilla believes that at no point should anybody take the fate of the entire internet in his/her own hands.

3. ‘The web should be ours’
This, again, is a motto central to the idea that is Mozilla. The thought that anybody who wishes to do so, should be able to make meaningful contributions to the internet. It is the people, the users, who should be the driving force behind this mission, and that the development should take a democratic route is a summation of this rule.

Now, as one of the things that sets it apart, the morality of the organisation shines through in the way that it should. The Mozilla Foundation bears a pledge outlining the goals of the organisation, according to which it will:
- build and enable open source technologies and communities that support the Manifesto’s principles;
- build and deliver great consumer products that support the Manifesto’s principles;
- use the Mozilla assets (intellectual property such as copyrights and trademarks, infrastructure, funds, and reputation) to keep the internet an open platform;
promote models for creating economic value for the public benefit; and
promote the Mozilla Manifesto principles in public discourse and within
the internet industry.

The manifesto itself is a comprehensive 10-point plan that details the
path that the company chooses to take. In its entirety, it’s as follows:
1. The internet is an integral part of modern life – a key component in
   education, communication, collaboration, business, entertainment and
   society as a whole.
2. The internet is a global public resource that must remain open
   and accessible.
3. The internet should enrich the lives of individual human beings.
4. Individuals’ security on the internet is fundamental and cannot be
   treated as optional.
5. Individuals must have the ability to shape their own experiences on
   the internet.
6. The effectiveness of the internet as a public resource depends upon
   interoperability (protocols, data formats, content), innovation and
decentralised participation worldwide.
7. Free and open source software promotes the development of the internet
   as a public resource.
8. Transparent community-based processes promote participation,
   accountability and trust.
9. Commercial involvement in the development of the internet brings
   many benefits; a balance between commercial goals and public benefit
   is critical.
10. Magnifying the public benefit aspects of the internet is an important
    goal, worthy of time, attention and commitment.

One thing that comes to mind immediately when you read the manifesto
provided by the organisation in detailing its aims for the future is that it
constitutes goals entirely unlike any other organisation in the browser cat-
egory. The not-for-profit nature of the organisation shines through, as does
the general intent to make the internet a better place for everybody involved.

Now, for the more visible of components, the main browser itself. Over
the past few years, as WebKit-based browsers such as Chrome and Safari
have gathered clout, the general clamour has been that, coming full circle,
it is Firefox that has become archaic in its approach to web browsing, and
that the Gecko model can’t keep up with latest developments in the tech
space. It’s worth mentioning here that it was Firefox, in fact, which began
the ‘modern browser’ race, and pioneered extensions on a browser platform. Having pioneered the concept of extensibility to a browser’s experience, Firefox paved the way (along with many other developments, of course) for ushering us into the new-age era of apps and smart-everything. Now, in principle if not always in reality, it’s hard to miss the similarities between a modern web browser and the mobile ecosystem. In fact, Google has taken this very approach and developed and deployed the Chrome OS on the Chromebooks that it has started selling as low-spec internet laptops.

Since we can’t continue to bask in the glory of the past, it can only mean a direct head-to-head between the best and latest browsers. While WebKit, being a newly built engine, enjoys the benefit of having a streamlined code-base for now, expect that advantage to diminish as time goes by. Numerous tests have been performed to test the allegedly ‘ageing’ Firefox against its new competitors, and judging from results by a rather reliable source in Tom’s Hardware, we can see that after a few years of dormancy, Firefox has risen up again to take the lead, in the benchmarks at least.

Of course, browser wars are not the bloodiest battles, and most breakthroughs have a way of finding themselves with either camp in a reasonably short span of time. While the makers may fight it out to gain that fraction of a performance boost, it can only be a win-win for the consumer, who gets to witness the latest and the greatest that some of the best minds can conjure up.
WHAT CAN YOU DO WITH FIREFOX?

Anything (well, almost)

You can create a wide array of extensions such as add-ons, plug-ins and themes

As the more tech-savvy readers would be aware, Firefox runs on Gecko, a free and open source layout engine. So, in essence, you can develop extensions for your browser. We’re using the term ‘extensions’ here in a very broad sense since it means anything created that was not shipped with the plain vanilla browser. There are many types of extensions that you can create for Firefox to customise your experience, and (hopefully) others’ as well, by sharing (the love and) the code.

At the time of writing, there are three broad categories of extensions that can be deployed using Firefox: add-ons, plug-ins and themes. There are some extensions that obviously blur the line between the three types, such as the excellent Vimperator extension that makes the browser conform to the look and feel of the very popular Vim text editor, or the very popular web development tool called Firebug, but in general, these are the
three rough divisions under which the extensions created for Firefox are classified. Let’s take a look at each of them.

- **Add-ons**
  Add-ons are extensions that are generally expected to add some new functionality to your browser, that may include (but is in no way limited to) new key bindings (keyboard shortcuts) and new menu items. If you’ve added an element that wasn’t present before (note the use of the word ‘adding’ instead of ‘replacing’) it’s called an add-on. They’re segregated into sub-categories, viz. SDK-based (or Jetpack) add-ons and non-SDK add-ons.

- **Plug-ins**
  Plug-ins are the kind of extensions that try to modify the behaviour of the native Firefox code and thereby the behaviour of the browser. Maybe you developed a new way to view PDF files within the browser, or discovered a technique to analyse the communication over HTTP in a new way. You could bundle either of them as an extension. Essentially, anything that can make Firefox overlook a piece of native code and instead perform the instructions you provide would fall under this category. Did you know: The technology that powers your browser to play Flash animation was not made by Mozilla or Google (although they’re trying their hands at it), but by Adobe.

- **Themes**
  While themes need no introduction, the degree of customisation that’s on offer on Firefox is worthy of a mention. Themes are generally any extension that can change the look and feel of the browser, with the change being either miniscule or massive. With a theme, you can either just graze the surface and change things such as the browser’s tone and color, or you can delve deep into the UI, and go so far as to implement a completely new interface to work with. ‘Complete themes’ make deep changes to the browser’s UI and behaviour while changes in ‘regular themes’ are simply changes to the appearance.

Yup, still Firefox, in Vimperator’s interpretation
Blurring the lines between the three extensions

Now, for an example of the degree of control that’s allowed to a developer in the Firefox environment, we take a look at an extension that we mentioned earlier: ‘Vimperator’ – one which closely resembles the (much too) famous command line text editor, Vim. The concept behind the project is to provide a completely different experience to the user, in contrast to that provided by the vanilla Firefox installation. This means, every unnecessary element in the browser’s chrome (chrome is defined as the UI elements in the browser, covering everything but the title bar and the page view area) is removed, and you get to view your page in full glory making the most of all of the screen’s real estate. The only navigation and control section provided is a bar at the bottom of the screen which takes input in the form of pre-defined key bindings, many of which are replicated from Vim itself. While a regular user might find this daunting, the interface’s accurate reflection of the original editor is comforting to a vast majority of its users, and is therefore one of the most widely used add-ons for the browser. Just like its command line counterpart, Vimperator is widely appreciated for reducing dependence on a mouse, while focusing almost all the necessary actions literally on a user’s fingertips, via the key bindings.

Gecko versus WebKit

Now, with that knowledge let’s move forward. The next appropriate step would be to get a rough idea of the soul of the whole operation. We, therefore, look into the architecture of the browser engine under consideration – the Gecko engine. It’s widely acknowledged that the rendering engine is the component most critical in the experience provided by a browser. If you’ve been watching the development scene closely, you would have definitely heard about the current buzzword on the horizon, ‘WebKit’ – which powers Apple’s Safari browser and Google’s Chrome browser. Now for a
general idea, we narrow down the workings of Gecko and WebKit (as used by the Chromium base) to two distinct points – the process architecture and extensibility.

The Gecko process architecture is such that it handles all of the concurrent tabs on a single process thread. While this results in a comparatively slower interface, in the long term, it pays off to not fork a new process for every little task that needs to be accomplished – the model commonly used under the WebKit architecture. What this means is that while the first few tabs with WebKit would seem faster, it's because the engine is only utilising a few extra resources from the computer to parallelise the work and bring with it the overhead involved, but when you go up to double digits in tabs opened, with some of them doing resource-intensive work, such as playing videos or working with comprehensive web-apps, this model returns to cripple the overall performance of the system, and by extension, of the browser itself, due to the sheer number of the processes that are now being requested to be forked and maintained by the system. Firefox, as has often been cited, tends to do much better at handling such high pressure loads, because all of the browser’s workings are restricted to a single thread, which is much more maintainable at a system level. The difference in the memory... to this, at the click of a button
footprint increases almost exponentially between the two, as we make our workings more and more resource intensive.

Then, there’s the concern about online privacy. Although not directly relevant to the browser engines, it’s worth noting that due to Firefox not being tied into any particular ecosystem, it offers an unparalleled sense of security due to an absolute lack of conflict of interests. The ‘Firefox Sync’ feature remains one step ahead of the usual username/password combination, and provides an encryption key that is stored locally on your device, which, when passed on to a new installation of Firefox, provides your data to you remotely, via Firefox servers, with nobody else being able to peek at the exact information. Also, Firefox pioneers the campaign to hamper the ability of online commodities to track the actions of users via a combination of IP address re-routing, and blocking other user-identifiable data. This has been reflected in the aftermath of the recent NSA-spying revelations by Edward Snowden, with users responding by exponentially increasing their downloads of non-conglomerate entities such as Firefox and Opera.

Of course, there will be entities who believe that this can be used for illegal activities, but then again, so can a regular phone call.
It’s time to look under the hood to know how exactly an add-on works

If you have any experience with building things that are supposed to work independently, you’d know that some standard files need to be present for stuff to work. Firefox add-ons have these structures too, as given on the MDN (Mozilla Developer Network – the repository of almost everything that you need to know about development using any of Mozilla’s products) website:

my_extension.xpi: //Equal to a folder named my_extension/
/install.rdf //General information about your extension
/chrome.manifest //Registers your content with the Chrome engine
/chrome/
/chrome/content/ //Contents of your extension such as XUL and JavaScript files
/chrome/icons/default/* //Default Icons of the extension
/chrome/locale/* //Building an Extension# Localization
/defaults/preferences/*.js //Building an Extension# Defaults Files
/plugins/*
/components/*
/components/cmdline.js
All of this is then bundled into an xpi (we’re expecting you to use some sort of a *nix environment here) using the command ‘zip -r <your_extension_name_here>.xpi *’, which broken down, tells the computer to ‘zip’ ‘*’ (all) your files ‘-r’ecursively (from the current directory, inwards) into a file named ‘<your_extension_name_here>.xpi’.

Now let’s have a look at the files and examine the purpose that they’re trying to fulfil here.

**Install Manifest**
The install.rdf file is the Install Manifest for the extension, which is supposed to contain metadata about the add-on that you’re trying to build. Written in the XML format – which is about the only language that a normal person can understand as well as a computer – the install manifest declares everything about the code that you’re supposed to write. Here’s a sample RDF file:

```xml
<?xml version="1.0"?>
<RDF xmlns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```
Note that not all of these properties are required. The ‘won’t-build-if-absent’ properties include:

1. **id**: An identifier for the add-on, which can be a GUID or an email address formatted as a string
2. **version**: The version of the extension being supplied
3. **type**: The kind of code being supplied, with 2 representing an add-on, 4 for theme, 8 for a locale, 32 for a Multiple Package Item, and 64 for a spell-check dictionary
4. **targetApplication**: The application that you’re building the extension for, such as Firefox or ThunderBird
5. **name**: The name of the add-on

The rest are optional properties, but it’s advisable to supply as much information as you can, so as to make the code more accessible, and also to aid the process of review at AMO (addons.mozilla.org).
XUL

While we’ll be looking into XUL (XML User Interface Language) later in this Fast Track, for now, all we need to know is that it’s an XML-based grammar, which is what the Firefox user interface is written using. Think of it as HTML for UI. The developer is provided all necessary tools such as buttons, toolbars, etc., with JavaScript being used to fill in the actions. To provide functionality for our extension, we can modify the browser’s UI, and integrate our add-on into the interface that way. The browser is implemented in an XUL file called browser.xul, archived inside $FIREFOX_INSTALL/chrome/browser.jar, under /content/browser/browser.xul.

Chrome URIs

As we know, ‘chrome’ refers to everything in the browser that is not the title bar or the page-view area, so essentially the entire user interface. Chrome Packages are UI bundles which can be loaded via dedicated locations in the address bar, in the form “chrome://______”. This is useful because it creates a layer of abstraction that allows the developers to transcend the underlying architecture of the system (OS and such), which would pose a problem of consistency if hardcoded “file://______” links were to be used. To cut a long story short, we now have chrome URIs for all UI work. For a fun detour,
enter “chrome://browser/content/browser.xul” into the Firefox address bar, and see what happens (nothing bad, we promise!).

Chrome URIs consist of several components:

1. **The URI scheme (chrome):** which tells Firefox’s networking library that this is a Chrome URI. It indicates that the content of the URI should be handled as a chrome. Compare chrome to http which tells Firefox to treat the URI as a web page.

2. **A package name (in the example above, browser):** which identifies the bundle of user interface components. This should be as unique to your application as possible to avoid collisions between extensions.

3. **Type of data being requested:** There are three types: content (XUL, JavaScript, XBL bindings, etc. that form the structure and behavior of an application UI), locale (DTD, .properties files etc. that contain strings for the UI’s localisation), and skin (CSS and images that form the theme of the UI).

4. **The path of a file to load**

### Chrome manifest

**comments**

A line is a comment if it begins with the character ‘#’. Any other character in the line is ignored.

- # this line is a comment - you can put whatever you want here

**manifest**

- manifest subdirectory/foo.manifest [flags]

This will load a secondary manifest file. This can be useful for separating component and chrome registration instructions, or separate platform-specific registration data.

**binary-component**

- binary-component components/mycomponent.dll [flags]

Instructs Mozilla to register and use a binary component. It should be combined with the ABI flag, since binary components are ABI-specific. Prior to Firefox 4, files in the components directory were registered automatically.

**interfaces**

- interfaces component/mycomponent.xpt [flags]
Instructs Mozilla to load interface information from a typelib file produced by XPIDL. Prior to Firefox 4, files in the components directory were registered automatically.

**component**

```
component {00000000-0000-0000-0000-000000000000} components/mycomponent.js [flags]
```

Informs Mozilla about a component CID implemented by an XPCOM component implemented in JavaScript (or another scripting language, if applicable). The ClassID {0000...} must match the ClassID implemented by the component. To generate a unique ClassID, use a UUID generator program or site.

**contract**

```
contract @foobar/mycontract;1 {00000000-0000-0000-0000-000000000000} [flags]
```

Maps a contract ID (a readable string) to the ClassID for a specific implementation. Typically, a contract ID will be paired with a component entry immediately preceeding.

**category**

```
category category entry-name value [flags]
```

Registers an entry in the category manager. The specific format and meaning of category entries depend on the category.

**content**

A content package is registered with the line

```
content packagename uri/to/files/ [flags]
```

This will register a location to use when resolving the URI chrome://packagename/content/.... The URI may be absolute or relative to the location of the manifest file. Note: it must end with a ‘/’.

**locale**

A locale package is registered with the line

```
locale packagename localename uri/to/files/ [flags]
```

That’s about it, as far as the absolute essential (non-code) files go. You’re free to add many more, of course. For example, you may want your extension to be a restartless service, which means that the browser doesn’t need
to be restarted in order to use your code. In that case, you’ll have to add a file called bootstrap.js (because these extensions are called bootstrapped extensions). What exactly goes into that file? You’re going to have to find that out for yourself, among a few other related things, since Mozilla allows you to work with a plethora of options.

**Debugging extensions**
- The DOM Inspector will inspect attributes, DOM (Document Object Model, look it up) structure and CSS style rules that are in effect (e.g. find out why your style rules don’t seem to be working for an element – an invaluable tool!)

Bugs shall not pass!

- Use the good ol’ console.log(“string”) if you initiated your session from the terminal
- Use Components.utils.reportError() or nsIConsoleService to log on to the JavaScript console

Note that most of these are auto-generated when you use the Add-on Builder for Jetpack add-ons. And don’t worry if some of it isn’t there – the builder knows what it’s doing!
To know our future, we must know our past!

Around 10 years ago, somebody came up with a plan. A plan that a browser shouldn’t be limited to what the developer of the browser shipped it as – but what the users want it to be. That meant giving anyone the authority to make changes to the default form and functionality provided by the browser. Such power in the hands of the people who were probably not actively involved in the development of a browser was unheard of. Fast forward to today, and such practices are the staple of our internet experience. About 85% of browser users take advantage of extensions, millions of which have been created by non-Mozilla employees – ranging from tools as simple as ad-blockers to complex DOM investigation and network sniffing tools.

Of course, that ‘somebody’ who was responsible for this revolution was a figurative for Mozilla. XUL (pronounced “zool” and rhyming with “cool”) was developed by the organisation with the intent of providing tools for writing GUIs. Essentially a subset of XML, it was championed as a user-interface markup language, and is short for XML User-Interface Language.
The grammar obviously has its own namespace maintained by Mozilla. XUL is based entirely on open web standards such as CSS, JavaScript and DOM. This means that, similar to many of Mozilla’s pet technologies, it’s very easy to get involved in, if you’re familiar with those languages. Obviously, being designed as a Mozilla specific entity, XUL has no formal specification and doesn’t inter-operate with non-Gecko implementations of layout rendering. There are physical separations for the structure of add-ons that are built this way. The three segregations are:

- **content**
- **locale**
- **skin**

While the names are mostly self-explanatory, for the sake of detailed investigation of the platform, let’s go through them sequentially. Aside from the essential files that are needed to be contained in an add-on, an XUL-based add-on segregates these three classes of components into their namesake directories.

- **The content/ directory:** holds the browser overlay with XUL (the .xul file), along with the corresponding JavaScript file. As the name would suggest, this directory holds all the portions of the code that will constitute the end-result. In other words, any code that’s meant to perform tasks other than print text or apply a style to the interface that’s being provided in the adjoining XUL, goes into the JS file.

- **The locale/ directory:** holds the localisation modules based on different regions and their respective languages. This is obviously not an essential feature, but the fact that it’s supported is a great boost to the credibility and outreach of one of the largest platforms for application development. It’s designed to contain dumps of text segregated by languages, which can be used in the running of the code based on preferential inputs.

- **The skin/ directory:** holds the styling information for the add-on. All the CSS (Cascading Style Sheets) associated classes and files go into this directory, which can be accessed by the add-on when it’s loaded.

One important question that should pop up is: how exactly does the XUL used for defining add-ons interface with the default Firefox code? Well, there are well defined “merge points” for the two sets of XUL information, so to speak. For the default action, this is how things go down: In browser.xul we can find the status bar, which looks something like this:
So here, the statement `<statusbar id="status-bar">` represents a “merge point” for a XUL Overlay. Not exclusively, though, and not limited to the same. But it provides a starting point from which integration with the browser can begin.

**XUL elements**

While we’ve discussed XUL at length, we haven't talked about the innards of the XUL system. In order to understand this, we must first know the possibilities offered by the XUL namespace, which are commonly known as XUL elements. Let's have a brief look at the wide range of elements available.

**<button>**

A button that can be pressed by the user.

**<button type="menu">**

A button that has a dropdown menu attached to it. Pressing the button opens the menu.

**<button type="menu-button">**

A button that has a separate arrow button with a menu attached to it. Unlike with the ‘menu’ type, a separate action may be performed when the main part of the button is pressed.

**<checkbox>**

A control which can be turned on and off, typically used to create options which may be enabled or disabled.

**<colorpicker>**

A control that may be used to select a color.
<colorpicker type="button">
A specialised type of color picker which shows only a button but when
pressed, a popup will be displayed to select a color from.
</colorpicker>

<datepicker>
A set of text boxes which may be used to allow the entry of a date.
</datepicker>

<datepicker type="grid">
A datepicker which displays a calendar grid for selecting a date.
</datepicker>

<datepicker type="popup">
A datepicker which displays a set of text boxes for date entry, but also has
a button for displaying a popup calendar grid.
</datepicker>

<description>
The description element is used for descriptive text.
</description>

<groupbox>
A groupbox displays a labelled box around other user interface controls.
</groupbox>

<filefield>
Allows the user to select a file.
</filefield>

<image>
An image specified by a URL.
</image>

<label>
A label is used to create text which labels a nearby control.
</label>

<listbox>
The listbox is used to select an item from a list of labelled items.
</listbox>

<menulist>
A menulist (or combobox) is used to create a control with a dropdown to
select a value.
</menulist>

<menulist editable="true">
An editable menulist is like a standard menulist except that the selected
value is displayed in a textbox where it may be modified directly or values absent from the popup list may be entered.

**<progressmeter>**
A progress meter is used to display the progress of a lengthy task.

**<radio>**
A radio button is used when only one of a set of options may be selected at a time.

**<richlistbox>**
The richlistbox displays a list of items where one or more may selected. Unlike the listbox which is designed to display fixed size rows, the richlistbox may display any type of content.

**<scale>**
A scale displays a bar with a thumb that may be slid across the bar to select between a range of values.

**<splitter>**
Allows the user to adjust the division of space between elements.

**<tab>**
Description goes here.

**<textbox>**
A textbox which allows a single line of text to be entered.

**<textbox multiline="true">**
A textbox which allows multiple lines of text to be entered.

**<textbox type="autocomplete">**
A textbox which provides a dropdown showing matches that would complete what the user types. The user must select one of the options to have it filled into the textbox.

**<textbox type="number">**
A textbox for entering numbers. Two arrow buttons are displayed for cycling through values.
<textbox type="password">
A textbox that hides the characters typed; used for entering passwords.
</textbox>

<textbox type="search">
A textbox for searching.
</textbox>

<timepicker>
A timepicker displays a set of textboxes for entering a time.
</timepicker>

<toolbarbutton>
A button that is displayed on a toolbar.
</toolbarbutton>

<toolbarbutton type="menu">
A button that is displayed on a toolbar with a dropdown menu attached to it.
</toolbarbutton>

<toolbarbutton type="menu-button">
A button on a toolbar that has a separate arrow button with a menu attached to it. Unlike with the ‘menu’ type, a separate action may be performed when the main part of the button is pressed.
</toolbarbutton>

<tree>
A tree displays a hierarchy of items in multiple columns.
</tree>

Overall, the various groups that you have access to here are:

- **Top-level elements**: window, page, dialog, wizard, etc.
- **Widgets**: label, button, text box, list box, combo box, radio button, check box, tree, menu, toolbar, group box, tab box, colorpicker, spacer, splitter, etc.
- **Box model**: box, grid, stack, deck, etc.
- **Events and scripts**: script, command, key, broadcaster, observer, etc.
- **Data source**: template, rule, etc.
- **Others**: overlay, iframe, browser, editor, etc.

An important feature, which should be rather intuitive when you realise that XUL is basically just an XML dialect, is that one can use elements from other applications of XML within XUL documents, such as XHTML, SVG, and MathML. Another innovation in this regard is Mozilla’s addition of some common widgets with Gecko 1.9 such as `<scale/>` (sometimes called “slider”), `<textbox type="number"/>` (spinbox), time and date pickers. You can have the power of full-blown web apps and such, with all the benefits of local applications. Truly the best of both worlds and then some!
If you’ve ever been on Reddit, you’d know that this is the perfect time for… a plot twist. So, here it is – almost everything that you’ve learnt until now is closely related to XUL. How’s that? Well, all the manifest, and mostly all of the other specification files and declarations that we’ve been talking about are intrinsically tied to XUL in their understanding and implementation.

For quick reference (since we’ll need to remember this later) here are some key features and benefits of XUL.

<table>
<thead>
<tr>
<th>KEY FEATURES AND BENEFITS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Powerful widget-based markup language</strong></td>
<td>The goal of XUL is to build cross-platform applications, in contrast to DHTML which is intended for developing web pages. For this reason, XUL is oriented toward application artifacts such as windows, labels and buttons instead of pages, heading levels and hypertext links. In fact, many developers invest a significant amount of effort to achieve these results in their DHTML web applications but at the cost of complexity and performance and without any supporting standards.</td>
</tr>
<tr>
<td><strong>Based on existing standards</strong></td>
<td>XUL is an XML language based on W3C standard XML 1.0. Applications written in XUL are based on additional W3C standard technologies featuring HTML 4.0; Cascading Style Sheets (CSS) 1 and 2; Document Object Model (DOM) Levels 1 and 2; JavaScript 1.5, including ECMA-262 Edition 3 (ECMAScript) and XML 1.0. Mozilla.org is going a step further by seeking W3C standardisation for the eXtensible Binding Language (XBL).</td>
</tr>
<tr>
<td><strong>Platform portability</strong></td>
<td>Like HTML, XUL is designed to be platform-neutral, making applications easily portable to all of the operating systems on which Mozilla runs. Considering the broad range of platforms that currently support Mozilla, this may be one of the most compelling features of XUL as a technology for building applications. Since XUL provides an abstraction of user interface components, it delivers on the promise of “write once, run anywhere”. The user interface for all of Mozilla’s core applications (Browser, Messenger, Address Book, etc.) is written in XUL with one single code base supporting all Mozilla platforms.</td>
</tr>
</tbody>
</table>
| **Separation of presentation from application logic** | One of the major downfalls of most web applications is the tight coupling of user interface elements with client application logic. This poses a significant problem in team environments because the skills required to develop these two parts are often satisfied by different people.

XUL provides a clear separation between the client application definition and programmatic logic ("content" consisting of XUL, XBL and JavaScript), presentation ("skin" consisting of CSS and images) and language-specific text labels ("locale" consisting of DTDs and string bundles in .properties files). The layout and appearance of XUL applications can be altered independently of the application definition and logic. Further, the application can be localised for different languages and regions independently of its logic or presentation.

This degree of separation results in applications that are easier to maintain by programmers and readily customisable by designers and language translators. The work flow of these interdependent activities is more easily co-ordinated than with HTML-based web applications, with less impact on the overall stability and quality of the system. |
| **Easy customisation, localisation, or branding** | Another highly practical benefit of the separation that XUL provides among application logic, presentation and language text is the ease of customising for different customers or groups of users.

A developer can maintain one primary code base for his application and customise the logo and branding for each of his customers by supplying different skins. An application that’s written and deployed with an English language user interface can be translated to French for the same customer. While such changes are extensive and affect most (if not all) of the application, they’re also isolated from one another enabling the core XUL definition and application logic to be shared among all of the custom versions. |
Technologies suitable for XUL-based products

Let’s now take a look at some of the compatible technologies that have been created to complement the development of XUL-based products.

XBL
The eXtensible Bindings Language (XBL) is a markup language that defines special new elements, or “bindings” for XUL widgets. With XBL, developers can define new content for XUL widgets, add additional event handlers to a XUL widget, and add new interface properties and methods. Essentially, XBL empowers developers with the ability to extend XUL by customising existing tags and creating new tags of their own.

By using XBL, developers can easily build custom user interface widgets such as progress meters, fancy pop-up menus, and even toolbars and search forms. These custom components can then be used in XUL applications by specifying the custom tag and associated attributes.

Overlays
Overlays are XUL files used to describe extra content for the UI. They’re a general mechanism for adding UI for additional components, overriding small pieces of a XUL file without having to resupply the whole UI, and re-using particular pieces of the UI.

Overlays are a powerful mechanism for customising and extending existing applications because they work in two related but highly different ways. In one respect, overlays are synonymous with “include” files in other languages because an application may specify that an overlay be included in its definition. But overlays can also be specified externally, enabling the designer to superimpose them upon an application without changing the original source.

In practical terms, this enables developers to maintain one code stream for a given application, then apply custom branding or include special features for customers with a completely independent code base. This leads to an overall solution that’s easier and less costly to maintain in the long run.

There’s an additional benefit of overlays for software developers who intend to add features to Mozilla that they wish to keep proprietary. The Netscape Public License (NPL) and Mozilla Public License (MPL) require developers who alter original work (source code files that are provided with Mozilla) to release the source code for these changes to their customers.
Overlays can be used to add features to Mozilla without contaminating the original open source code with proprietary alterations.

**XPCOM/XPConnect**

XPCOM and XPConnect are complementary technologies that enable the integration of external libraries with XUL applications. XPCOM, which stands for Cross Platform Component Object Model, is a framework for writing cross-platform, modular software. XPCOM components can be written in C, C++, and JavaScript, and can be used from C, C++, JavaScript, Python, Java and Perl.

XPConnect is a technology which enables simple interoperation between XPCOM and JavaScript. XPConnect allows JavaScript objects to transparently access and manipulate XPCOM objects. It also enables JavaScript objects to present XPCOM-compliant interfaces to be called by XPCOM objects.

Together, XPCOM and XPConnect enable developers to create XUL applications that require the raw processing power of compiled languages (C/C++) or access to the underlying operating system.

**XPInstall**

XPInstall, Mozilla’s Cross Platform Install facility, provides a standard way of packaging XUL application components with an install script that Mozilla can download and execute.

XPInstall enables users to effortlessly install new XUL applications over the internet or from corporate intranet servers. To install a new application the user need only click a hypertext link on a web page or in an email message and accept the new package through a Mozilla install dialog.

While XUL was developed to aid the process of development at Mozilla, it has also been used beyond the realms of that single organisation, with many web apps, such as the Mozilla Amazon Browser, being developed in recent times. But, it has suffered due to the general indifference of the industry towards newer entrants in the technology, and also due to the distinct lack of standardisation. Now, armed with this extensive knowledge of XUL, let’s try our hand at developing.
A BOILERPLATE XUL ADD-ON

Should we put our knowledge to some use and create a template? Let’s do that!

The core purpose of this Fast Track is to acquaint the reader with what it means to write code for the Firefox platform, and how it could be done. With that in mind, let’s go ahead and test the waters, just to see what it feels like. We’ll work on a rather straightforward codebase.

What codebase is more straightforward than code that’s not supposed to do anything? No finding problems, or scrutinising for other specifics, and examining without the pressure of trying to figure out “how does that work?” Beginning with the understanding that programming is a tough task, it would probably be in your best interest if we started with something so ‘boilerplate’ that there’s almost nothing on the plate. That way, you’d be able to examine a template that, howsoever correct, is still easy enough to grasp the underlying architecture and design principles, while serving as a base for extensive revisits and eventually development in the future. However, that’s not something we’ll be doing in this particular booklet (not specifically XUL, that is).
Code, code everywhere

- `<code>`
- `<?xml version="1.0"?>`
This kind of file is called the Install Manifest, which, as we know, is the metadata holder for the packaged add-on. This RDF file, of course, could be a lot more (and, on the other end, a lot less) descriptive about the add-on, with a structure that we’ve already seen in the previous chapters. Still, with the file being a necessity for any architecture that builds into a Firefox extension, we declare it here, with some of the essential values.

Next up, we look at the required structure that the rest of the add-on needs to follow. An important file here is the Chrome Manifest (chrome.manifest). This file declares the chrome resources that the add-on will be using, and that’s what we see in action in the following piece of code.

```xml
<Description about="urn:mozilla:install-manifest">
  <em:id>testapp@watteimdocht.de</em:id>
  <em:name>Testapp</em:name>
  <em:description></em:description>
  <em:version>0.1</em:version>
  <em:creator>Fabian Franzen</em:creator>
  <em:homepageURL>http://www.watteimdocht.de/test</em:homepageURL>
  <em:type>2</em:type>
</Description>

<!-- Mozilla Firefox -->
<em:targetApplication>
  <Description>
    <em:id>{ec8030f7-c20a-464f-9b0e-13a3a9e97384}</em:id>
    <em:minVersion>4.0</em:minVersion>
  </Description>
</em:targetApplication>
</RDF>

<code>
content   xulschoolhello                content/
overlay chrome://browser/content/browser.xul  chrome://
</code>
xulschoolhello/content/browserOverlay.xul

•
•
•
</code>

Three lines of code in the whole file – that’s about as easy as it gets. However, this is more declarative than executing code, so it probably doesn’t count. Moving on, we examine a non-critical, but interesting part of the code – the locale. As discussed earlier, this is the language dump for localisation into regional languages for better outreach of the extension. Here, as an example, the structure that’s being used is root/locale/en-US/browserOverlay.properties.

•
•
•
<code>
•
•
•
xulschoolhello.greeting.label = Hi! How are you?

•
•
•
</code>

A corresponding controller for that record is the adjacent DTD (Document Type Definition file), which looks something like this:

•
•
•
<code>
•
•
•
<!ENTITY xulschoolhello.hello.label "Hello World!">
<!ENTITY xulschoolhello.helloMenu.accesskey "l">
<!ENTITY xulschoolhello.helloItem.accesskey "H">

•
•
•
</code>

If you look closely, these lines make ample sense for reading by a human (which, incidentally, is one of the great triumphs of XML and all of its descendents). It’s being defined that:

- hello.label property of the object xulschoolhello is being set to “Hello World!”
- helloMenu.accesskey property of xulschoolhello is being set to “l”
- helloItem.accesskey property of xulschoolhello is being set to “H”

It’s important to note that the DTD is an XML spinoff that was created as a set of markup declarations that define a document type for an SGML-
family markup language (SGML, XML, HTML) – a family to which XUL also belongs, due to being derived from XUL itself.

We then proceed to take a look at the JS that pairs with the core of our recent discussions, the XUL. In order to keep things simple, this line has been kept to a minimum, but so long as it doesn't violate an add-on's behavioural policies, any valid JavaScript file could be used here. The important consideration is that it should solve the purpose (implementing the logic behind the XUL file), and do so in a suitable manner.

```javascript
window.alert(message);
```

Hold your breath, because we're finally at the point when we should start looking at the famed XUL files. As you can see, they're valid XML files, which are dictated by the naming that Mozilla hosts at “http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul”.

```xml
<?xml version="1.0"?>
<overlay id="xulschoolhello-browser-overlay"
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
<script type="application/x-javascript"
src="chrome://xulschoolhello/content/browserOverlay.js" />
</overlay>
```

So there you have it. A piece of code that's absolutely correct, and is guaranteed to not work! It's not the hardest thing to do in the world to make something that actually works, but the steep learning curve that
stares newcomers in the eye discourages us from pursuing that route in our writing. Of course, that idea would be re-evaluated in the future, and we’ll see if we can bring out a more advanced tutorial.

Consider this as more of a template to show you how an add-on’s code is internally distributed, and how the different sections interact with each other and the browser itself.
A NEW WAY TO DEVELOP – THE JETPACK API

Learn from the past, look to the future!

By general estimations, about everybody is happy with their browser’s extensions. Of the billions that use browsers, 85% have some sort of an extension installed. Over the years, there’s been some criticism of the fact that a lot of background knowledge was required for developing what would eventually just be a new feature in the browser.

And so, Jetpack was born. The resounding success that it was, Firefox decided to ship it with the main browser as the official SDK for developing Firefox add-ons six months ago. Till date, while criticism has been directed at the fact that it’s still very limited in its capabilities, this will obviously improve as time goes by. For now, the attempt at standardising some of the more rogue elements of extension development is well appreciated.

Now, let’s examine Jetpack’s internals, within which a few segregations can be made. The top level difference is the depth of features. Jetpack offers both high-level and low-level APIs, and we’ll take this space to try and
A NEW WAY TO DEVELOP — THE JETPACK API

walk you through them. But first, for a general overview, you must know that they were designed keeping in mind that the developer will be using them to:

- create user interfaces
- interact with the web
- interact with the browser

High-Level APIs

addon-page
With the Add-on SDK you can present information to the user, such as a guide to using your add-on, in a browser tab. You can supply the content in an HTML file in your add-on’s “data” directory (we’ll get there, don’t worry). Note that this module has no effect on the Android browser, Fennec.

base64
The module provides data encoding and decoding using Base64 algorithms.

clipboard
The clipboard module allows callers to interact with the system clipboard, setting and retrieving its contents. You can optionally specify the type of data to set and retrieve. The following types are supported:

- text (plain text)
- html (a string of HTML)
- image (a base-64 encoded png)

context-menu
The context-menu API provides a simple, declarative way to add items to the page’s context menu. You can add items that perform an action when clicked, sub-menus, and menu separators. Instead of manually adding items when particular contexts occur and then removing them when those contexts go away, you bind items to contexts, and the adding and removing is automatically handled for you.

hotkeys
The hotkeys module enables add-on developers to define hotkey combi-
nations. To define a hotkey combination, create a Hotkey object, passing it the combination and a function to be called when the user presses that combination.

**indexed-db**
The indexed-db module exposes the IndexedDB API to add-ons. Scripts running in web pages can access IndexedDB via the window object. Note that it’s still in experimental stages of implementation.

**l10n**
The l10n module enables add-ons to localise strings appearing in the add-on’s JavaScript code. Note that you can’t currently use localise strings appearing in content scripts or HTML files.

**notifications**
The notifications module allows you to display transient, toaster-style desktop messages to the user.

**page-mod**
The page-mod module enables you to run scripts in the context of specific web pages. To use it, you specify:
- one or more scripts to attach. The SDK calls these scripts ‘content scripts’.
- a pattern that a page’s URL must match, in order for the script(s) to be attached to that page.

**page-worker**
The page-worker module provides a way to create a permanent, invisible page and access its DOM (Document Object Model). The module exports a constructor function ‘Page’, which constructs a new page worker. A page worker may be destroyed, after which its memory is freed, and you must create a new instance to load another page.

**panel**
This module exports a single constructor function ‘Panel’ which constructs a new panel. A panel is a dialog. Its content is specified as HTML and you can execute scripts in it, so the appearance and behaviour of the panel is limited only by what you can do using HTML, CSS and JavaScript.
**passwords**

The passwords module allows add-ons to interact with Firefox’s Password Manager to add, retrieve and remove stored credentials. A credential is the set of information a user supplies to authenticate herself with a service. Typically a credential consists of a username and a password. Using this module you can:

1. **Search** for credentials which have been stored in the Password Manager
2. **Store** credentials in the Password Manager
3. **Remove** stored credentials from the Password Manager

**private-browsing**

Per-window private browsing

The way Firefox works now is that private browsing status is a property of an individual browser window. The user enters private browsing by opening a new private browser window. When this is done, any existing non-private windows are kept open, so the user will typically have both private and non-private windows open at the same time.

Opting in to private browsing

Add-ons built using the SDK must opt in to private browsing by setting the following key in their package.json file:

```
  "permissions": {"private-browsing": true}
```

**querystring**

Module exports utility functions for working with query strings.

**request**

The request module lets you make simple yet powerful network requests.

**selection**

The selection module provides a means to get and set text and HTML selections in the current Firefox page. It can also observe new selections.

**self**

The self module provides access to data that is bundled with the add-on as a whole. It also provides access to the Program ID, a value which is unique for each add-on.
A New Way to Develop — The Jetpack API

simple-prefs
The simple-prefs module lets you store preferences across application restarts. You can store booleans, integers and string values, and users can configure these preferences in the Add-ons Manager.

simple-storage
The simple-storage module lets you easily and persistently store data across Firefox restarts. If you're familiar with DOM storage on the web, it's kind of like that, but for add-ons.

system
The system module enables an add-on to get information about the environment it's running in, access arguments passed to it via the cfx (a tool that we'll shortly get to know) --static-args option and quit the host application.

tabs
The tabs module provides easy access to tabs and tab-related events.

timers
The timers module provides access to web-like timing functionality.

url
The url module provides functionality for the parsing and retrieving of URLs.

widget
The widget module provides your add-on with a simple user interface that is consistent with other add-ons and blends in well with Firefox. Browser widgets are small pieces of content that live in the Firefox add-on bar. They can be simple icons or complex web pages. You can either attach panels to them that open when they’re clicked, or define a custom click handler to perform some other action, such as opening a web page in a tab.

windows
The windows module provides basic functions for working with browser windows. With this module, you can:
- enumerate the currently opened browser windows
- open new browser windows
- listen for common window events such as open and close
**Low-Level APIs**

These modules fall roughly into three categories:

- Fundamental utilities such as collection and URL. Many add-ons are likely to want to use modules from this category.
- Building blocks for higher level modules such as events, worker, and api-utils. You’re more likely to use these if you’re building your own modules that implement new APIs, thus extending the SDK itself.
- Privileged modules that expose powerful low-level capabilities such as window/utils and XHR.

These modules are still in active development, and we expect incompatible changes to be made to them in future releases. Note that we’ll not be including already deprecated APIs in this list, for the sake of future-proofing, to some degree.

/`loader`

The loader module allows for creating CommonJS module loaders. The code is intentionally authored so that it can be loaded in several ways, such as under a script tag, a JavaScript of CommonJS module.

`console/plain-text`

SDK add-ons can log debug messages using the global console object, and the plain-text-console module implements this object.

`console/traceback`

The traceback module contains functionality similar to Python’s traceback module, which means you’re given the full stack trace in the event of code failure.

`content/content`

The content module re-exports three objects from three other modules: Loader, Worker and Symbiont. These objects are used in the internal implementations of SDK modules which use content scripts to interact with web content such as the panel or page-mod modules.

`content/loader`

The loader module provides one of the building blocks for those modules in the SDK which use content scripts to interact with web content such as the panel and page-mod modules.
content/mod
The mod module provides functions to modify a page content.

current/symbiont
The symbiont module exports the Symbiont trait, which is used in the internal implementation of SDK modules such as panel and page-worker, that can load web content and attach content scripts to it.

current/worker
This module is used in the internal implementation of SDK modules which use content scripts to interact with web content such as the tabs, panel, or page-mod modules.

core/heritage
Doing inheritance in JavaScript is both, verbose and painful. Reading or writing such code requires a sharp eye and lots of discipline, mainly due to code fragmentation and lots of machinery being exposed, which is why this module provides a minute degree of control over the same.

core/namespace
Provides an API for creating namespaces for any given objects, which effectively may be used for creating fields that are not part of objects’ public API.

core/promise
With most JS APIs being asynchronous, callbacks are the law of the land, but are rather difficult to maintain. Now, consider the scenario where instead of continuation passing via callback, function returns an object, that represents eventual result, either successful or failed. This object is a promise, both figuratively and by name, to eventually resolve. We can call a function on the promise to observe either its fulfillment or rejection. If the promise is rejected and the rejection is not explicitly observed, any derived promises will be implicitly rejected for the same reason. This functionality is provided by the module core/promise.

event/core
Many modules in the SDK can broadcast events. For example, the tabs module emits an open event when a new tab is opened. The event/core module enables you to create APIs that broadcast events. Users of your
API can listen to the events using the standard on() and once() functions.

**event/target**
Many objects in the SDK can broadcast events. For example, a panel instance emits an show event when the panel is shown. The event/target module enables you to create objects that broadcast events. Users of the object can listen to the events using the standard on() and once() functions.

**frame/hidden-frame**
The hidden-frame module creates Firefox frames (i.e. XUL <iframe> elements) that are not displayed to the user. It’s useful in the construction of APIs such as page-worker that load web content not intended to be directly seen or accessed by users. It is also useful in the construction of APIs such as panel that load web content for intermittent display.

**frame/utils**
The frame/utils module provides helper functions for working with platform internals such as frames and browsers.

**io/byte-streams**
The byte-streams module provides streams for reading and writing bytes.

**io/file**
The file module provides access to the local filesystem.

**io/text-streams**
The text-streams module provides streams for reading and writing text using particular character encodings.

**lang/functional**
The lang/functional module provides functional helper methods.

**lang/type**
Provides simple helper functions for working with type detection.

**loader/cuddlefish**
cuddlefish is the name of the SDK’s module loader. This is used when you use the cfx tool to run an intermediate build of your add-on.
**loader/sandbox**  
Provides an API for creating JavaScript sandboxes and executing scripts in them.

**net/url**  
The net/url module enables you to read content from a URI.

**net/xhr**  
Provides access to XML ‘Http Request’ functionality, also known as AJAX.

**page-mod/match-pattern**  
The match-pattern module can be used to test strings containing URLs against simple patterns.

**places/favicon**  
Provides simple helper functions for working with favicons.

**platform/xpcom**  
If all you need to do is use XPCOM objects that someone else has implemented, then you don’t need to use this module. You can just use require(“chrome”) to get direct access to the Components object, and access XPCOM objects from there. The xpcom module makes it simpler to perform three main tasks:
- Implement XPCOM object interfaces
- Implement and register XPCOM factories
- Implement and register XPCOM services

**preferences/service**  
The preferences/service module provides access to the application-wide preferences service singleton. To define preferences for your own add-on and expose them to the user in the Add-on Manager, you can use the simple-prefs module.

**stylesheet/style**  
Module provides Style function that can be used to construct content style modification via stylesheet files or CSS rules.

**stylesheet/utils**  
Module provides helper functions for working with stylesheets.
**system/environment**
Module provides API to access, set and unset environment variables via exported env object.

**system/events**
Module provides core (low level) API for working with the application observer service, also known as nsIObserverService.

**system/runtime**
The runtime module provides access to information about Firefox’s runtime environment. All properties exposed are read-only.

**system/unload**
The unload module allows modules to register callbacks that are called when they’re unloaded.

**system/xul-app**
The xul-app module provides facilities for introspecting the application on which your program is running.

**tabs/utils**
The tabs/utils module contains low-level functions for working with XUL tabs and the XUL tabbrowser object.

**test/assert**
The assert module implements the assert interface defined in the CommonJS Unit Testing specification.

**test/harness**
This module contains the bulk of the test harness setup and execution implementation.

**test/httpd**
Provides an HTTP server written in JavaScript for the Mozilla platform, which can be used in unit tests.

**test/runner**
This module contains the package’s main program, which does a bit
of high-level setup and then delegates test finding and running to the harness module.

**util/array**
The util/array module provides simple helper functions for working with arrays.

**util/collection**
The collection module provides a simple list-like class and utilities for using it. A collection is ordered, like an array, but its items are unique, like a set.

**util/deprecate**
The deprecate module provides helper functions to deprecate code.

**util/list**
The “list” module provides base-building blocks for composing lists.

**util/object**
The util/object module provides simple helper functions for working with objects.

**util/uuid**
The uuid module provides a low-level API for generating or parsing UUIDs. It exports a single function, uuid().

**window/utils**
The window/utils module provides helper functions for working with application windows.

**The CFX tool**
The exact specifics of the CFX tool, which is pretty much at the heart of the Jetpack initiative, is a little beyond the scope of this booklet, but what we'll do is give you an idea of what you can strive to accomplish with it. The most important commands associated with this tool are: cfx init, cfx run and cfx xpi.

**cfx init**
This command is used to initialise an empty directory in order to house a new add-on codebase.
cfx run
This command is used to test an intermediate build. It will temporarily compile and install your add-on into a vanilla installation of Firefox for testing purposes. Note that there would be no trace of either that session or the packaged add-on once that window is closed or the command is stopped.

cfx xpi
This is the command that is used to build and store tangible packaged add-ons from an add-on’s codebase, which will then be directly readable with any session of the browser, for a genuine installation.
I hear and I forget; I see and I remember; I do and I understand.

If you’ve given the table of contents for this Fast Track a quick glance, you’d know that we’ll be shortly developing a comprehensive add-on in this booklet. But before we do that, let’s test the waters on a much smaller scale.

We’ll do that with one of the more popular and rather nifty add-ons available on AMO, Aapptabs. Aapptabs was released under MPL 1.1, and developed and maintained by Felipe Gomes. A copy of the license for informational and/or legal purposes can be found at http://www.mozilla.org/MPL/1.1/.

Now, first things first – we need to set up the development environment. Go to Mozilla’s add-on development resource website and download the latest version of the Add-on SDK, and place it at a convenient location. Obvi-
iously, extract the contents of the archive if you downloaded it in that form. Now, change into the that folder, and do 'source bin/activate', which would load all the environments into your session. Now, you can utilise the real power of the SDK – the 'cfx' tool. Also, you'd see (addon-sdk-<version>) before your command prompt. This means that we're now ready for some action.

Proceed to create a folder with your extension's name, and give yourself a pat on the back – you're one step away from making a real world contribution to the software domain. Alright then, now, change into your shiny new folder, and run 'cfx init'. That's the instruction given to the cfx tool to initialise the folder as the designated ground for erecting a new add-on. Once that's done, you'll see some barebone files just waiting for you to bring them to life. We’ll start with the ‘package.json’ file, which is the clearest way of declaring your intent. In the aforementioned Aaapptabs add-on, the following is what the package.json file looks like. The package.json file is about as simple a piece as we’re going to encounter. The bunch of JSON put together imparts values into an object that defines metadata for the extension. Let’s have a look at what that looks like:

```json
{
}
```
Simple enough, right? The only nondescript part is the ugly-looking code under the ‘id’ property. But then what’s the SDK for if not to protect you from the ugliness? It does its job well, supplying that value (which by the way, is a Firefox identifier) beforehand, at initialisation. Leave a file that looks something like this, at the root of your extension’s directory.

Now, we move on to the first point of contact between the browser and the add-on in the Jetpack scenario, the ‘main.js’ file. Located inside the root/lib folder, this is where the execution starts, and continues unless you make an effort to include some other elements of code in the process. Mainly a standard JavaScript file, let’s look at a sample of what it should look like. Here’s a snippet of code from the add-on that we’re examining at the moment:

```
const wu = require("window-utils");
let mappings = [];

function start(chromeWindow) {
  let XULBrowserWindow = chromeWindow.XULBrowserWindow;
  if (XULBrowserWindow == undefined) {
    return;
  }

  let originalFunction = XULBrowserWindow.hideChromeForLocation;
```
mappings.push({window: chromeWindow, originalFunction: originalFunction});

XULBrowserWindow.hideChromeForLocation = function(aLocation) {
  return originalFunction.call(XULBrowserWindow, aLocation) ||
  chromeWindow.gBrowser.selectedTab.pinned;
}

chromeWindow.document.addEventListener("TabPinned", triggerOnLocationChange, false);
chromeWindow.document.addEventListener("TabUnpinned", triggerOnLocationChange, false);
}

function stop(chromeWindow) {
  for (let i in mappings) {
    let pair = mappings[i];
    if (chromeWindow === pair.window) {
      chromeWindow.XULBrowserWindow.hideChromeForLocation = pair.originalFunction;
      chromeWindow.document.removeEventListener("TabPinned", triggerOnLocationChange, false);
      chromeWindow.document.removeEventListener("TabUnpinned", triggerOnLocationChange, false);
      mappings.splice(i,1);
      break;
    }
  }
}
</code>

What, what? Did we just see XUL there somewhere? Well, yes. As we’ll examine in a minute, XUL and Jetpack aren’t so much rival models as they’re complementary. Let’s see how. First, we notice that ‘wu’ is taken as a holder for ‘window-util’ API from Jetpack. Then, a couple of functions are defined, which would expect windows in their arguments that start and stop the effects of the add-on. Their purpose will become clearer when we look at the rest of the code.
A Boilerplate Jetpack Add-on

function triggerOnLocationChange(event) {
  let win = event.target.ownerDocument.defaultView;
  win.XULBrowserWindow.onLocationChange(
    {DOMWindow: win.content}, /* stub aWebProgress obj */
    null,
    win.gBrowser.selectedTab.location
  );
}

var delegate = {
  onTrack: function (window) start(window),
  onUntrack: function (window) stop(window)
}

new wu.WindowTracker(delegate);

The code is pretty and functional as well. We see that WindowTracker is used from the window-utils API, and that we use the ‘start’ function defined earlier for tracked entities and ‘stop’ for untracked, and an event listener is applied to track any changes in state. The event listener would report any changes that need to be made (whether or not the chrome has to be altered) when the selected tab is changed.

Aapptabs in action. Neat!

The purpose of this add-on is removing the names of tabs, which hold apps, thus giving you more screen real estate with your currently open tab. Of course, you’ll have to use this feature to appreciate it.

Now, your code should ideally be different, one that implements your own idea. So have fun creating your own!
The Case for Choosing Jetpack Over XUL

When you have to pick sides, make sure you pick the right one

Now that you know of the ways by which you can develop for Firefox, it’s probably time for you to play favorites. And we should probably pronounce our verdict about the better option. Without further ado, we think it’s Jetpack.

XUL and Jetpack aren’t exactly in direct competition here. XUL, as we know, is not a standalone way to develop for Firefox – it simply defines the UI, for which you then proceed to fill up the JavaScript. XUL is to a browser what HTML is to a webpage.

As we’ve already said, the average Jetpack add-on is not really in direct competition with the XUL / JS pairing. Look at it this way: XUL tells you what you can’t do, giving you the ability to do anything other than what was explicitly denied. Jetpack, on the other hand, tells you what you can do, and you can’t do anything else. That, of course, extends from the fact
The Case for Choosing Jetpack over XUL

that Jetpack is, in fact a set of APIs, and this is just a part of its function. So, to summarise, we can say that both methods have their own merits and downfall; but we can safely say that if you were to start today, Jetpack would be easier to understand.

So, having agreed that both sides hold merits, it would only be fair to get on to examining them. First up, the XUL side of the argument:

As we’ve discussed time and again, a browser written in XUL offers a level of functionality and deep integration with the environment that’s nigh impossible to match, whatsoever the opposition. This is obviously a huge plus point. Then, with XPCOM, a vast number of possibilities have been exposed and used by developers over the years. XPCOM stands for Cross Platform Component Object Model, and it allowed for a great deal of clarity and consistency in Mozilla’s products. Also, it dealt with intricacies throughout the architecture, so the way it empowered a developer was unmatched. But the simple fact of the matter is that most developers don’t actually need that kind of power with the tasks that they’re doing. If we had to cook up a game analogy, it would be like using a sniper rifle in a head-on battle – powerful, but possibly detrimental, and certainly overkill. This was, undoubtedly, one of the premier motivations for starting the Jetpack project.

<table>
<thead>
<tr>
<th>User interface flexibility</th>
<th>XUL overlays offer a great deal of options for building a UI and integrating it into the browser. Using only the SDK’s supported APIs you have much more limited options for your UI.</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPCOM</td>
<td>Traditional add-ons have access to a vast amount of Firefox functionality via XPCOM. The SDK’s supported APIs expose a relatively small set of this functionality.</td>
</tr>
</tbody>
</table>

If we just have to have a winner...
Which brings us to the benefits of, well, the Jetpack project. As you’ll notice by the size of the table that follows, there are many. The most striking part is the platform-agnostic nature that you get when you use it. That means, you can develop for all Firefox platforms, desktop and mobile, which is a big plus in that the developer need not know the intricacies of the platforms, but only the kind of APIs available for that particular target build. This layer of abstraction is in very active development at the moment, and is only expected to get better as it moves into the later stages of being a complete product. Of course, we’ve talked at length about how simple (and most importantly, centralised) it seems, in comparison to other ways of add-on development. This streamlining is very important to have the developers constantly engaged and interested in the platform, not that Mozilla has to worry about the opposition any time soon.

Another very important benefit that standardisation of the development process does is that it enforces rather than allows the use of best practices on development. It’s not entirely impossible to go wrong with the SDK – that would be a fatal underestimation of well, just how wrong one can go when playing with computer bits. But, still, you’re going to have to try hard to do that. Of course, because the entire interaction is with APIs, better security could be maintained, which previously integrated add-ons have sometimes been accused of compromising on. No such worries here, as only the essentials are exposed. And, with restartlessness built in, it eliminates some degree of work that needed to be done by the user in order to take his newest toy for a drive. All in all, the experience is more wholesome, and the process is beneficial for both, external developers and the Firefox team, while the users get access to the same innovative extensions that have always set the bar in terms of delivering the best browser experience.

But then, in our humble opinion, Jetpack is what add-on development should be. For years, if there’s one thing you could accuse Firefox of, it would be the assumption that for developing for Firefox you needed plenty of
### The Case for Choosing Jetpack over XUL

| Simplicity | The SDK provides high-level JavaScript APIs to simplify many common tasks in add-on development, and tool support which greatly simplifies the process of developing, testing and packaging an add-on. |
| Compatibility | Although there’s no promise of the kind, maintaining compatibility for the High-Level APIs across Firefox versions is a top priority. For now, the APIs are be forward-compatible with the new multiple process architecture (codenamed ‘Electrolysis’) planned for Firefox. |
| Security | If they’re not carefully designed, Firefox add-ons can open the browser to attack by malicious web pages. Although it’s possible to write non-secure add-ons using the SDK, it’s not as easy, and the damage that a compromised add-on can do is usually more limited. |
| Restartlessness | Add-ons built with the SDK can be installed without having to restart Firefox. Although you can write traditional add-ons that are restartless, you can’t use XUL overlays in them, so most traditional add-ons would have to be substantially rewritten anyway. |
| User Experience | The UI components available in the SDK are designed to align with the usability guidelines for Firefox, giving your users a better, more consistent experience. |
| Best Practices | Starting from SDK 1.5, Mozilla has added experimental support for developing add-ons on the new native version of Firefox Mobile. |

knowledge of almost unnecessary sections of the code-base, which are not necessarily essential to the final purpose. The biggest advantage is that it saves developers from the deep, dark corners of coding. The layer of abstraction is such that the SDK will communicate with the developer and the add-on, and will convey the sentiment of the code to the Firefox engine. Now, with Firefox being in such active development, it’s entirely plausible that at some point, some of the code that
you wrote to interface with a specific part of Firefox might change over time, and that would have previously left you out in the cold. You would have had to closely look into exactly what changes were made, and take steps at a personal level to ensure that your code continue to be in accordance with the platform. The Jetpack SDK promises to expose consistent APIs to the developers, which means that the entire task of having to maintain a working interface with prone-to-change browser code is not your responsibility anymore. Cheers! 🍻
It’s time to take those training wheels off!

We guess by now, you’ve developed a good understanding of the Firefox add-on world. Now it’s time to put this knowledge to use, and create something tangible. We’ll now proceed to create an interactive dashboard which will represent the memory consumption statistics of the browser, and the elements running inside it.

In the interest of full disclosure, this piece of code has already been released. It was Fast Track writer and creator of this add-on, Abhishek Choudhary’s Google Summer of Code project with Mozilla. It’s released under the Open Source Mozilla Public License, so feel free to tinker around with the code on GitHub at https://github.com/abbisbekchoudhary/fx-statistics

For this process, we’re assuming that you’re working on a *nix-based system, because we can never really start from scratch. So, create a folder of your choice, and go ahead and run cfx in it to kickstart our add-on into existence.

You’ll now find that a few files have been created. As much as we’d like to walk you through the whole development process, space constraints...
prevent us from doing that. So here’s the target version of the file that will be our package.json.

```json
{
  "name": "fx-stats",
  "fullName": "fx-statistics",
  "dependencies": ["about", "addon-sdk", "toolbarbutton"],
  "id": "jid1-jnK9rw3l4bxjZQ",
  "description": "Browser statistics that make sense.",
  "permissions": {"private-browsing": true},
  "author": "Abhishek Choudhary",
  "license": "MPL 2.0",
  "version": "1.0.1"
}
```

As you can see, this is regular JSON (JavaScript Object Notation) here. If you don’t know much about JSON, we recommend learning about it first – it will serve you loyally for the rest of your existence. Seriously, go ahead and do that.
Great, you’re back. So, as you can see, these are just some properties that we’re now setting the values for. The ‘id’ attribute contains a string that tells the add-on suite that you’re developing the add-on for Firefox, as opposed to ThunderBird, or another Mozilla product. Also, rest easy in the knowledge that it’s provided by default by the builder when you run cfx init.

The rest of the attributes are more easily understandable, with the first two declaring the ‘name’ and ‘fullName’ values for the add-on. We then proceed to tell the builder the package dependencies to compile for our add-on. Don’t worry about the contents of that array – we’ll get to it later. Then, we’ll provide a small description of the add-on that we’re building. For now, just make a mental note of what’s going on. All of that is then followed by the author’s name, the license under which the code is to be released and the current development version. In order to have the add-on working, it should be allowed to be executed in the ‘private browsing’ mode as well.

The version number is decided by semantic versioning. To explain this simply, it’s done using the x.y.z structure. Here, x represents the instance when you make absolutely incompatible changes from your previous version, and/or have created an experience significantly different (hopefully for the better) from the last one. Now, y, one the other hand, is for the minor revisions that are made over the due course of a release. Lastly, z stands for the patches that you might need to release in order to fix bugs that inevitably land in any piece of software that is more than 100 lines in extent.

Now, let’s take a look at possibly the single largest file that we’ll encounter in the course of this exercise. The chrome.manifest file, which is about...one line in its entirety. Hence, gotcha!

```xml
<code>
content fx-statistics data/
</code>

Let’s examine what this behemoth is trying to accomplish. Now, add-ons work on a registry basis, which means that they need to be acknowledged by the underlying engine for an action to take effect. After this acknowledgement is given, an add-on can be made available for access (in addition to whatever other mode you’ve chosen, because it’s generally bad practice to direct the user to a dedicated URL for using your extension). This is where
we specify to the registry that the ‘content’ for the add-on called ‘fx-statistics’ is under the ‘data/’ folder (that we’ll shortly create).

Okay then, moving on. If you ran the ‘cfx run’ command the correct way, you’ll see three folders that are already present, namely ‘doc/’, ‘lib/’, and ‘test/’. For now, we don’t need to work with doc/, since we don’t really need to have any sort of documentation for now – what we’re doing is rather easy and (will be) quite understandable. The test/ folder is a whole different can of worms, so we’ll skip that one as well. Don’t worry, these are not critical to our purpose.

Our first interaction will be with the lib/ folder. It is important to note, again, that lib/main.js is the first point of execution for the compiler when it tries to pack our add-on into a Firefox readable file. So, with that in mind, let’s now look at what main.js should look like in our build. We’ll examine it in parts.

```javascript
const {Cc, Ci, Cu, CC, components} = require("chrome"),
{data} = require("sdk/self"),
tabs = require("tabs"),
BrandStringBundle = require("app-strings").StringBundle("chrome://branding/locale/brand.properties");

const debugging = true,
aboutStatsUrl = "about:stats",
statsTagLine = "Get To Know Your Browser",
brandShortName = BrandStringBundle.get("brandShortName"),
statsFullName = brandShortName + " Statistics";

var gMgr = Cc["@mozilla.org/memory-reporter-manager;1"].(getService(Ci.nsIMemoryReporterManager), // Master memory-reporter object

memArray = [],
// Object to hold data from single reporters

memTotal,
memCollector = {};
```
First, let’s declare a few variables that we’d need to use over the course of building this add-on. We’ll set up holders which will allow us to access the chrome’s properties. For those of you who are doing this for the first time, ‘chrome’ here refers to all the elements in the browser’s visible area, that is the user interface.

Then, we’ll create accessors for the ‘tabs’ and data functions we’ll make use of. While the ‘data’ accessor is used to maintain a link with the data/folder that we had previously created, the ‘tabs’ accessor will give us the ability to manipulate tabs in a browser, both actively and passively. The brandStringBundle gives the ability to sense which browser we’re running the code in. Among many possible release versions of Firefox, this includes Firefox Nightly and Firefox Aurora. Then, we’ll set up a few other variables that would be important in the upcoming lines.

```javascript
let toolbarButton = require('toolbarbutton').ToolbarButton({
  id: "stats-toolbar-button",
  label: "How we doin'?",
  tooltips: statsFullName,
  image: data.url("chart-icon.png"),
  onCommand: function () {
    let aboutStatsAlreadyOpen = false;
    // Check to see if aboutStatsUrl is already open in one of the tabs
    for each (let tab in tabs) {
      if (tab.url.toLowerCase() == aboutStatsUrl) {
        aboutStatsAlreadyOpen = true;
      }
    }
  }
});
```
Congratulations! You’ve created your first serious piece of code. What we have here is a toolbar button, which will be a part of our browser’s interface. It will serve as a shortcut button to our memory reporting page. You’ll notice an instruction in the code that checks if our page is already open in one of the tabs. If it is, we switch to that tab, otherwise, we open it in a new one.

Now, let’s create the function that will do the real work that needs to be done here – fetching the data that we can then work with.

```javascript
function processMemoryReporters() {
    dataArray = [];
    let e = gMgr.enumerateReporters();
    while (e.hasMoreElements()) {
        let rOrig = e.getNext().QueryInterface(Ci.nsIMemoryReporter);
        dataArray.push(JSON.parse(JSON.stringify(rOrig)));
    }
    transmission["memdata"] = dataArray;
    tabArray = [];
    let f = gMgr.enumerateMultiReporters();
    while (f.hasMoreElements()) {
        var mr = f.getNext().QueryInterface(Ci.nsIMemoryMultiReporter);
    }
}
```
Now, as a broad overview of what we're trying to accomplish here, let's go through the code. After the reporters have been enumerated, we query all the values in the single reporters, and add that as the general 'mem-data' into our (will be) carrier object / has / dictionary. Then, we empty the intermediate variable, and do the same for multi-reporters. Note that the single and multi-reporters are structurally different, and therefore the multi-reporters are queried and calculated in a manner that is very different from the way we access the single reporters. Also, we focus only the window-objects in the multi-reporters and nothing else, because that's where the tab data is. The rest of the content of the multi-reporters is not our primary concern at this point.

```javascript
function fixTab(t){
  var a;
  for each (let tab in tabs) {
    let tabInfo = t.split(/, /);
    log(tabInfo[0]);
  }
}</code>
```
It's important to note that the data given out by the multi-reporters is structurally different from what we'd expect, with the URLs consisting of backward slashes rather than forward, among other things. Now, in computers, backward slashes are generally used to escape the following character or to perform a special function. But we obviously don't want that to be happening with our data, so we proceed to fix the situation by some minor string processing to bring this data in line with our expectations, and much closer to what we'd have pulled from the single reporters. Also, the end-user would probably not be very interested in the URL of the tabs in our page, but rather the page title. Therefore, we replace the instances of URL with the tab title wherever we can. This attempt would fail if we were dealing with some internal Firefox page instances that we don't necessarily see in the interface as a user.
var regExp = aUnsafePath.match(/top\(([^)]+)\)/);

if (regExp) {
    var tabUrl = regExp[1];
    if (tabCollector[tabUrl]) {
        tabCollector[tabUrl] += aAmount;
    } else {
        tabCollector[tabUrl] = aAmount;
    }
}

</code>

This is how we pull the data out of multi-reporters. This method is invoked via callback, and is used to pull out the relevant data when the reporters are ready to interact post enumeration.

We use regular expressions because such is the structure of the memory reporting tree in Firefox that URLs and all information relevant to them are placed inside the name of the URL, preceded by the word ‘top’. This is unimportant at the moment. We just need to identify the important blocks of data and take those to form a part of our ‘tabdata’ entry in the carrier, ‘transmission’.

<code>

function onStatsPageOpened(tab) {
    log("Stats Page Opened");

    let styleCss = data.url("stats.css");
    tab.attach({
        contentScriptFile: data.url("stats.js"),
        contentScript: "populate(’’ + escape(styleCss) + ‘’,’’ + escape(statsFullName) + ‘’,’’ + escape(statsTagLine) + ‘’’);’’,
    });
    var worker = tab.attach({
</code>
contentScriptFile: [data.url("d3.v3.js"), data.url("visualizr.js")],

processMemoryReporters();

// Collect all necessary reports
worker.postMessage(transmission);
// Reports collected, send to Visualizr module

function loadStatsPage() {
  tabs.open({
    url: aboutStatsUrl,
    onReady: log("Via loadStatsPage.") // We don’t need to actually *do* anything, statsOnDemand will be invoked by the URL
  });
}

function statsOnDemand(tab) {
  if (tab.url.toLowerCase() == aboutStatsUrl) {
    log("Via statsOnDemand");
    onStatsPageOpened(tab);
  }
}

Here we’ve defined the handlers for the page itself. Once we realise that our page has, in fact, been opened, we would then need to attach the required styling (CSS) file, which is done by supplying the location and information of the said file to another attacher script, which would then bind the styling file to the page, and allow us to proceed with enhancing the page.

Next, we bind the library that we need to be using for the project – the excellent D3 library by Mike Bostock. Remember to have this file in your data folder when you’re preparing to compile the xpi, because the add-on needs the code to be stored locally and not via an online reference in the HTML page that we’ll shortly proceed to develop.

Now, with all the prerequisites ready, we can now invoke the main controller script that will handle the page and all of its associated information for us. Note that this is where we’ll eventually get that script attached to
the page, and then send our carrier (the ‘transmission’ object) to the script which will then work its D3 magic and bring about beautiful visualisation for the rather drab numbers.

Then, we have a simple worker function, loadStatsPage, which, when invoked, opens a new instance of the page in a new tab in the browser. Super simple stuff. Note that this worker doesn’t actually do anything to open the page, except supply the URL that we want to access. When the listener for the address bar sees this, it automatically does the rest and prepares the page as needed.

Then, there is the preparatory function that takes care of populating the new page with the section that we’d like and are working towards creating. It assumes that the page that it is working on is already a blank page that is on the target URL.

Here is the listener that we were talking about, which examines every input in the address bar, to see if we need to load our page there:

```
    tabs.on('ready', statsOnDemand);
```

Finally, we’d like to have close control over the workings of the page. For this, we’d need to take a focussed look at all the errors, if any, the code generates. If not, we need to examine that condition by witnessing a statement of success. For this purpose, we create a try-catch block and place the requisite code here.

```
    try {
      require('about').add({what: 'stats', url: data.url("stats.html")});
      toolbarButton.moveTo({
        toolbarID: "nav-bar",
        forceMove: true
      });
    }
```
As you can see, here we try to register the special page, and bind it to the boilerplate HTML that we’ll then proceed to populate with our prepared styling and handler scripts.

Next, we should prepare the packages that made it possible to register the page, and add things like the toolbar button that was a breeze to develop. Create a folder named ‘packages’ and add the following packages into it – ‘vold-utils’ and ‘toolbarbutton’. Note that you can simply take the dump of the code from GitHub and place it into the folder, and the builder will do all of the deciphering and linking for you.

Once that’s done, we can now create the holder HTML that we promised to populate.

```html
<!DOCTYPE html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" id="stat-stylesheet"/>
    <title></title>
</head>
<body link="#000000">
    <div id="wrapper">
    </div>
    <div id="finisher">
    </div>
</body>
```

This is just, very plain HTML with absolutely nothing fancy. We have
a couple of div buckets, whose purpose will become clearer when we get to the handler script. For now, this is as simple as HTML gets.

Similarly, for the CSS, we have a very simple structure that works with this thread-bare architecture that we have going on here.

```<code>
body {
  background: #C0C0C0;
  font: message-box;
  padding: 0 2em;
  margin: 0;
  min-width: 45em;
  margin: auto;
}

div.tooltip {
  position: absolute;
  text-align: center;
  height: 30px;
  padding: 8px;
  font: 10px sans-serif;
  background: #ddd;
  border: solid 1px #aaa;
  border-radius: 8px;
  pointer-events: none;
}
</code>
```

The body styling is very straightforward, with only the essentials such as background and font included, along with margin and a few other attributes. What’s a little more interesting is that we have a mouse-over tooltip that’s supposed to go with the add-on and that has a very detailed CSS style. A quick CSS overview should tell you easily what any of those attributes mean, although the naming is pretty self-explanatory.

Next up is the 'populate' function that we’d used to bind the styling and other attributes with the page in the main script. This function is defined in a separate file (which, if you remember, we also attached in our process), called 'stats.js'.
<code>

/* This contains the function that will prepare the HTML page with some of
* the cosmetics that we will not look at through the 'lib/main.js' or the
* 'data/visualizr.js' module. Currently, it will:
* * 1. Set the title for the page
* * 2. Apply the CSS to the page
* *
* Both are supplied as parameters to the 'populate' function, and the
* expected invocation is through 'lib/main' by attaching the function
* to the execution scope as a contentScriptFile via Firefox Add-on SDK's
* * 'tab.attach' method.
* */

function populate(styleCss, statsFullName, statsTagLine) {
  document.title = unescape(statsFullName) + " | " + unescape(statsTagLine);
  document.getElementById("stat-stylesheet").href = unescape(styleCss);
}
</code>

Now stop. No seriously, you have now reached a point where it’s absolutely essential that you take a break. If you’ve managed to successfully accomplish everything this FT expects you till now, you deserve a pat on your back.

Let’s move along. Now, a word on what we’re setting out to do. We’ll be taking a lot of help from the D3.js data visualisation library written by Mike Bostock, which is one of the most modern (not to mention beautiful as well as functional) pieces of code in this domain. D3 is supposed to build Data-Driven-Documents, although we can’t confirm if that’s how the name came about. Essentially, data is bound to an SVG (Scalable Vector Graphics) element, and the two then proverbially become one, and we do the manipula-
tion work from there forward. We won’t do many advanced tasks (that D3 is well capable of, by the way) so you should be good even if you have scant prior knowledge of the library, or even the domain. But it’s always a good idea to acquaint yourself with the beast that you’re dealing with.

Okay then, you should now be mentally prepared to tackle the most important script in the add-on, but, on a positive note, also the last one that ties everything together, and makes this comprehensive exercise complete. Let’s get on with ‘visualizr.js’.

```javascript
self.on("message", function(transmission) {
    // Start when message containing data is received
    var version = transmission["version"],
        // Collect browser version from transmission
        memData = transmission["memdata"],
        // Collect single-reporter data from transmission
        tabData = transmission["tabdata"],
        // Collect multi-reporter data from transmission
        name = d3.select("#wrapper")
            .append("p")
            .style("font-size", "30px")
            .style("font-weight", "bold")
            .text(version + " Statistics");

    var margin = {top: 20, right: 10, bottom: 20, left: 10},
        // Defining margins for the display area
        width = 1000 - margin.left - margin.right,
        // Width of the SVG element to be drawn
        height = 500 - margin.top - margin.bottom,
        // Height of the SVG element to be drawn
        rad = 150,
        // Outer Radius of the Donut Charts
        innerRad = 50,
        // Inner Radius of the Donut Charts
        color = d3.scale.category20();
        // Selecting one of D3’s built-in color scales
}
```
Now, notice that this is completely listener-based architecture, with the code kicking into action when we receive the data-set from the main script. Once that’s done, we define the prerequisite common items that will be used throughout the code, such as the data-buckets to collect specific items from the transmitted object along with the visual entities for the page. We’ll be structuring the whole thing as two donut charts (which are basically pie charts with a hole in them), so you can also see the values for their size. Also, there would need to be color scales as we can’t afford to define the color structure for a data-set as dynamic as the one that we have here. We’ll also apply a heading to the page, which we had extracted from the brandStringBundle used earlier in the main script.

Next, onto some D3 dirty work...

```html
•
•
var canvas = d3.select("#wrapper") // Create SVG on 'wrapper' div
  .append("svg:svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom);

  var vis1 = canvas.data([memData]) // Create first visualallization, bind memData
    .append("svg:g")
    .attr("transform", "translate(" + width/4 + "," + height/2 + ")");

  var arc = d3.svg.arc() //this will create <path> elements for us using arc data
    .outerRadius(rad)
    .innerRadius(innerRad);

  var outline = d3.svg.arc()
    .outerRadius(rad+3)
    .innerRadius(rad);

  var pie = d3.layout.pie() //this will create arc data for us given a list of values
    .value(function(d) { return d.amount; });
•
```
If you look at the HTML page, you’ll see that we’re now starting to handle the wrapper divs that we’d declared there. Then, there are some other extractor functions that would be useful in tackling with and applying the data onto the chart when we eventually are ready to have one. For now, we’re just declaring the entities to modulate the code structure, and have all that we need in the namespace. We’ve created the groundwork for the first visualisation which will be fed by the internal data from the single reporters.

```javascript
var arcs = vis1.selectAll("g.slice")
  .data(pie) //associate the generated pie data
  .enter() /*this will create <g> elements for every "extra" data element that should be associated with a selection.
  The result is creating a <g> for every object in the data array*/
  .append("svg:g") //create a group to hold each slice (we will have a <path> and a <text> element associated with each slice)
  .attr("class", "slice"); //allow us to style things in the slices (like text)

var div = d3.select("#wrapper").append("div")
  .attr("class","tooltip")
  .style("opacity",1e-6);

arcs.append("svg:path")
  .attr("fill", function(d, i) { return color(i); })
  .attr("d", arc) //actual SVG path created here
  .on("mouseover", function(d) { d3.select(this).style("fill", "white"); mouseover(); })
  .on("mousemove", mousemove)
  .on("mouseout", function(d, i) {d3.select(this).style("fill", color(i)); mouseout(); });
```

As you can see, we’re now creating the actual arcs that would be holding the data. But what good is a data projection if we don’t know what that
data means? If you remember, we had talked about some very cool tooltips which would be helpful in navigating the charts. Well, here they are. They are supposed to tell the name and amount associated with each slice. As you can see, there are three handler functions bound in the previous code: mouse-over, mouse-move and mouse-out. While mouse-over and mouse-out functions only provide the visual enter and exit scene options, mouse-move does more, injecting the required HTML to get the tooltip to fulfil our requirements. So, here are the three handlers:

```javascript
• function mouseover() {
  div.transition()
  .duration(500)
  .style("opacity", 1);
}

• function mousemove() {
  div
  .html("<b>" + this._data_.data.path + "</b><br/>" + (this._data_.data.amount/1024).toFixed(3) + " kb")
  .style("width", 30+10*this._data_.data.path.length + "px")
  .style("left", (d3.event.pageX) + "px")
  .style("top", (d3.event.pageY) + "px");
}

• function mouseout() {
  div.transition()
  .duration(500)
  .style("opacity", 1e-6);
}
```

Now, what about tab data? We’re getting there. But first, we need to make sure that our data is never engulfed of mixed in with the surrounding, and what better way to do that than to encircle exactly what we want people to see? So we’ll do just that. Then, we do to tab data what we had previously
done to mem data. In exactly the same way too. So you shouldn’t have any problem following the code laid out before you now.

```html
<code>
  arcs.append("svg:path")
  .attr("fill", "black")
  .attr("d", outline);

  var vis2 = canvas.data([tabData])
  .append("svg:g")
  .attr("transform","translate(" + 3*width/4 + "," + height/2 + ")");

  var morepie = d3.layout.pie()
  .value(function(d) { return d.amount; });

  var morearcs = vis2.selectAll("g.slice")
  .data(morepie)
  .enter()
  .append("svg:g")
  .attr("class","slice");

  morearcs.append("svg:path")
  .attr("fill", function(d, i) { return color(i); })
  .attr("d", arc)
  .on("mouseover", function(d) { d3.select(this).style("fill", "white"); mouseover(); })
  .on("mousemove", mousemove)
  .on("mouseout", function(d, i) {d3.select(this).style("fill", color(i)); mouseout(); });

  morearcs.append("svg:path")
  .attr("fill","black")
  .attr("d",outline);
</code>

Finally, now that all of the work that was supposed to be responsive is out of the way, we can now move on to the second div that has been waiting in
the corner, gloomy because we’ve been ignoring it for so long. Don’t worry, ‘finisher’, we’re getting to you. What we’re trying to do here is simply get the user to understand that we’re pulling the same data which is reflected in the about:memory page.

```html
<code>
var footer = d3.select("#finisher")
// Link to `about:memory` as the source of data
.append("a")
.attr("href","about:memory")
.attr("target","_blank")
.style("font-size","15px")
.style("font-style","italic")
.text("Where do you get your data from?");
</code>

And done! That was quite something, wasn’t it?
The Mozilla project has grown way beyond the ‘friendly neighbourhood browser’ stage. The team is working on creating a gamut of technologies and, of course, all of them could use your help!

We hope that by now, you’ve developed a rather comprehensive (but obviously, in no way complete) knowledge of what developing for Firefox entails. With the kind of goals that the Mozilla Foundation set out to achieve, it was inevitable for it to move beyond solely developing a browser – it would need to expand to arenas that are touching the lives of people in a big way, through a medium that’s supposed to be one of the driving forces behind the next generation of internet experience – the mobile.

This means all the awesomeness of Firefox is now packed into a mobile device. The two ways in which Firefox has entered that space are via the mobile browser, Fennec and the mobile operating system, Boot2Gecko (B2G).
Fennec
Fennec is the name of the Firefox for Android project (not the Firefox for mobile umbrella, because there’s no such thing, yet). The project ports Gecko to run on one of the largest, while at the same time fastest growing, platform of our time – Android. Before you ask, no, Firefox isn’t available for iOS, in case you didn’t already know that. That’s because iOS, for some reason, doesn’t allow a browser engine other than its own, to run on the device. This means that all the other browsers that are available on the platform, either run the custom UI on the same core that powers Safari (we’re looking at you, Chrome), or are not really browsers at all, but rather just front-end windows into caching and rendering pages from proxy servers, where the real browser engine resides (that would be Opera). Mozilla has, till date, resented this restriction, and has refused to release the browser for iOS. But that doesn’t mean that it’s any less interested in developing for the mobile platform. The Fennec project, started in the early days of the Android revolution, ran into some early problems, but was released to mild acclaim and generally lukewarm response from the public. Not satisfied with such mediocrity, the browser that you see today underwent a complete overhaul, and yes, a complete rewrite, and turned out to be one of the top rated applications on the Google Play Store, further empowering
Mozilla’s commitment to providing the best web experience to all users of the internet.

Some rather interesting challenges present themselves on the Android platform. We’d just like to take a minute and acknowledge how meta it feels to develop what’s essentially an app for an app on a mobile operating system. Anyway, back to the point – Android doesn’t support XUL, and every add-on must be restartless on the platform. When you think about it, both of the restrictions seem very reasonable, as one can’t expect a mobile user to really shut down and restart an application to use an extension. Now, getting your desktop add-on to comply with such significant restrictions is a rather difficult job, but then that’s what the SDK was developed for – to make the job of the developers easy! The SDK allows you to develop for the mobile platform as well, but is accompanied by relevant flags. Because it’s so early in the stage of development, it doesn’t (yet) allow the developer to extend the degree of control which it does on other platforms. But that doesn’t mean it won’t do so in the future. Also, the much-loved PageMod has been implemented almost entirely on the system, which means that you can change the look and feel of the pages with suitable tools. With the chronic lack of chrome that the platform provides (and the great focus on the page rather than the browser that Firefox seems to be pushing for at the moment), that will probably be the most important and recurring use case for the mobile-based add-ons anyway.

**Boot to Gecko (B2G)**

Boot to Gecko (B2G) is the official moniker for Firefox OS. As you can guess by the name, Boot2Gecko attempts to cut out the middleman and have the whole phone just running Firefox and apps. The extent to it manages to do this is surprising – the structure of the OS is such that all the apps on the system are structurally similar, in that they follow the general struc-
The structure of a web app, which is the HTML/CSS/JS (Hyper Text Markup Language/Cascading Style Sheets/JavaScript, for the uninitiated) holy trinity. What’s interesting here is that this is the general structure for a regular extension of any other form that runs Gecko, showing that it’s actually the very same engine that drives your desktop browser which is running under the hood of phones that are to be powered by B2G.

The core architecture of the Boot2Gecko project includes a Linux kernel at the system’s core, which loads on boot, and continues to load into the Gecko-based run-time engine that thereforforth manages the operation of the device. Mozilla’s embrace of the HTML5 web standard is on clear display, as all the code and APIs are compliant of the specified web standards. Now, the three components that make up the complete operating system are:

1. Gaia
2. Gecko
3. Gonk

Let’s go through a very brief overview of what each of those terms mean, and how they fit into the big picture.

- Gaia is the upper-level of the operating system, and is used to interface with the user and applications.
- Gecko, as we’ve discussed, is the rendering engine that runs at the core of the whole experience.
- Gonk is the lowest tier of the architecture, running the system level calls, and overseeing the Linux base of the operation.

Many may be wondering whether a new mobile operating system was really the need of the hour, considering the impact that has been made by the two existing behemoths, Android and iOS. But the fact of the matter is, the current mobile ecosystem doesn’t fall in line with Mozilla’s goal of making the web open and interoperable. The two operating systems run with
more than just the experience for the end user in mind, with the respective parent organisations tying the users to its own restrictions. It’s rough on the developers as well, because in order to make the same application on two different platforms, they need to learn two different environments, that of Objective-C for iOS and Java for Android. It’s important to remember that when iOS launched, it only supported application development using web standards, but APIs to access low-level hardware were not provided along with it, resulting in a distinct lack of quality of the apps that were produced for the platform during that time. It was only when Objective-C based development was opened up that the ecosystem started to flourish. But, in the process, the web-standard based approach to development took the backseat, and was restricted to being the unloved step-son who nobody cared about anymore. Mozilla aims to rectify this by allowing plenty of freedom with its low-level APIs to the most critical sections of the platform, just the way other platforms do, but to web-standard based apps. What this means is that no special learning curve is involved in getting started to develop for the OS, and that it requires minimal effort to port entities from or to other platforms. When we say device capabilities we actually mean access to hardware and OS-level features and services: we’re talking about

The look and feel of the new operating system

thinkdigit.com
things such as updating the address book, sending SMSes and accessing the camera and media gallery. Another earlier platform, WebOS, also offered hardware access via JavaScript but never tried to standardise its APIs. Mozilla is working with the W3C and other stakeholders to ensure that the Web APIs are an open standard and that other browsers adopt them too.

Now, on to what you need to develop for the platform – almost nothing! All you need is the OS simulator which can be downloaded from AMO, and installed as a regular add-on to your browser. It will provide the basic testing ground that you need for your experiments with the system as you get your feet wet. As for the development process – it’s pretty standard, and similar to our many attempts in this booklet as well. There are Application Manifests in place of add-on manifests that we’ve seen, and so on. Both hosted and packaged apps are supported, suited to the preferences of the developer; however more complex kinds might be supported in the future, such as system-modifying apps like custom keyboards.

There are also numerous side projects initiated by the foundation which attempt to change the web for the better. We’re about to discuss some of the more front-line projects as part of that initiative.

**The Webmaker Project**
The Webmaker Project is built around the idea that passive users of the internet can (and should) be people who contribute to the ‘making’ of
the internet. What it provides is a WYSIWYG (What You See Is What You Get) editor, and an interface to build simple content for the web. This includes, in its own words, Popcorns and Thimbles. Popcorn is a tool to create and maintain effective video-based interactions, while Thimble is used to prepare web pages. It provides a very easy-to-use interface, with a focus on acquainting the user with the basics of web development. One of the most important aspects of the tool is that it's an initiative driven by the force of the community, and that all of the innovations are openly available. You can simply head over to webmaker.org, and view the thousands of examples that are on offer, and start hacking away on any of them (or with a clean slate, there’s no restriction!) to create your own contribution to the community, and to the internet as a whole. Mozilla’s Executive Director, Mark Surman, says that Webmaker is the product of Mozilla's growing commitment to learning, and the culmination of experiments that began with the Mozilla Drumbeat project. “The web is becoming the world’s second language, and a vital 21st century skill — as important as reading, writing and arithmetic,” says Surman. “It’s crucial that we give people the skills they need to understand, shape and actively participate in that world, instead of just passively consuming it. That maker spirit and open ethos is vital to Mozilla, our partners, and the web.”
Now, the important thing to note here is that Mozilla understands that the future of the internet lies in open information for all, and that this can only be made possible if more people are made literate to the possibilities that are on offer on this amazing platform. How remarkable is it that they’ve managed to kill both these birds with one stone. The advanced users will find that Webmaker provides a good WYSIWYG platform for rapid prototyping and simple yet beautiful frameworks to ship web-based projects, while those who are new to the concept would find it an eye-opening experience in how easily their ideas can be worked upon and deployed on the web. We suggest heading over to the Webmaker project at least once; you may find it to be a useful asset, whatsoever be your level of computer literacy.

**The Mozilla Appmaker**
The Mozilla Appmaker is a brand new project aimed at simplifying the process of development of mobile apps for everyone. And everyone means everyone here – including the folks who don’t have a significant digital skill-set. As of this writing, the project is in pre-alpha concept stage, but this is an idea to watch out for. Although many attempts have been made to simplify and effectively automate the process of app development, to be honest, none of them have really panned out. The market is ripe for an initiative of this kind, and who better than Mozilla to make the first significant moves.

There may be an app developer in each one of us!
Props to Mozilla

Finally, we must tip our hats to some of the internal, high complexity projects that the organisation has undertaken recently. First, there's the ongoing attempt to play Flash content without the Flash Player plug-in which, if you've ever been on the internet, is as much of a nuisance as it is a necessity. Google's recent attempts at doing the same thing with Swiffy have produced lukewarm results, but Mozilla looks like it might have hit the nail on the head, with the requisite update scheduled to be bundled with Firefox 27.0.

Then there's the MemShrink Project, which aims to further reduce the consumption of memory in the browser. From the looks of it, the project seems to have hit gold, with internal benchmarks at the company pitting it at saving more than 50% of the memory that was being consumed earlier. Way to not rest on your laurels! 🎉
NEWS AND REVIEWS
Comprehensive news, unbiased reviews and ratings to help you make the right purchase

DOWNLOADS
Check out the latest software downloads, games for your Windows, Linux, Mac and PDA/mobiles

All this and more in the world of Technology

VISIT NOW www.thinkdigit.com
Join 1 lakh+ members of the Digit community

facebook

Digit

Your favourite magazine on your social network. Interact with thousands of fellow Digit readers.

facebook

I Think Gadgets

An active community for those of you who love mobiles, laptops, cameras and other gadgets. Learn and share more on technology.

facebook

Glad to be a Programmer

If you enjoy writing code, this community is for you. Be a part and find your way through development.

devworx

devworx, a niche community for software developers in India, is supported by 9.9 Media, publishers of Digit.