

HTML5, Javascript and Mobile Apps

Introduction

There is currently a revolution in the way that the world-wide-web is being put to use. Web access has gone from being desktop-bound to being ubiquitous, due to the now cheap and easy availability of smartphones and tablet devices with web connectivity. As a consequence of this, people are using the web more frequently, as part of their daily life. Looking up a location in Google maps, playing an online game, accessing social networks on the go to, writing emails, uploading photographs - the range of activities gets wider and more inventive all the time.

This module introduces two of the technologies that are powering this revolution, both on the server and on the actual devices that are used. HTML5 is the most up to date version of the HTML standard, and while it is still a draft standard, it promises to be the most widely used. The draft standard is already supported by all of the mainstream browsers, and all browser manufacturers have committed to implementing the full standard as it emerges. HTML5's features are game-changing, powering up browsers to do things that were previously not possible or difficult to do.

Javascript is the core language of the web. Most current browsers, even those on smartphones, implement it in full, and most implement it in a way that maximizes its speed and power. A decade ago, Javascript was a poor-relation programming language used mainly to validate data entered on web forms. Now it is used as a first-class language to create complex applications that run in a browser - for example, Google Docs (full featured word processing, spreadsheet and graphics), Picasa (a photo-editing suite) and even an operating system shell (Gnome Shell, used to front a version of the Linux O/S).

In this module, we'll take some time to learn the intricacies, strengths, weaknesses and (often) frustrations of working in Javascript. It is likely to be the most important programming language you will ever learn.

HTML and its evolution to HTML5

Sir Tim Berners-Lee devised Hyper-Text Markup Language (HTML) as a specification (i.e. not a working version) in around 1989/90, while he worked as a physicist at CERN. A simple application, called HTML Tags, was released on the web in late 1991 and quickly followed by an application called simply WorldWideWeb. At this stage the web, and HTML, were specialist tools used by scientists and engineers to create distributed, hyper-linked documents. It was capable of creating text documents - no graphics, interactivity or data input.

Marc Andreessen released a browser called Mosaic in 1993, and this quickly became the first popular browser, installed on machines in universities throughout the world and by early adopters of computing technologies (you might call them nerds). Since then, there has been a continuous flow of new and updated browsers so that, today, there are five mainstream desktop browsers (Firefox, a direct descendent of Mosaic, Microsoft Internet Explorer, Opera, Apple's Safari and Google's Chrome). All of these have advanced through numerous updates, most of these being proprietary as each browser manufacturer tried hard to distinguish their own browser from the rest by adding new features to how much better their browser was. There have been a lot fewer updates to the HTML

standard.

HTML 1 & 2 set the basic requirements for a browser – version 1 was essentially a way of marking up and hyper-linking documents, with the addition of images so that diagrams could be incorporated. Version 2 added Forms (so that users could enter information that could be extracted by software at the server-side of a web connection), Tables, image maps and the capability to cope with the different character sets used in different languages across the globe.

Version 3 was the first specification to be published by the World-Wide-Web Consortium (W3C), and so was mostly responsible for nailing down a standard that browser manufacturers would adhere to (to a degree). This version was the one that was in use during the early stages of the public's discovery of the Internet.

HTML4 was also known as Dynamic HTML, and was the first version to incorporate the (new) Javascript language. Version 2.0 of Netscape Navigator (formerly Mosaic, eventually Firefox) was the first browser to incorporate the scripting language, which was rushed to meet the browser's release deadline (this explains a lot about the structure of Javascript as a programming language). The scripting language was formalized as ECMAScript (named by the European Computer Manufacturers' Association), based on the Javascript specification delivered to ECMA in 1996. In response, Microsoft developed a language called JScript, which was completely compatible with ECMA/Javascript, and included it in Internet Explorer 3.0. Although it has maintained the same Javascript compatibility throughout its history, Microsoft still insist on calling it JScript.

Adobe also developed a proprietary version of ECMAScript, called ActionScript, which is an important component of Flash, the web, graphics, animation and interactivity plug-in. ActionScript does have some structural differences from ECMAScript and so will not run in a standard browser without a plug-in.

HTML4 introduced a lot of interactivity into web pages in the late 1990s, but also caused a lot of standards problems. Basically, the combination of browser + HTML, Cascading Style Sheets (a way of attaching templates to a web page to define its appearance) and Javascript was becoming so complex that browser manufacturers could exploit differences in their browsers even though they 'adhered' to the W3C standards. The web was becoming a fragmented platform, where a page that looked one way in one browser could look completely different in another, and scripted operations that worked as intended in one browser may not work at all in another. Clearly, the standards had to be tightened up.

However, near the end of the HTML4 standard's working life, other things were happening to the Internet. Largely because of the addition of scripting, both with Javascript in the browser, and with other languages (like PHP, VBScript and PERL) at the server side, the web was becoming a much more dynamic and interactive place.

Developers were starting to exploit tricks based on relatively simple browser or scripting features - IFrames made it possible to create pages that were mash-ups of more than one source, which introduced the possibility of collaborative sites that would eventually become the backbone of social networking. ActiveX controls developed by Microsoft made it possible to update part of a web page leaving the rest untouched, and thereby give browser pages some of the responsiveness of desktop

applications - this facility was copied by the other browser manufacturers and led to the development of AJAX-based websites such as Google Docs and Picasa. It was now becoming important to incorporate interactivity into a web site just to get people to visit it and keep coming back.

These innovations had to be considered for the next HTML standard, so when the HTML 5 standard was published as a working draft in 2008, facilities for incorporating interactivity, multimedia, cross-document messaging, local storage of data, graphics and other features important to social networking had to be included. However, the main idea behind HTML 5 was that it should be evolutionary (it should not affect the vast majority of web pages already in existence) and yet it should allow new features to be added (with the support of the standards group). At the same time, it should encourage stricter web standards for things like the way pages are marked up and the way interactive elements of a page behave.

World-Wide Web Principles

The World Wide Web is often confused with the Internet. The Internet was developed to allow a number of individual Local Area Networks to be inter-connected using low-level networking protocols – effectively a set of standard for moving binary data between separate networks. IP (Internet Protocol) was developed to be the primary way for machines running on different networks to inter-communicate. It was necessary because different networks from different vendors of the time used different protocols (conventions for communication), and it was necessary to establish a common communications standard. IP is the Esperanto of digital communications.

Once the Internet was established, a number of higher-level protocols were created to support more specific aims – e.g. moving data between different applications running on machines on different networks (TCP), moving whole files from network to network (FTP), moving email messages around the internet (POP, SMTP and IMAP), moving data securely between machines (SSL) etc. One protocol in particular was the direct result of Berners-Lee's work: HTTP is the Hyper-Text Transport Protocol and is the one that is used by computers and network across the world to create the World Wide Web.

HTTP is a client-server protocol. What this means is that it operates between two machines – one is a client, and it requests information from the other, a server. HTTP is special in a couple of respects. Firstly, it is a text-only protocol. Data transported using HTTP must be encoded into a plain text format (i.e. data must be organised as sequences of plain text characters in ASCII (a limited character set containing only a small set of 128 characters – the Latin alphabet plus a few control characters) or Unicode, a newer standard which includes character sets from most of the written languages used in the world. Secondly, it is message-based, meaning that a client sends a request to a server in the form of a message. This message indicates the resource that the client wants to access, and the server's job is to return this resource.

The protocol incorporates a range of return response types, so that if the server cannot be found or the resource is not found at the requested server or the request is denied because client has no permission to access the resource, then a suitable error response is returned to the server.

A typical HTTP request is the URL for a specific HTML document that exists on a specific server. For example, <http://www.webopedia.com/TERM/H/HTTP.html> specifies a HTML document (HTTP.html)

that is located in a specific directory (/TERM/H) of a specific server/domain (www.webopedia.com). The request starts with the name of the protocol (http://). This is the standard format for any web request:



Figure 1.1: The HTTP URL format

This request will either be directed to the appropriate server or a HTTP response code will be returned if the server does not exist or the request was in a bad format, unauthorised or forbidden. If the request reached the server but the requested document or directory was not found, a different error code would be returned. If the request reached the server and the appropriate document was found, the server would return the whole document, along with a response code 200, to indicate the request was successful. You can find a full list of HTTP status codes at <http://www.w3.org/Protocols/HTTP/HTRESP.html>.

Some types of WWW request are a bit more complicated than a request for a file from a server. For example, you are bound to have filled in a form on a web page and pressed the Submit button to set it off (to Amazon to buy a book, to Google to sign up for a new email account etc.). In that case, the URL indicates the web resource that will receive and process the request, typically by taking the data from the form and storing it on a database so that it can then be processed by other systems. The data on the form does not (normally) fit into the URL, and so this additional data (name, credit card details etc.) needs to be packaged so that it is sent along with the request. HTTP provides for additional information to be encoded in the form of Headers – pieces of information related to the request and the response.

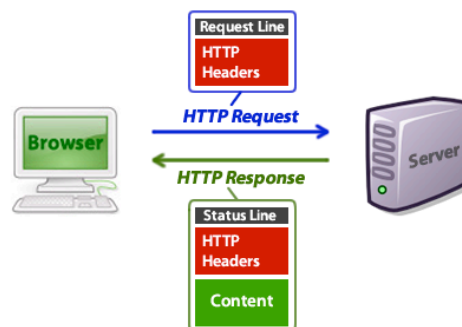


Figure 1.2: HTTP Headers (source <http://net.tutsplus.com/tutorials/>)

You can find a list of HTTP headers at http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

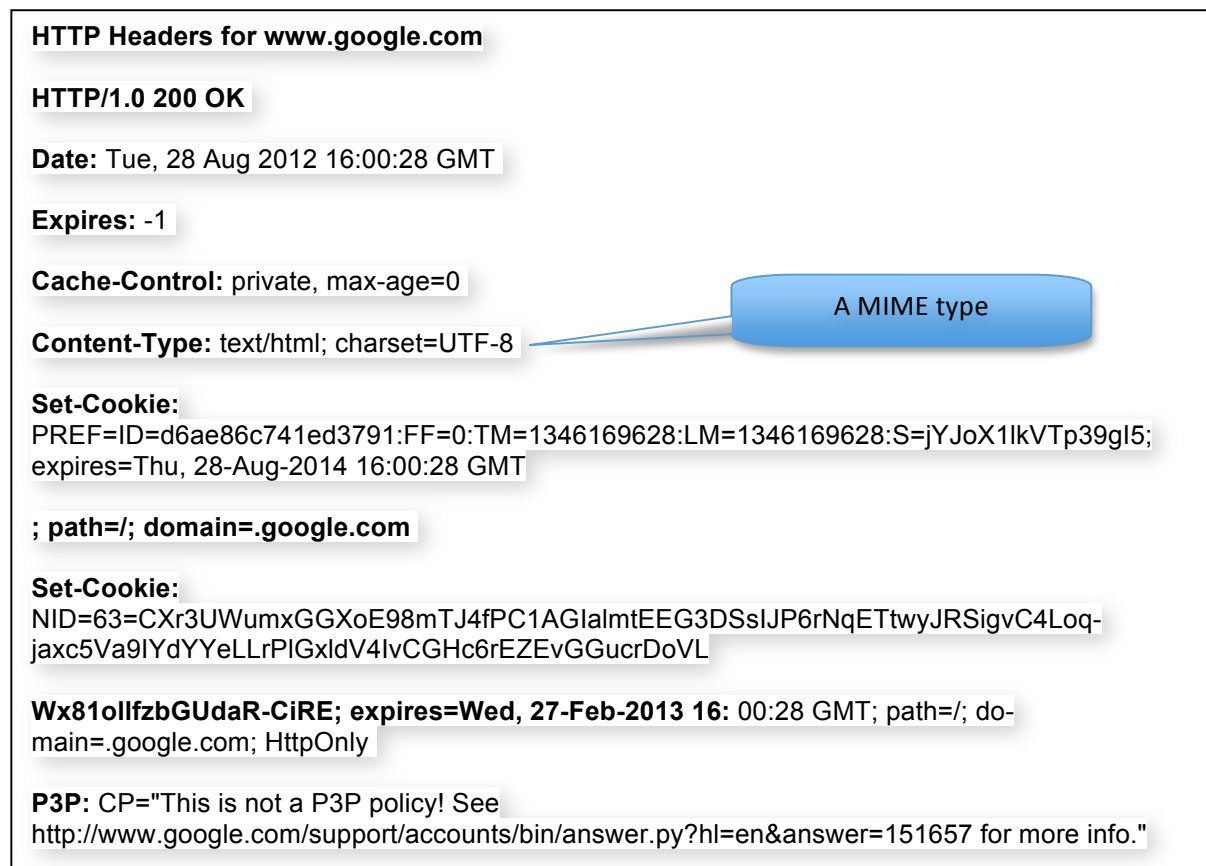
Mime Types

As mentioned previously, the HTTP protocol is restricted to transporting request and response data in a plain text format. This is fine for text documents, but causes a bit of a problem when the client's request is for something that does not fit into plain text – e.g. an image, an audio file, a video or a document that contains non-textual formatting (e.g. in MS Word format).

The HTTP requirement for plain text is not one that can be relaxed, and so anything that is not plain text has to be encoded in a form that passes as plain text. UUEncoding is a form of binary to text

encoding that turns Jpeg, PNG, Ogg etc. files into (much larger) text files that can be sent via HTTP. Web servers and browsers understand this form of encoding and so the server or client can decode a message that has been encoded by a client or server to return it to its original form.

The general-purpose name for file formats that need to be encoded and decoded to cross HTTP channels is MIME types; Multipurpose Internet Mail Extensions – since the formats were originally devised for encoding email attachments. A request or response will always contain headers to indicate the MIME types that are included in the message. For example, here are the headers returned with the response from the www.google.com URL with no search query:



The image shows a screenshot of HTTP headers for a request to www.google.com. The headers are displayed in a light blue box with a black border. A blue callout bubble points to the **Content-Type** header, which is labeled "A MIME type".

HTTP Headers for www.google.com

HTTP/1.0 200 OK

Date: Tue, 28 Aug 2012 16:00:28 GMT

Expires: -1

Cache-Control: private, max-age=0

Content-Type: text/html; charset=UTF-8

Set-Cookie:
PREF=ID=d6ae86c741ed3791:FF=0:TM=1346169628:LM=1346169628:S=jYJoX1IkVtp39gl5;
expires=Thu, 28-Aug-2014 16:00:28 GMT

; path=/; domain=.google.com

Set-Cookie:
NID=63=CXr3UWumxGGXoE98mTJ4fPC1AGIalmtEEG3DSsIJp6rNqETtwyJRSigvC4Loq-
jaxc5Va9IYdYYeLLrPIGxldV4IvCGHc6rEZEvgGucrDoVL

**Wx81oIfzbGUdaR-CiRE; expires=Wed, 27-Feb-2013 16: 00:28 GMT; path=/; do-
main=.google.com; HttpOnly**

P3P: CP="This is not a P3P policy! See
<http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657> for more info."

Figure 1.3: HTTP Headers showing the returned MIME type from a web request (www.google.com)

MIME types are the internet equivalent of the file type information given by an extension to the file name – e.g. .doc or .docx for a Word Document, .xls or .xlsx for an Excel spread-sheet, .png for a type of graphics file etc. However, while some applications and operating systems ignore the file name extensions and use a file's internal structure to work out what type the file is, Internet MIME types are too important for this. The MIME type information contained in a header can tell the recipient the type of resource (i.e. file) before it is downloaded. That can save a lot of wasted effort if the client machine has no way of dealing with a particular type; since there is no point in downloading a file that can't be dealt with, the client will simply ignore any files of that type.

To be able to encode a file appropriately for transporting it, so that it gets properly decoded at the receiving end, the inclusion of MIME type information is essential. You can examine a list of file types and the associated MIME types at <http://www.feedforall.com/mime-types.htm>.

Javascript

Javascript first appeared as a component of the Netscape Navigator 2.0 browser in 1995; it was initially called LiveScript, but the name was soon changed in an early browser update. Javascript was added to the browser to make it a stronger competitor to Internet Explorer. The name Javascript was due to a collaboration between Netscape and Sun (the developers of the Java programming language), to make both languages appear related. In fact, Javascript is a very different language from Java.

Brendan Erlich, a developer for Netscape Communications Corp, was the original designer of Javascript. It is now defined as a standard as the ECMAScript specification (currently ECMA-262) by the European Computer Manufacturers Association) in much the same way that HTML standards are defined and maintained by the W3C. You can read the full ECMAScript specification at <http://www.ecma-international.org/publications/standards/Ecma-262.htm>. It is crucial that an independent body (which ECMA is) maintains this specification, because browser manufacturers tend to create their own Javascript engines, and these all have to be compatible with correct Javascript code, or websites would worked well in one browser may not work in another. Douglas Crockford (see later) does not have very good things to say about the ECMA standards team, but even he will agree that Javascript is better of with a standard than without one.

Javascript files have several MIME types associated with them, depending on the version of Javascript that is supported by a browser. The standard encoding that is put into most HTML web pages is **text/javascript**, which is an obsolete format. The current version for this is **application/javascript** or **application/ecmascript**, both of which acknowledge that Javascript has now been promoted to an appropriate language for creating whole web applications.

Javascript Characteristics

Javascript was devised as a lightweight interpreted language that would appeal to non-professional programmers (unlike Java). Its origins are in several languages of the late 20th century, but mainly C (for much of the syntax) and Lisp (the dynamic data structures and first class functions that appear in Javascript are typical of this language).

Important features of Javascript are:

- Interpreted – i.e. the program code is read, translated into machine executable code and executed *a line at a time* by an interpreter. Some current versions of Java go against this by using a clever form of compiler called a Just-In-Time compiler to turn the whole script into executable code just before it executes it, but the results these deliver are the same as for the earlier interpreted versions
- Lightweight – the core language is very small, and can be implemented fairly quickly by browser manufacturers. It is supported by a very robust test suite, against which a new implementation can be exercised to determine if it meets the ECMA specification
- Dynamic – Javascript variables and objects are created as needed, and the types supported by the language (e.g. numbers, strings, Booleans etc.) are not associated with variable declarations, but with the values assigned to them. This means that a variable that is initially assigned a number, could later have a string, a Boolean, a HTML Element or an AmphibiousLandingCraft object assigned to it. The process of assigning a value to a variable name (called binding) dynamically can make for sloppy programming habits, and so Javascript has only recently been accepted by many programmers as a suitable language for

big programs – mainly because of the emerging discipline of Test Driven Development, which can mitigate against the worst problems introduced by careless programming

- Functional – Javascript functions are also objects, which means that they can be assigned to variables like any other value. This gives the language some very flexible capabilities that lend it huge power. A function's name is an object that can be assigned to a variable – to invoke a function, simply add a pair of brackets () to its name
- Object-based – Javascript can be used for object-oriented programming, but the style of object-orientation it uses is very unlike that used in the so called 'classical' languages – C++, Java, C#, Smalltalk etc. Javascript objects are containers of sets of names bound to values, some of which might be functions
- Prototype-based – a simple Javascript object is nothing more than an associative array, where each element has a name. To enable object types to be extended (providing a similar feature to inheritance in a classical language), Javascript objects have a prototype, which can be extended to add new functionality to the type
- Client-side and server-side – Javascript was developed to be the scripting language of the browser/client. Javascript code delivered as text from a server would be interpreted and executed within the web browser, gaining several advantages – efficiency, security and speed being the major ones. However, various other developments have led to Javascript being used for other types of software. For example, both Apple machines and Windows PCs provide operating system components to allow Javascript code to be executed on the desktop. Also a programming framework called Node.js allows Javascript code to be executed on a server as a very lightweight replacement for web server software like Apache or Internet Information Services (IIS). This is leading to new types of Javascript applications, which can use large numbers of lightweight servers distributed across networks.

One further advantage of Javascript, possibly its biggest, is that Javascript is available on more computers than any other programming language. Since it is built into most browsers, it is almost certain that a given computer will be able to execute Javascript code. For a while, Microsoft set up VBScript, a cut down version of Visual Basic, as a competitor to Javascript, but even they now base much of the client-side application code on Windows machines on a Javascript interpreter.

If you can program in Javascript, you can write code for Windows PCs, Apple Macs, Linux machines, Beos/Haiku systems and, of course, the world-wide-web. If you were given the opportunity to learn only one programming language, you would have to have a very good reason not to choose Javascript.

Javascript Development Practices

Javascript is a general purpose scripting language that has been used for many years to perform simple operations on web pages – validating information entered into forms (e.g. checking that a user has entered a valid date or credit card number), performing simple calculations, driving animations on web pages; also nasty stuff like creating browser pop-ups, phishing, attempting to hijack data and processes on other web pages within the same browser etc. Fortunately the nasty items on this list tend to be mitigated by the browser manufacturers updating their interpreters frequently as new exploits are discovered. The basic premise of a web browser is that script code running on it should not be able to access information outside the page it belongs to, and largely this security is well enforced.

Given that for much of its life, Javascript has been relegated to performing small programming tasks,

it is hardly surprising that many people who use Javascript have tended in the past to not write programs in a very rigorous way. The problem that this introduced was that when these programmers stepped up to performing bigger tasks, for example creating stand-alone browser applications, they often carried their limited programming skills with them.

This may seem like the arrogant opinion of an old-style programmer (it probably is), but programmers with a background in other languages do generally have better programming skills because these languages embodied good general programming principles such as solid design, properly selected data structures, careful management of data types etc. It is important here to emphasize that serious, highly skilled programmers have used Javascript for many years. Typically these were niche programmers working on web-specific applications like Flickr, Google Docs etc. For a long while, the serious Javascript programmers were not promoting their successes because they were working on them too hard.

Douglas Crockford has been the most vocal evangelist for Javascript (see http://en.wikipedia.org/wiki/Douglas_Crockford). Originally a specialist in television technologies, he worked in Atari and Lucasfilm on early video games, Crockford was an early web developer and originated the JSON (JavaScript Object Notation) data format, widely used for exchanging data between web clients and servers by many programming languages (not just Javascript). He developed JSLint – a program that checks Javascript code for potential bugs and misused features (lint is fluff, and JSLint picks the fluff out of Javascript code) and then became the most famous Javascript developer when he published **JavaScript – the Good Parts**. The title of this book says a lot about Javascript; i.e. it is not all good. However, Crockford's book homes in on the parts of Javascript that make it a *very good* programming language, and also points out the areas of Javascript that are best avoided, including how to mitigate the worst problems that the language fosters.

Javascript – Good and Bad bits

I encourage you to have a look at Douglas Crockford's website for a much better explanation of these (<http://www.crockford.com>). However a list of the ups and downs of Javascript follows just so you have some idea what will be coming up in this module:

Good features

- A true dynamic programming language – variables can be used flexibly and objects include all the information needed to identify their type, and therefore how that can be safely used
- Data structures – Javascript has only two built-in data structures: arrays and associative arrays (a.k.a. maps). Both are well implemented, fast and flexible. Since there are only two available, you can learn all you need about data structures without having to cover much ground
- First class functions – Javascript functions are objects, which can have their own properties and methods. A function definition can contain other function definitions - since a function is also an object, one can be assigned anywhere other objects can. This can lead to advanced data management through **closures** – functions with their own private scope
- As a consequence of the flexibility of functions in Javascript, the language provides a

simple and natural syntax for creating event-driven systems. Most working programmers became interested in Javascript around the time when AJAX (Asynchronous Javascript and XML) was devised – in an AJAX enabled website, a web-page can be made highly responsive by executing functions on the web server asynchronously (i.e. in their own time). This facility is only easy to do because of how Javascript functions can be assigned to respond to asynchronous operations

- Prototypical object-orientation – unlike most other languages, Javascript uses the idea of an object prototype (every object has access to a set of functions bound to a prototype, which can be shared among many objects) to provide a form of inheritance that is easy to use and powerfully flexible
- This list could go on, but that's enough for now

Bad Features

- Scope: most programming languages provide methods for reducing the scope of variables – i.e. the amount of program code that can access a variable determines its scope. In general, the smaller the scope that a variable can have, the better. Java, for example, has five types of scope: Global (accessible by the whole program), Package (accessible within a group of classes within the same package), Class (accessible only to code within a class), Object (access limited to the object that owns the variable) and Local (accessible only within a function). Javascript has only two scope types – Global and Local. This makes it harder to do proper object-oriented programming (OOP), since one core part of object-oriented design is *information hiding*, the process of making the internal components of an object inaccessible to other objects. Without class or object scope, this is hard to do (but not impossible). This is Javascript's worst feature by many miles
- Overloading of the + operator: in Javascript, + can be used to add numbers and strings. Unfortunately, it can also be used to add numbers TO strings and strings to numbers. This may seem like a good thing, but because Javascript can also coerce a variable into a different type (e.g. the number 3 can be forced to operate like the string "3") it can lead to lots of errors which are often difficult to find (e.g. does "3" + 3 equal "33" or 6?)
- Semicolon insertion: to make Javascript 'friendlier' to untrained programmers, it was decided that the use of semicolons as statement terminators (taken from the C language) would be relaxed – if a programmer missed out a semicolon, one would be inserted automatically. This was a very bad decision, which can result in code that behaves very differently from the intentions the programmer had for the code that was written. There are good style conventions that get around this problem, but a programmer has to choose to use them
- Amateur programmers providing lots of example code, not very-well written books, a poor quality standard etc.: all of these things make Javascript harder to learn and harder to write good code in.

Javascript Tools

In the next few weeks, we'll concentrate on the good features of Javascript and learn how to work around the bad ones. Fortunately it is very easy to write good Javascript programs, mostly because of the very good tools for supporting Javascript development. These tend to be easier to install and use than tools for other languages.

In this module, one of the best code editors/integrated development systems I've seen (WebStorm)

has been licenced and set up in the labs (a licenced version is also available for installation on your home PC or laptop, although the licence will expire at the end of the academic session). This incorporates JSLint, which means that your Javascript code is checked for bad stuff as you write it. There are also good tools for documentation, testing and code packaging, all available in open source form at no cost.

Mobile Application Development

The general aim of this module is to provide grounding in creating Mobile applications – that is, programs that will run well in a mobile device such as a phone (e.g. an iPhone or Samsung S3) or tablet (e.g. an iPad or Google Nexus 7). HTML 5 and Javascript can go a long way to making this possible; in fact the current HTML 5 specification owes a lot to Apple Computing, who promoted the use of HTML 5 for building iPhone apps in the days before they made the iOS programming frameworks available to developers. However, writing a decent mobile application using HTML 5 and JS can be a complex task, and has the downside that the code written to work on one type of device (e.g. an iPhone) will probably not work well in another (e.g. an Android device).

Mobile Application frameworks have become available to smooth out this process. The two most popular ones currently are jQuery Mobile (which we will use in this module) and Sencha Touch.

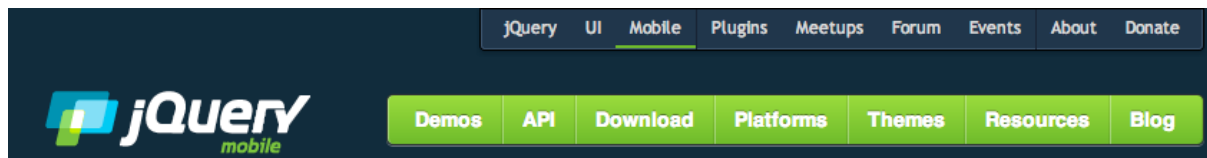


Figure 1.4: The jQuery Mobile Website

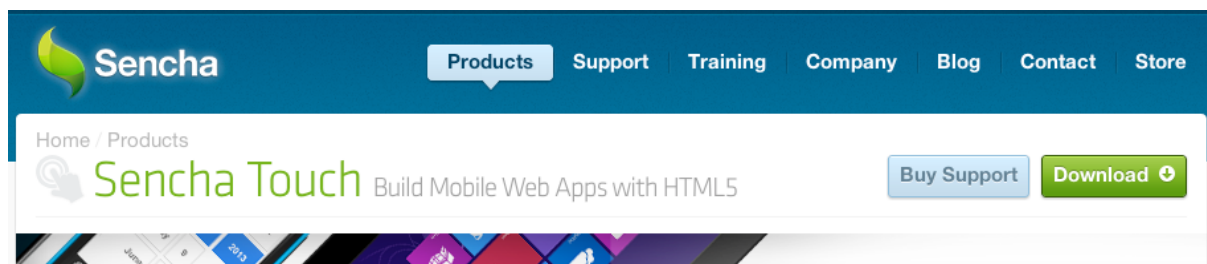


Figure 1.5: The Sencha Touch Website

Both of these frameworks operate on top of HTML 5 by providing page styling and operations that make a standard web-app appear like a native mobile application. Since the alternative is to build native applications, which are specific to each platform they are built for, Mobile Web Apps provide a good basis for creating several types of program that can work on a wide range of devices. Mobile Web Apps do not get full access to the underlying device, so, for example, it is currently not possible to build a web app that can get direct access to the accelerometers on a phone; these are used to detect the movement of the device and so provide different ways for the user to interact with the application. However, web frameworks are continually being extended to incorporate new features common to phones and tablets, so it is possible that this type of control will some become accessible to a web-app.

The main good and bad features of using Mobile Web App frameworks to build mobile applications are:

Good Points

- Mobile web-applications are inherently cross platform, meaning that a single app can work

on an iPhone, a range of Android Devices, a Blackberry etc.

- Mobile web-applications can be distributed from a normal website host – e.g. you could upload the application files to the website space provided by many ISPs and distribute it from there. Native applications for iPhone, Android, Blackberry or Windows Mobile can only be distributed via the corresponding app stores, and generally these have strong restrictions on the apps you can upload
- It is generally easier and cheaper to develop a mobile web-app; there are many free development tools, and specialist software applications for testing, packaging etc. are available as open source programs. Since Javascript and HTML are the common languages for app development, you can do much of the work of building an application with nothing more than a free code editor (Windows Notepad at a push, Notepad++, HTML-Kit or TextWrangler are better) and a web browser
- Many of the specialist features on phones and tablets (e.g. geo-location, local data storage, touch-screen interaction etc.) are accessible to Javascript code and may even be easier to use than the native code libraries for the same features

Bad Points

- Mobile web-apps operate within a browser on a mobile device: that means that there are automatically some restrictions imposed on what the application is allowed to do. For example, browser-based apps can not access the local file system and are not allowed to interact directly with various device applications, such as the contacts app, the diary, the phone or SMS messaging
- Mobile web apps are delivered in the same form as normal HTML applications. That means that instead of a single package file, the app will probably contain several file types (HTML, JS, CSS, graphics, audio etc.) and potentially a large number of files. If any of these files goes astray or gets out of date while other files are updated, there is a strong possibility that the app will crash. iOS, Android, Blackberry and Windows Mobile applications are delivered as a single package, which normally means that they will have been optimised, compressed to take up less space and delivered as a single "all or nothing" component. This is not possible for a Web App, and so web applications have to contain a separate component (a manifest file) that gives the browser knowledge of ALL of the files that will be included in the app.
- None of the platform vendors (e.g. Apple, Google) allow web-apps to be sold through their online app-stores. This makes it more difficult to market a web-app since there is no direct channel to the devices (as there are on an iPhone with the iTunes store, or Android through Google Play), and this means that as a developer, you will need to handle all of the online distribution, sales, payment processing etc. or delegate it to an 'alternative' online store. Of course, the vendor's stores all take a large cut of every sale (30% is typical) and so this can be seen as an advantage. However, there is no denying that an app that appears on the iTunes store or Google Play will sell in much bigger numbers because of the in-built sales, marketing and trust that these stores provide

This Module

A week-by-week description of the content in this module is available on Moodle. See the Module Guide, at the end of the Module Overview panel. There is no practical work for this week, but next week you should come prepared to get your hands dirty writing HTML5 & Javascript code. There is a download link for the WebStorm software in the Week 1 panel in Moodle, and a guide for installing and registering the software.