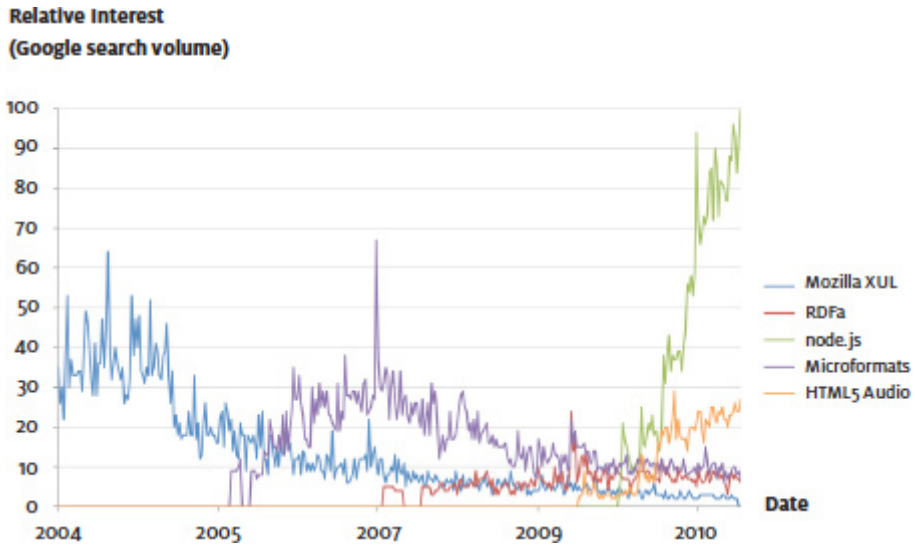
# Web technology fundamentals

Web technologies are in a constant state of flux. It's impossible to predict which will fail, which will shine brightly then quickly fade away, and which have real longevity. Rapid innovation is what makes web app development so exciting, but shiny new things shouldn't be pursued without a solid understanding of the underlying web platform.




## Web architecture primer

Let's start with DNS (domain name system) and HTTP (hypertext transfer protocol). These are the underlying systems that web browsers use to send and fetch data to and from web apps. Familiarity with these protocols is essential for later discussions on application programming interfaces (APIs), performance and security.

### DNS

When you type an address into a web browser or follow a link, the browser first has to identify which computer server in the world to ask for the content. Although web addresses use domain names like fivesimplesteps.com to make them easier for people to remember them, computers use unique numbers to identify each other[1].

To convert names to numbers, the browser queries a series of DNS servers, which are distributed directories of names and numbers for all web servers. To speed up this process, the lookups are cached at a number of locations: your internet service provider (ISP) will hold a cache, your operating system may hold a cache and even your web browser software will hold a short lifetime cache.

**Host resolver cache** Clear host cache

- Capacity: 100
- Time to live (ms) for success entries: 60000
- Time to live (ms) for failure entries: 0

| Hostname | Family | Addresses | Expires |
|---|---|---|---|
| 0.gravatar.com | ADDRESS_FAMILY_IPV4 | 93.184.216.169:80 | Sun Feb 20 2011 10:17:44 GMT+0700 (ICT) |
| 1.gravatar.com | ADDRESS_FAMILY_IPV4 | 93.184.216.169:80 | Sun Feb 20 2011 10:17:44 GMT+0700 (ICT) |
| ad.doubleclick.net | ADDRESS_FAMILY_IPV4 | 209.85.175.148:80<br>209.85.175.149:80 | Sun Feb 20 2011 10:33:57 GMT+0700 (ICT) |
| addons.mozilla.org | ADDRESS_FAMILY_IPV4 | 63.245.213.91:443 | Sun Feb 20 2011 10:08:41 GMT+0700 (ICT) |
| api-secure.recaptcha.net | ADDRESS_FAMILY_IPV4 | 64.34.251.153:443 | Sun Feb 20 2011 10:08:49 GMT+0700 (ICT) |
| api.flattr.com | ADDRESS_FAMILY_IPV4 | 95.215.16.13:80 | Sun Feb 20 2011 10:03:53 GMT+0700 (ICT) |
| b.scorecardresearch.com | ADDRESS_FAMILY_IPV4 | 110.164.2.40:80<br>110.164.2.24:80 | Sun Feb 20 2011 10:34:32 GMT+0700 (ICT) |
| bbc.112.2o7.net | ADDRESS_FAMILY_IPV4 | 66.235.132.152:80 | Sun Feb 20 2011 10:34:31 GMT+0700 (ICT) |
| bits.wikimedia.org | ADDRESS_FAMILY_IPV4 | 208.80.152.118:80 | Sun Feb 20 2011 11:09:21 GMT+0700 (ICT) |
| blogbuildingu.com | ADDRESS_FAMILY_IPV4 | 74.207.242.89:80 | Sun Feb 20 2011 10:17:41 GMT+0700 (ICT) |
| button.topsy.com | ADDRESS_FAMILY_IPV4 | 74.112.128.18:80 | Sun Feb 20 2011 10:58:11 GMT+0700 (ICT) |
| c.statcounter.com | ADDRESS_FAMILY_IPV4 | 216.59.38.123:80 | Sun Feb 20 2011 10:03:55 GMT+0700 (ICT) |
| careers.stackoverflow.com | ADDRESS_FAMILY_IPV4 | 64.34.80.176:80 | Sun Feb 20 2011 10:07:54 GMT+0700 (ICT) |

*Google Chrome's integrated DNS cache*

### HTTP requests

Once your browser has identified the correct number associated with the domain name, it connects to the server with the equivalent of, "Hello, can I ask you something?" The connection is agreed and your browser sends a message to request the content. As a single web server can host thousands of websites, the message has to be specific about the content that it is looking for.

Your browser will add supplementary information to the request message, much of which is designed to improve the speed and format of the returned content. For example, it might include data about the browser's compression capabilities and your preferred language.

An HTTP request message for the BBC technology news page will look similar to the example below. Each separate line of the message is known as an HTTP header.

```
GET /news/technology/ HTTP/1.1
Host: www.bbc.co.uk
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.2.13) […]
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
```

The first line states the method of request (GET), the local path to the requested resource, and the version of HTTP used. GET is the most common HTTP method and asks the server to return the content found at the specified location. POST is another common method, which sends data collected in the browser to the server.

The *Host* header field tells the server which of the potentially thousands of locally hosted websites to check for the resource, and the *User-Agent* describes the browser making the request.

The various Accept fields define preferences for the returned content. Rather than waste time with numerous back and forth messages ("Can I have it in this format? No? OK, how about this format?"), Accept header fields can specify multiple preferences separated by commas. Each can be assigned a degree of preference defined by a quality score q of between 0 and 1. If a q value isn't specified it is assumed to be 1. In the example above, the browser is asking for HTML or XHTML with equal full preference (q=1), followed by XML (0.9), and finally any format (0.8).

The *Keep-Alive* and *Connection* fields ask the web server to temporarily create a persistent connection. This speeds up requests that immediately follow this request, as they don't need to each perform the initial connection handshake of "Hello, can I ask you something?" An added benefit of persistence is that the server can stream back the content in chunks over the same connection, rather than waiting for it all to be ready for a single response.

### HTTP responses

The response from the server to the browser also contains an HTTP message, prefixed to the requested content.

```
HTTP/1.1 200 OK
Date: Sun, 20 Feb 2011 03:49:19 GMT
Server: Apache
Set-Cookie: BBC-UID=d4fd96e01cf7083; expires=Mon, 20-Feb-12 07:49:32 GMT;
path=/;domain=bbc.co.uk;
Cache-Control: max-age=0
Expires: Sun, 20 Feb 2011 03:49:19 GMT
Keep-Alive: timeout=10, max=796
Transfer-Encoding: chunked
Content-Type: text/html
Connection: keep-alive

125
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
[Remainder of HTML…]
```

The opening *status line* contains the HTTP version number, a numeric response code and a textual description of the response code. The web browser is designed to recognise the numeric response code and proceed accordingly. Common response codes include:

- **200:** *OK*
  Successful request
- **301:** *Moved Permanently*
  The requested content has been moved permanently to a new given location. This and all future requests should be redirected to the new address.
- **302:** *Found*
  The requested content has been moved temporarily to a given address. Future requests should use the original location.
- **404:** *Not Found*
  The server cannot find the resource at the requested location.

- **500:** *Internal Server Error*
  A generic error message, shown when an unexpected condition prevents the request from being fulfilled.

The browser doesn't understand the textual part of the line. It can be used by the web server to elaborate on a response, and is intended for users or developers who read the HTTP headers. For example, a 200 OK response could also be sent as *200 Page Found*.

The *Keep-Alive* and *Connection* header fields establish rules for the persistent connection that the browser requested. In the example, the Keep-Alive field tells the browser that the server will hold the connection open for up to 10 seconds in anticipation of the next request. It also specifies that up to 796 additional requests can be made on the connection.

The *Cache-Control* and *Expires* fields control caching of the returned content, which might occur within the browser or at any number of intermediate proxy servers that exist between the web server and the user's computer. In the example, the immediate expiry date and cache age of zero inform the browser that it should check for a new copy of the page before using a locally cached version on subsequent requests.

The *Transfer-Encoding* value of *chunked* notifies the browser that the content will be transferred in pieces. The content begins after the final header field and is separated from the HTTP header by two newlines. Each chunk of content starts with a hexadecimal value of its size expressed in octets (units of 8 bits): 125 in the example.

### Statelessness and cookies

HTTP is *stateless*. This means that multiple requests from the browser to the server are independent of one another, and the server has no memory of requests from one to the next. But most web apps need to track state to allow users to remain logged in across requests and to personalise pages across sessions.

HTTP cookies are the most common solution to this problem. A cookie is a small text file that the browser stores on your computer. It contains a name and a value associated with a specific website (for example, a name of *age* and a value of *43*). Cookies can be temporary or can persist for years.

Each website domain can create 20 cookies of up to 4kb each. Cookies are created and read through HTTP headers. In the BBC HTTP response, the *Set-Cookie* header field demonstrates the creation of a cookie.

```
Set-Cookie: BBC-UID=d4fd96e01cf7083; expires=Mon, 20-Feb-12 07:49:32 GMT;
path=/;domain=bbc.co.uk;
```

In this example, the web server asks the browser to create a cookie with the name *BBC-UID* and a value of *d4fd96e01cf7083*. The cookie is valid for all domains that end with bbc.co.uk and all directories. The expiry date for the cookie has been set to a year after the time of the response.

Subsequent HTTP requests from the browser that match the valid domain and path will include the cookie as an HTTP header, which the server can read:

```
Cookie: BBC-UID=d4fd96e01cf7083
```

What does this random-looking BBC cookie mean?

Although cookies enable real user data to be stored and read across requests, in practice they are usually used to store unique identifiers for users rather than actual user data. The small size of cookies, the additional bandwidth overhead in HTTP headers and the security risk of storing sensitive data in plain text cookie files all combine to make unique identifiers a better solution for cookie data. With this model, user data is stored securely on the server and associated with a short unique identifier; it is the identifier that is subsequently used in cookies for persistence.

If the expiry date for a cookie isn't set it becomes a *session cookie* and is deleted when the user ends their current session on the website. Due to privacy concerns, some users may configure their web browser to allow session cookies but disallow standard *persistent* cookies.

## Content type

Your browser now has the content it requested, thanks to the HTTP response from the server. Before the content can be processed and displayed though, the browser needs to determine what type of content it is: an image, PDF file, webpage, or something else.

One way a browser can achieve this is through content sniffing. The browser examines the first few bytes (and sometimes more) of the content to see if it recognises a pattern in the data, such as a PDF or JPG header. Apart from the accuracy and performance issues that this may introduce, it can also have security implications[2].

The better solution is for the server to tell the browser what the content is with an HTTP header field in the response, such as the one in the BBC example:

```
Content-Type: text/html
```

The *Content-Type* field specifies the internet media type[3] of the content. Media types can identify most common file formats[4], including videos, images, webpages, spreadsheets and audio files. The web server is normally configured to send the correct content type header field based on the file extension. If your app delivers any special data or file formats, ensure that the relevant media types are configured on the web server.

## Character encoding and Unicode

At this point, images and other binary files can be correctly interpreted and displayed by the browser. However, HTML pages and other text-based content are still unreadable by the browser due to the different *character encodings* that can be used.

Like all other files, text files are streams of bytes of data. In an image file, a set of bytes might define the colour of a single pixel. In a text file, a set of bytes might define a single character, perhaps a Japanese kanji character, the uppercase letter B of the Latin alphabet, or a semicolon. How exactly do the bytes map to characters? The answer is: it depends on the

character encoding. Until the browser knows what the character encoding is, it doesn't know how to create characters from the bytes.

In the early days of computing most text was stored in ASCII encoding, which can represent the basic Latin alphabet with only seven bits per character. Additional encodings followed, each designed to handle a specific set of characters: Windows-1251 for the Cyrillic alphabet, ISO 8859-8 for Hebrew, among many others.

Each encoding standard stored one character in one 8-bit byte. ISO 8859-1, also referred to as Latin-1, became a popular encoding that remains widely in use. It uses the eighth bit to extend ASCII with accents and currency symbols found in many of the western European languages. Thanks in part to the internet, this system became increasingly unworkable as multiple diverse alphabets were required in a single file. The sensible way to achieve this was to start using more than eight bits to represent a single character.

*Unicode* was born. Rather than defining a specific encoding, Unicode sets out over one million *code points*, which are numbers that represent characters. For example, the Greek capital letter Sigma Σ is Unicode number 931, the Arabic letter Yeh ي is 1610, and the dingbat envelope character ✉ is code point 9993.

Multiple encodings of Unicode exist that define how to store the code point numbers in bytes of data. UTF-8 and UTF-16 are two such encodings. Both can encode the full range of more than one million characters and both use a variable number of bytes per character. The main practical difference between the two is that UTF-8 uses between one and four bytes per character, whereas UTF-16 uses two or four bytes per character.

Most importantly, because the first 128 characters defined by Unicode exactly match those of ASCII, UTF-8 is backwards compatible with ASCII, as it only uses one byte per character for these lower code points. This is not so for UTF-16, which uses a minimum of two bytes per character and therefore uses twice as many bytes to store standard ASCII characters.

A web server can notify a browser of the character encoding through an additional parameter in the Content-Type HTTP header field:

```
Content-Type: text/html; charset=utf-8
```

This allows the browser to decode the content immediately. Alternatively, the character encoding can be specified inside the HTML `<head>` element with an HTTP equivalent `<meta>` tag:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

or the HTML5 version:

```
<meta charset="utf-8">
```

This is useful if you don't have access to the web server to configure the correct HTTP header. The browser will usually be able to read this directive no matter what the encoding is, as most encodings extend ASCII, which are the only characters used in the `<meta>` tag. However, the browser may consequently have to reparse the document a second time with the correct

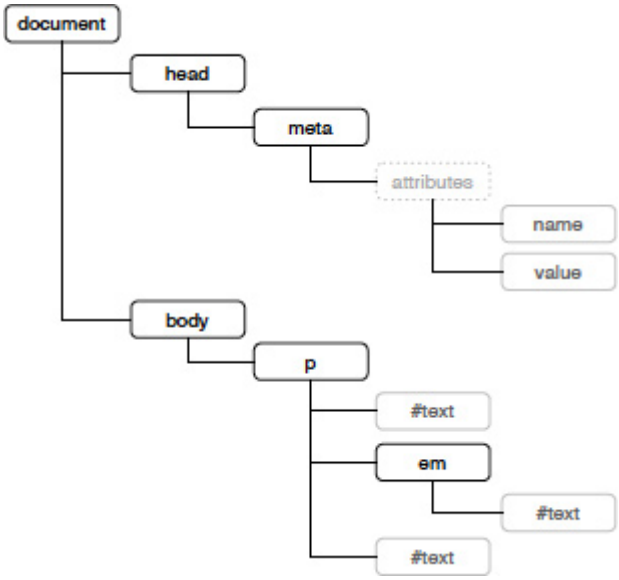
encoding. For this reason, the HTTP header field is the preferred option, but if you do use the `<meta>` tag ensure that it is the first element inside the `<head>` element so that the browser reads it as soon as possible.

As with media types, omission of the character encoding is not recommended and can be a security risk.

## Document object model (DOM)

With knowledge of the encoding, the browser can convert the incoming bytes into the characters of the webpage. It runs a *parser* through the content which recognises HTML elements and converts them into a tree-like hierarchy of nodes in memory called the document object model[5] (DOM) or content tree.

DOM nodes for HTML elements become *element nodes*, text strings within tags are *text nodes*, and attributes of an element are converted to *attribute nodes*.




Apart from structure, the DOM defines a standard interface (API) that scripts can use to access, modify and move between nodes, though this interface is implemented with varying degrees of completeness across different browsers.

When the HTML parser reaches a reference to an external resource like an `<img>`, it requests the file from the web server even if it is still downloading the remainder of the HTML content. Most modern browsers allow six simultaneous requests per host and over thirty requests in total at any one time.

Style sheets and JavaScript files are notable exceptions to this rule. When the parser encounters an external style sheet file, it may block further downloads until the style sheet is downloaded, although this is now rare in modern browsers.

JavaScript files are a little more problematic. At the time of writing, Internet Explorer 9 and earlier block the download of subsequent image files until a JavaScript file is downloaded and executed[6]. What's more, all browsers will stop any rendering of the page until the JavaScript is processed, in case the code makes a change to the DOM. We'll discuss this in more detail shortly.

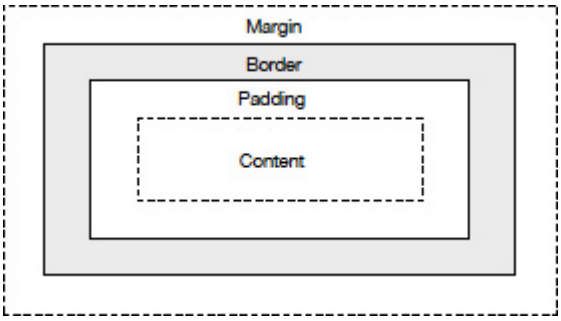
If your JavaScript doesn't modify the DOM, you can add an async or defer attribute or both to your `<script>` elements to prevent blocking. As these attributes aren't currently supported in all popular browsers, the best cross-browser advice at the moment is to:

- Include style sheets before scripts so that they can begin to download before any JavaScript blocks.
- Place scripts at the end of the HTML, just before the `</body>`, so that they don't block downloads or rendering.
- Force scripts to download asynchronously using one of many workarounds; search the web for *loading scripts without blocking* to find a variety of options.

## The render tree and layout

After the style sheets have downloaded, the browser starts to build a second tree of nodes, even if the DOM is not yet complete. The render tree is a visual representation of the DOM with a node for each visual element[7]. Style data is combined from external style sheets, inline styles, outdated HTML attributes (such as `bgcolor`) and the browser's default style sheet.

Render tree nodes are rectangles whose structure is defined by the *CSS box model* with content, padding, borders, margins and position:




*The CSS Box Model*

Render nodes are initially created without the geometric properties of position and size. These are established in a subsequent *layout* process that iterates through the render tree and uses the CSS visual flow model[8] to position and size each node. When the layout is complete the browser finally draws the nodes to screen with a *paint* process.

If a geometric change is made to a DOM element post-layout, by JavaScript for instance, its relevant part of the render tree is invalidated, rebuilt, reflowed and repainted. Conversely, changes to non-geometric DOM node properties such as background colour do not trigger a reflow and are, therefore, faster.

Historically, some browsers didn't fully conform to the CSS specification for the layout process, causing inconsistency in HTML layout between browsers. In an effort to fix the situation while providing backwards compatibility, this gave way to three layout modes: *standards*, *quirks* and *almost standards** (which is identical to standards mode except for the layout of images inside table cells).

*Layout rendering is the biggest difference between the modes, but there are also some minor non-layout differences too, such as HTML parsing.

Unless your app is specifically designed for an environment that exclusively uses old web browsers, you should trigger the browser into standards mode by including a valid `DOCTYPE` at the start of the HTML:

```
<!DOCTYPE html>
```

*HTML 5, but backwards compatible with most popular web browsers, down to IE6*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
```

*HTML 4 Strict*

If necessary, use *conditional comments*[9] to resolve layout issues in older versions of Internet Explorer, the main quirks mode villain:

```
<!--[if lte IE 6]>
Include special CSS or other quirk fixes here
<![endif]-->
```

## JavaScript and the browser object model

Most web apps need to modify the DOM to deliver interactive content without the cost of a new page request. The DOM provides an API for this purpose but it isn't a programming language. Instead, client-side scripts are written in JavaScript, which interfaces with the DOM API. The DOM is separate from the JavaScript engine in a web browser and consequently there is some overhead for each JavaScript request to the DOM.

As far as JavaScript sees the world, the DOM is part of the larger *browser object model* (BOM), which contains several sets of data about the browser. Unlike the DOM, the BOM is not an agreed industry standard and it exhibits greater discrepancy between browser vendors.

The BOM is represented in JavaScript by the `window` object and typically contains the `navigator`, `frames`, `location`, `history`, `screen` and `document` (DOM) objects. The global `window` object is the default context of JavaScript, which means that it is the default location to store variables and functions.

Two key security policies limit JavaScript's access to the browser. Firstly, the JavaScript *sandbox* restricts functionality to the scope of the web, to prevent scripts from opening or deleting files on the user's local operating system. Secondly, the *same* origin policy prevents communication between scripts on different domains or protocols: that is, between dissimilar pages or third-party embedded frames. A notable exception is that scripts included from other hosts behave as if they originate on the main page host. This exception allows third-party widgets to modify the DOM if they are included within a `<script>` element.

## Ajax

We've seen that client-side JavaScript code can interface with the DOM inside the browser to modify the webpage without a page refresh, but what if it needs to modify part of the page with additional data from the web server?

This is where Ajax comes in. The term originates[10] from a contraction of asynchronous JavaScript and XML, though in modern usage it requires neither asynchronicity nor XML, and is used as a catch-all term for the various technologies and methods that enable communication between JavaScript and a web server.

The heart of most Ajax is the XMLHttpRequest API inside JavaScript. This feature enables JavaScript code to send invisible, customisable HTTP requests to the web server without altering the current page, and the asynchronous nature of JavaScript and XMLHttpRequest (XHR) allows other processing to continue while the web server responds. If the response is chunked into pieces, the XHR can trigger multiple responses in the JavaScript code to process the content as it is received. It's worth noting that, as with full page requests, the browser may cache the data returned from XHR GET requests, depending on the HTTP header returned.

XML data is neither particularly lightweight nor quick to process. It is now common practice to use the alternative JSON (JavaScript object notation) data format for Ajax communication, which is smaller to transmit and faster to process in JavaScript. Most modern web browsers can natively parse JSON data into JavaScript objects, and all popular server-side technologies offer JSON libraries.

Alternatively, an XHR response may contain a section of ready-made HTML. This may be larger than an equivalent JSON response but it reduces client-side processing and can be inserted directly into the DOM.

XHR is restricted by the same origin policy and cannot communicate with a server on a different domain. The restriction is removed if the server includes an explicit instruction to allow cross-domain requests for a resource:

```
Access-Control-Allow-Origin: http://fivesimplesteps.com
```

*Allows cross-domain requests to the resource from fivesimplesteps.com*

This header is supported in most modern browsers: IE8+, Firefox 3.5+, Safari 4+ and Chrome. Cross-domain Ajax requests in older browsers require workarounds, of which JSONP[11] is the most popular option, albeit the most convoluted.

## JSONP

The JSONP technique (JSON with padding) uses JavaScript to dynamically insert a `<script>` element into the page, the source of which is loaded from the third-party domain.

```
<script src="http://www.anotherdomain.com/getjsondata?function=responseFunction">
</script>
```

This is valid because, as we noted earlier, scripts loaded into the page don't face the same cross-domain restrictions. Still, so far the returned data (typically JSON) will simply be inserted into the `<script>` element, which isn't accessible to the JavaScript:

```
<script>
{"Name": "Dan Zambonini", "Age": 35}
```

```
</script>
```

This is where the padding comes in. In the earlier `<script>` element, the URL specified an existing function name as a parameter: in our case, `responseFunction`. The server-side code processing the request takes this name and wraps it around the JSON output, to modify the response from a simple line of data to a function call:

```
<script>
responseFunction ({"Name": "Dan Zambonini", "Age": 35})
</script>
```

When the script is processed, the requested function will automatically execute with the returned data as entered, enabling the data to be processed.

While workable, the JSONP hack has major drawbacks compared to XHR. HTTP headers cannot be modified in the request or examined in the response, which limits communication to GET only, and complicates error handling and retries. Response data cannot be processed in chunks and must be formatted as executable JavaScript code. Perhaps most importantly, the browser executes the returned code immediately and therefore the trust and ongoing security of the third-party server must be considered.

## Summary

Knowledge of the underlying web technologies enables you to develop workarounds for web browser restrictions and optimise performance and security.

- DNS converts domain names to computer-usable identification numbers.
- HTTP messages govern the requests and responses between web browsers and web servers.
- HTTP is stateless, but cookies can be used to remember a computer from one request to another.
- Content-type HTTP header fields tell the browser what type of content is being sent.
- Character encoding headers tell the browser how to understand text files.
- UTF-8 is the most practical character encoding for the web.
- Web browsers convert HTML into a document object model (DOM) tree in memory.
- A second render tree is created in browser memory from the DOM, to represent the visual page layout.
- Use a `DOCTYPE` to tell the browser which layout mode to use.
- JavaScript can modify the DOM, and Ajax techniques can request additional data from the server and make partial updates to the DOM.

---

1 I cunningly sidestep the IPv4 vs IPv6 issue here, as IPv4 was exhausted a week before I wrote this chapter. See http://en.wikipedia.org/wiki/IPv6#IPv4_exhaustion

2 http://code.google.com/p/browsersec/wiki/Part2#Survey_of_content_sniffing_behaviors

3 http://en.wikipedia.org/wiki/Internet_media_type

4 http://www.iana.org/assignments/media-types/index.html

5 http://www.w3.org/DOM/

6 http://www.browserscope.org/?category=network

7 And sometimes more than one node in the render tree per DOM node, e.g. for multiple lines of text

8 http://www.w3.org/TR/CSS2/visuren.html

9 http://www.quirksmode.org/css/condcom.html

10 http://blog.jjg.net/weblog/2005/02/ajax.html

11 http://en.wikipedia.org/wiki/JSON#JSONP