Web Technologies: Client-Side

COVERING DIVERSE TOPICS RELATED TO
CLIENT SIDE WEB DEVELOPMENT AND
FOCUSING ON THE CORE TRIUMVIRATE
OF HTML, CSS, & JS

BY

DR SIMON WELLS

SESQUIPEDALIA VERBA PUBLISHING LTD

Contents

Chapter 1

Administrivia

1.1 Introduction

Welcome to the Web Technologies module from Edinburgh Napier University. This module has a slightly different structure to many modules so it's worth reading through this guidance before you get stuck into the good stuff.

How should we use this workbook? Ideally we would work through it on a chapter-by-chapter basis supplementing our work with background reading and wider exploration of each topic introducted. Some chapters will take longer to complete than others, and other chapters will need to be returned to multiple times. This is particularly true for the first two chapters. To learn both Linux and Python in a fortnight is a tall order so I'd suggest working iteratively, do enough to make some progress, then frequently return to the respective chapters to learn a little more, usually by following the links and footnotes to further practise materials.

In the first week work through the first chapter in the lab section of the workbook. You can read ahead if you want but don't try to run any Flask web-apps on the dev server until you've been assigned your personal virtual server to run your own web-apps on. This week is mostly concerned with the foundation of our learning environment. Logging in, learning to navigate and do simple tasks at the command line, using a non-gui text editor, and using Git. There are links, usually in footnotes, throughout the chapter, for example, to practise the Linux command line then there are online web sites like the *LinuxZoo*[1] that you can use to practise your skills. Similarly, the links to Vim practise tutorials, particularly the Vim game, will help you practise the skills you need to work efficiently in subsequent weeks. Finally, make sure to work through the linked Git tutorials and ensure that you are confident that you understand each of these tools and their place within the learning environment before moving on to subsequent chapters.

Each chapter is meant to cover about an entire week of study, so don't rush through things within the scheduled lab session just to tell yourself the you've done all the work. As I mentioned in the introduction, topics, whether in lectures or labs are meant to be a starting place, a framework to guide your self-directed study, but not the totality of your learning.

[1] `http://www.linuxzoo.net`

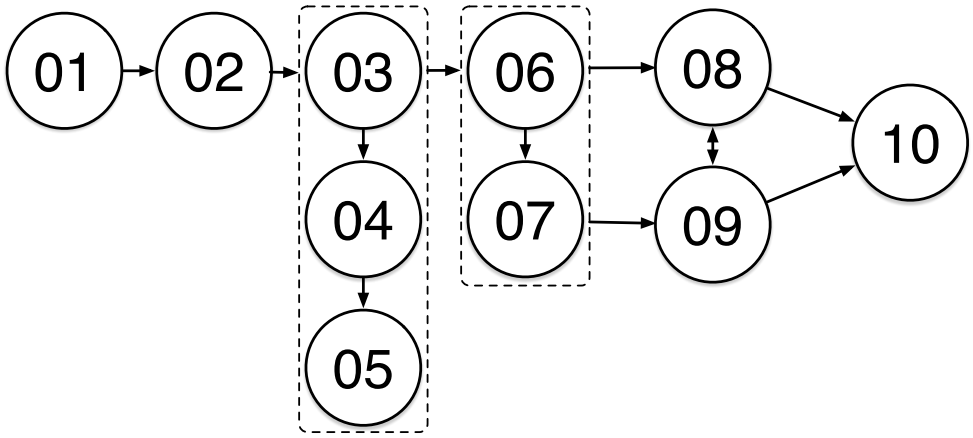
In subsequent weeks you will start to build knowledge of the Python language and the Flask library which provides functionality for building server-hosted web-apps. The next chapter, on Python, is meant to cover at least a weeks work and deliberate practise. Mostly that week is concerned with developing basic skills in a new language, Python, which is actually quite a challenge. This is not because Python is particularly difficult but because learning a computer language well takes time and effort and you have to start somewhere. Subsequent weeks will require you to work through various chapters of the workbook. You will find that as you progress you will want to skip ahead to different sections, especially once the assessments are released and you want to include specific functionality in your coursework. So after about chapter 3 I expect that many of you will navigate your own path through the remainder of the workbook. The only proviso is that you should aim to have worked through every chapter by the end of the module.

1.2 Reading This Book

This book covers a variety of technologies, some of which you might already be familiar with. Also, depending upon your goals, you might have specific topics that you want to study. We'll cover the following topics:

1. The World Wide Web - An Introduction & Overview
2. Hypertext+Markup+Information+Semantics=HTML
3. CSS Intro & Overview
4. HTML Page Layout Using CSS
5. Design for Hackers
6. Core JS
7. Client-Side JS: Browser APIs
8. Client-Side JS: Data Storage APIs
9. Client-Side JS: Sound & Vision APIs
10. Deployment

THe following figure suggests a few routes through our topics, grouping together thematically related topics, and ensuring that pre-requisit chapters are indicated.


01
02
03
04
05
06
07
08
09
10

Chapter 2

The World Wide Web

In this unit we begin the module with an introduction to the Web and an overview of the technologies which it comprises. From this starting point we can use subsequent units to take deeper dives into the various specific tools and technologies so that we can build a balanced and professional toolbox of Web related skills. You've probably used the web before, even if only to access this module, but more likely it is a part of your daily life so you probably have a number of preconceived ideas about how it works, how it should work, and where it is annoying and in need of improvement. You might also have had some previous experience in the Web. This is good but not necessary. We'll start from scratch in our approach to the Web which is an opportunity to reconfirm your ideas if you already know some parts and an opportunity to build new skills and understanding otherwise.

Before we can go anywhere, we should really know how we got here. This is important because the Web has been developed incrementally over around 30 years. However it is also based upon ideas that go back to at least 1945 when Vannevar Bush described the ideas of Hypertext and an associated machine, the Memex, and probably earlier. The Web, on the most basic level is really a conceptual tool for making knowledge capture, communication, and reasoning easier, but over the years there has been feature creep to do an awful lot more. We are going to spend a little time in this unit doing a potted history of the Internet, the Web, Hypertext, DOM, & Basic Web Architecture (Servers & Clients). We'll then build upon, and iterate over this, iteratively deepening our knowledge and skills in subsequent units. Whilst you are reading and thinking, consider the following question, What happened technologically & socially, that lead to the situation we are in now and gave us the Web that we have now? By thinking about these things, and considering the tools that underpin the Web, we begin to construct a framework for understanding how it all fits together so that we can better use and develop our skills later.

vThis specific unit covers a lot of ground and it can be difficult to see how it all fits together at first. We will cover the following topics: Communication, The Internet, Internet Protocols, HTTP, Hypertext, the invention of the Web, HTML, Servers, Web-clients, Browsers, Basic Web Architecture, the client-server model, and the DOM. All are important topics when considering the Web, especially the process of getting a web page from a given Internet server to your web browser There is no single route through all of these topics as, like the web itself, they form a network of interconnected topics which each play their part in that story of getting a file from a remote server and turning it into a page in your browser. Let's consider a few scenarios: For example, there is a historical

story involving how people have thought about communicating ideas. Most ideas are hierarchically organised and related to other ideas, which lead to concepts for arranging ideas as "hypertext".

There is also a computer communication story. For a computer to be able to move (communicate) information from one place to another it requires a network of multiple computers (the Internet) and agreements on how the communication should occur. These agreements are known as protocols. The Web relies on a hierarchy of protocols for things like finding another computer (using addresses call Internet Protocol, or IP, addresses), then transporting messages between the computers once they can find each other (using the Transmission Control Protocol ,or TCP), then because there are lots of different kinds of messages that computers can understand, they need a protocol for communicating web pages instead of other messages such as email data, or instant messages. This is achieved using the HyperText Transport Protocol (HTTP) which is responsible for moving Web data around. But Web data can comprise lots of different things, including the page itself (using HTML), the way the page should be presented (defined using CSS), and any dynamic functionality that the page should have (using JS). There is also an organisational story. Whilst every computer on the Internet has an IP address (is a part of the Internet) not all of them are parts of the Web. Also, different machines play different roles, i.e. in requesting a page, serving a page, and displaying a page. So a server will provide pages to clients who make requests, and requests are made by software called "user agents" the most common type of which is the users browser. The browser and server are organised in terms of a client server organisational model. As you go through the remaining parts of this unit, it can be worth returning to this overview to try to understand how the various parts relate to each other.

So let's start by considering communication. The primary thing that the Web does is to communicate information from one place to another. For a computer to be able to communicate this information usefully, that is so that the computers at each end of the communication can understand what to do with the information, there needs to be an agreement on how the communication should proceed and how the contents of the communication should be structured and understood. This shouldn't surprise you. Think about how you communicate with other people. For most of us, the primary means is through language. Amongst other things, successful communication between people using language involves a shared language, and some common vocabulary and understanding of the domain that the people are communicating about. The advantage that people have is that they have a brain which can solve problems and gloss over the gaps in understanding. Computers don't have this advantage, they need every aspect to be precisely specified in an unambiguous fashion.

The solution, for computers is to use protocols to underpin communication. Protocols are agreements for how to communicate and they closely specify, for everything within the scope of a communication, exactly what is legal, and what is not. This means essentially that a protocol defines what a message, from one speaker to another can consist of. Any message that doesn't fit what the protocol defines is considered wrong and there are various ways to handle such wrong, garbled, incomplete, etc. messages. Computer protocols are agreements that are specified with sufficient clarity and lack of ambiguity that a computer can follow them automatically.

The Internet and the Web are really just communication methods (protocols) - agreements for how two machines can exchange information. The Web itself is just one specific way that machines can exchange information. Let's now explore a couple of these concepts in some more detail...

2.1 The Internet

The Internet is a global system of interconnected computer networks. Over the years there have been many different protocols underpinning computer networks, because what else is a network but a way to move information, or communicate information, between different machines. Over the years some of these protocols have become standard, and others have become obsolete. The modern Internet is currently built on a limited number of shared & agreed protocols. These are the Internet Protocol suite which comprises the Transmission Control Protocol (TCP) and the Internet Protocol (IP). Together these are often referred to as "TCP/IP".

IP is used to govern how machines on the Internet find each other. To do this they use IP addresses, essentially numbers that are formatted as dotted-quad layouts like this: 134.16.46.98

Occasionally you might see IP addresses in your browsers address bar. Every Web site is stored on and "served" by web server software. This software runs on a hardware server which is not only a part of the web but is also a part of the Internet, i.e. every web site is a part of the Internet. Every web host is a part of the Internet. However, not every Internet host is a part of the web. The Internet is much bigger and incorporates functionality that is not part of the web.

Every Internet host has an IP address which is used to find other machines. The exact mechanism for finding machines from their IP address, which is literally about identifying a specific piece of hardware somewhere in the world, is outside the scope of this module, but the correct functioning of IP is essential to the working of the web. Think of an IP address as being analogous to, but not exactly the same as a postal address or a phone number, or a GPS location, a way to identify one entity, to distinguish it from others, and to locate it.

TCP works with but is different to IP. Once you've located a machine, via its IP address, TCP is used to communicate data between that machine and any other machine that has an IP address. TCP is the basic mechanism for transmitting data between Internet hosts and controlling that process. Importantly, if something goes wrong during transmission, TCP has lots of mechanisms for repairing the communication. It is designed to be reliable so that a piece of software can send information, i.e. a message, to another host with guarantees that the communication will either fail or succeed. The sender and receiver don't have to worry about how the data actually gets from one to the other.

An important concept here is that each protocol has its own responsibilities and they are designed to work both independently but complementary to each other. We can't send a message without knowing where it's going, but just knowing the destination doesn't

necessarily tell us how to get the message to that destination. We should also consider that these protocols not only work independently but still in a complementary fashion, but also that they are designed to be layered. That is the current Web assumes that there are lower level protocols which are responsible for finding other Internet hosts, moving data to them, and returning the results. TCP/IP don't need to be the lower level protocols, and others exist, but they are the default that the modern Internet and Web currently use.

The development of TCIP/IP and other networking protocols specific to the Internet dates back to research in the 1960s which was commissioned by the US government. The aim was to build robust, fault tolerant communication via computer networks. To a large degree this has succeeded. The World Wide Web (or just Web) is just one information resource/service that communicates using the Internet (although people often use the terms interchangeably).

2.2 Internet Protocols

In the last section we discussed how the Internet and the Web are built on the idea of layers of protocols. Formally there is a seven layer model called the OSI model that defines all of these layers and their various responsibilities. We don't need to know about all of them for this module so will concentrate on just those necessary to understand how the Web fits into the idea of the Internet and networks.

Let's explore this a little further. On the lowest level that we'll consider here is the link layer, think of this as both the physical infrastructure, such as your WiFi or ethernet cable. It is concerned mostly with reliably moving information from place to place. Although this seems similar to TCP above, it is much lower level and the main difference is that the link layer doesn't care about how that information is broken up into packets or reassembled, it only cares that the stream of data it is given is reproduced at the other end. In comparison, TCP takes a particular approach to breaking up the data to transmit into packets, and uses specific algorithms to reassemble those packets into a whole at the other end, and also to identify and retransmit lost or broken packets. A similar protocol that works at the same level as TCP is the User Datagram Protocol (UDP) which doesn't provide the same guarantees on message integrity that TCP does. However the trade-off here is that UDP can be more efficient and faster to communicate information, with all of the risks associated with a send and forget communication protocol. We have these different protocols to enable us to use computer networks to communicate different information in different ways. Perhaps consider it as analogous to the difference between the different ways of physically sending messages around the world. We could write a postcard, or a letter in an envelope, a package or box, right up to a shipping container (and possibly other mechanisms as well but these are fairly standard).

As we talked about before, once we have two networked machines (link layer) that can transmit data, they need a mechanism that enables them to address messages to specific machines, which is the role of the Internet layer and IP. Once we have a machine that we can identify individually we can use the transport layer (TCP) to move data in a specific way to the destination. It is only at this point, once we have the link, internet, and

transport layers that we look at the application layer. It is this layer at which the Web protocols operate. The HyperText Transport Protocol (HTTP) is one example of this that we will investigate next but there are lots of other applications layer protocols which govern other tasks like email (the IMAP and POP protocols) or turning domain names (web addresses) into IP addresses so that, given a web address or URL (different names for the same thing) we can tell the IP protocol which machine we want to communicate with. Although people generally think of websites in terms of domain names, our computers only care about IP addresses when it comes to finding a given web server.

To summarise, we have:

- Link Layer: Ethernet/Wifi
- Internet Layer: IP
- Transport Layer: TCP
- Application Layer: DNS, HTTP, IMAP, POP

2.3 The HyperText Transfer Protocol (HTTP)

Just a little more on protocols before we consider some other aspects of the Web. The HyperText Transfer Protocol (HTTP) is our application layer protocol that is critical to facilitating the Web. Other protocols like TCP and IP don't care that the data they are transmitting or the machines between which they are moving the data are Web machines or not. Those protocols can transmit lots of data for different reasons and many of those reasons are not related to the Web. When information arrives at a machine and is routed to software that understands HTTP it is treated as Web data. HTTP is concerned primarily with moving hypertext between Internet hosts. We'll consider what we mean by hypertext in the next section but for the moment we can think of it as being text documents. Although the modern web can manage to handle much more than just text the default is still for HyperText Markup Language (HTML) documents. HTTP is concerned with moving these documents around, requesting particular documents and processing the responses, and delegating the actual movement of the document to lower level protocols along the way. Note that HTTP does much more than just merely governing requesting documents and responding with the correct document. It also has the mechanism for responding in different ways depending upon the request made, or how successful the request was. It is this additional functionality that enables rich, server generated behaviour and on the fly page creation, but these aspects of HTTP are outside the scope of this module, although you are encouraged to do some additional background reading if it interests you. A good place to start is with the W3C specification if you want the gory details:
`https://www.w3.org/Protocols/`

HTTP is an application layer protocol for distributed and collaborative hypermedia/hypertext systems. It is based upon a request-response protocol that uses the client-server model. We'll investigate the client-server model in a little while but the basic idea is that the Client (i.e. your Web browser) makes a request. This is probably due to the user making a choice such as clicking on a link or typing in a web address. This request

gets passed down the layers of the networking stack, transmitted to the remote machine, and heads back up the networking stack until it reaches the server software that understands HTTP. Mostly though we can ignore the full stack and just consider that there is an HTTP server (software running on an Internet connected computer) that listens for requests and responds according to what the protocol allows. Any other response is problematic and will cause some form of error or failure in the communication.

The HTTP Server stores resources and returns a response that may include providing access to those resources. For our purposes in this module those resources should be considered to be the things that you'd commonly expect a web site to provide, e.g. HTML pages, associated files in the CSS and JS languages that the HTML pages reference, and any other media like images, videos, or audio that the pages want to make use of.

These request-response cycles occur as part of an HTTP Session. This is a sequence of request-response transactions transmitted over TCP. One thing to note at this point is that Internet hosts can listen on different ports. The default port for the Web is port 80. In fact it is so common that most browsers don't display that the address it is connecting to is listening on port 80. A web client can connects to any legal port on any legal IP address. HTTP servers listen on that port. Whilst the default is 80 you might occasionally see other ports in your address bar, for example 8080 which is also reasonably common but some development servers run on other ports like 5000. If you see other ports, don't worry.

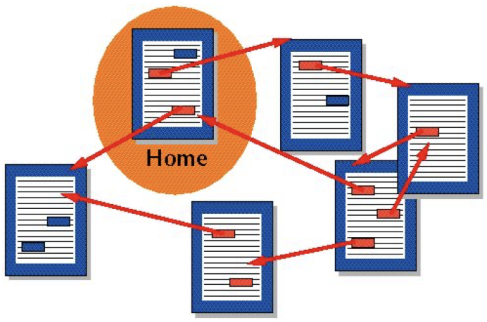
HTTP requests are more than just a simple request to retrieve a web page, They can also involve lots of other request Methods. These methods are usually referred to as HTTP verbs, e.g. GET, HEAD, POST, PUT, DELETE, OPTIONS, and PATCH to name a few. These verbs are essentially actions that the request is asking to be performed. The most common is HTTP GET. This is asking to retrieve or "GET" the resource (web page or HTML document) that is being asked for and return it to the client. A client makes a request and the requested resources is transmitted back to the client. What is transmitted between the client and server though is just plain text. HTML web pages are plain text, CSS files are plain text, and javascript files are plain text. They are readable and understandable by people. If you were to eavesdrop on the connection between client and server you would see text passing by containing the HTML, CSS, or JS content of the files that are being requested and then "served" up. Note that modern versions of HTTP will retrieve binary files, e.g. for audio and video, but the default is for plain text, a simple and robust encoding mechanism that is easily inspectable without specialised tools. It is this focus on plain text file, the same kind of files that any text editor can produce, which is arguably a part of making the Web so successful. Anyone with a text editor can produce the files that will make up a web page. If they then put those pages on a web server, then voila, you have published a web site.

Before we finish up this introduction to HTTP, note that by default HTTP is insecure. The transmission of information is in plain text and any one machine between source and destination could intercept or manipulate it. This risk has largely been mitigated by the use of secure variants of HTTP, such as HyperText Transfer Protocol Secure (HTTPS) & HTTP Strict Transport Security (HSTS). Browsers are slowly moving towards defaulting to HTTPS addresses. The main difference between HTTPS and HSTS is that HSTS is

more strict about how it deals with mixture of secure and insecure resources on a single page. HSTS mandates for a given page that all resources on that page are served securely. This is an example of the slow improvement of Web and Internet protocols.

2.4 Hypertext

In the last section we discussed HTTP, a protocol for transferring hypertext resources between Internet hosts, but we didn't really define what we meant by hypertext. So let's square that away right now. Hypertext is a term that was coined byTed Nelson as part of the Xanadu research project sometime around 1965. This was as part of a model for linked content, the idea that a concept is related to other concepts and that it would be useful to create links between concepts within and between documents so that instead of reading a page, and then reading another page in linear fashion, as we traditionally read books, instead it is useful instead to be able to follow the thread of related ideas wherever they may take us. The following diagram illustrates this idea schematically, starting with the home page highlighted in orange, individual sections of text are linked to other pages which, in turn contains links to other pages, and so on.



Ted Nelson didn't invent this idea though, an earlier incarnation which was visionary but came before the technology was capable was in an article by Vannevar Bush in 1945. Bush proposed the Memex, an attempt to turn this idea for turning information into a searchable and interlinked database of knowledge, however he was limited by the capabilities of 1940's technology. It is well worth reading Bush's article which can be found here: `https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/`

But it is arguable that Bush didn't really invent this concept either. Before the Memex proposal we already had books that work in a similar, albeit manual and non-computational way. Dictionaries, Thesauri, and Encyclopedias also work in a similar way, referencing other sections and entries within themselves, other volumes in the collection, and other books. Encyclopedias in particular are interesting to consider because they often have summary books and indexes that give different ways to approach the articles contained within the encyclopedia proper.

A modern incarnation of hypertext that is quite well defined and circumscribed is Wikipedia. Consider how you start on one Wikipedia page and then follow links to other pages until you've lost an entire afternoon discovering things you never thought that you'd want to know. This is hypertext. Text is displayed on an electronic device

that incorporates references, called Hyperlinks, to other text. These Hyperlinks can be followed or navigated immediately. By doing this, text becomes non-linear as a result - instead of one page following the next we can jump between them, consuming them in whichever order makes most sense.

Note that the modern web goes beyond just links between text and other forms of media are often consumed on the Web now so you will often find the related term hypermedia used. The terms hypertext and hypermedia are frequently used interchangeably.

One implementation of the hypertext idea is in HTML. Whilst the bulk of HTML is about describing or "marking up" text documents one part of HTML concerns defining links within and between documents. Even just getting this far into the module, you'll already have navigated a whole bunch of hyperlinks as you've "browsed the web". Browsing the web is just another phrase for following hyperlinks or using hypermedia.

So the Web itself is an implementation of a hypertext system as well. That said it isn't the only possible implementation of a hypertext system, it's just the most prevalent one that we have now. Perhaps we should now consider how we ended up with the particular implementation of a hypertext system that we call "the Web"?

2.5 Invention of the Web

The Web, or more formally, the World Wide Web (WWW) is an implementation of a hypermedia system. It was invented in 1989 by Sir Tim Berners-Lee whilst he was working at CERN. This involved developing HTTP to move documents around and HTML to structure and define documents themselves, with embedded links between them. The first web browser was written in 1990 but it was 1992 before there was really a usable and publicly accessible web browser (but which was very different to what we have now).

Berner-Lee's original aim was to build a system for sharing scientific research, e.g. amongst physicists. He was, after all, a scientist and the two core function of scientists are to find out things, and then, usually, to share those discoveries. Obviously weve come a long way since then and the Web has moved far beyond its original scientific information sharing ideals.

The early Web was initially formatted text only, but rapidly moved to support images, audio, video, and other media types. This was driven by user need. Rather than a top down, well organised, grand plan for the Web, instead open protocols were created and shared. These are open because anyone can implement their own version of, for example, a web server, or a web browser, and because of the availability of the specifications for the protocols, these implementations are, generally, inter-operable. This means that my im-

plementation of a web browser can understand the pages returned by your implementation of a web server.

More information about the early days of the Web, and some resources such as the very first published web page can be found here:
`https://home.cern/science/computing/birth-web`

Given that we've spent a fair amount of time considering what the Web is, where it came from, and how it built on lower levels of networking infrastructure, perhaps we should now change tack a little bit. We've concentrated so far on how "web pages" are moved around, and the networking protocols needed to do this, but what actually is a web page? That's our next topic....

2.6 HTML

A web page is simply a document that can be retrieved from a web server. However, our browsers do more than merely retrieve documents and show them to us as plain text. We are used to web pages generally being perhaps quite colourful and often well presented (although the converse is also true). It seems that there is more to the Web than merely retrieving documents from the web server. It seems that the document itself is actually structured in a particular way and that the browser turns this structure into a particular presentation (perhaps with the aid of CSS and JavaScript as we'll see in subsequent units). It turns out that web pages are written using a particular language called the HyperText Markup Language (HTML). HTML is simply a language for turning text into structured hypertext using markup. Structured because it incorporates semantic information such as that a given sentence be treated as headings or paragraphs, or lists, and hypertext because it enables links to be defined between sections of text and other locations such as places within the current document or within other documents. HTML is called a language because it is a means for communication. There are many kinds of language, e.g. natural (e.g. English or Spanish) & artificial (e.g. programming languages like Python or Javascript). There are also formal languages like HTML which are not general purpose programming languages because they lack the facility to handle things like variables or expressions but are better described as domain specific languages, languages that are devised to do something well in a given domain.

We'll look at HTML in more detail in the next unit but for now we need to know that HTML handles the domain of text. That is strings, or sequences of characters, that are encoded using an agreed format. Generally that format is UTF8. That is where the T in HTML comes from. So far I've glossed over what we mean by markup so let's put that straight right now. There are many ways to do markup and these are not specific to HTML and use a variety of techniques. The approach taken by HTML is to use Tags. HTML tags, or just "tags" are generally placed around the element being tagged, e.g.

```
1    <h1>Hello</h1>
```

This places the <h1> tag around the word "hello". Tags usually work in pairs to encapsulate a section of text so we indicate the closing tag with the '/' character. This

just helps to pair them up so we can see where the encapsulated elements starts and ends. The angle brackets '<' and '>' indicate that they enclose a tag. In this case the specific tag is the h1 tag which is the name for heading level 1, which is basically the largest level of pre-defined heading. We will see this and more in the HTML unit but it is enough for now to realise that there are a lot of tags that pre-define, or capture, different aspects of the text. Think of all of the documents you have ever read or written. Many will have similar elements like a title, various headings and sub-headings, paragraphs, and other typographical aspects. HTML tries to capture these so that the browser can process an incoming document and treat differently tagged parts of the HTML document in different way, perhaps by visualising and depicting the elements in various ways or by making different functionalities available, e.g. a hyperlink can be clicked upon to automate the process of navigating to the destination that the hyperlink points to.

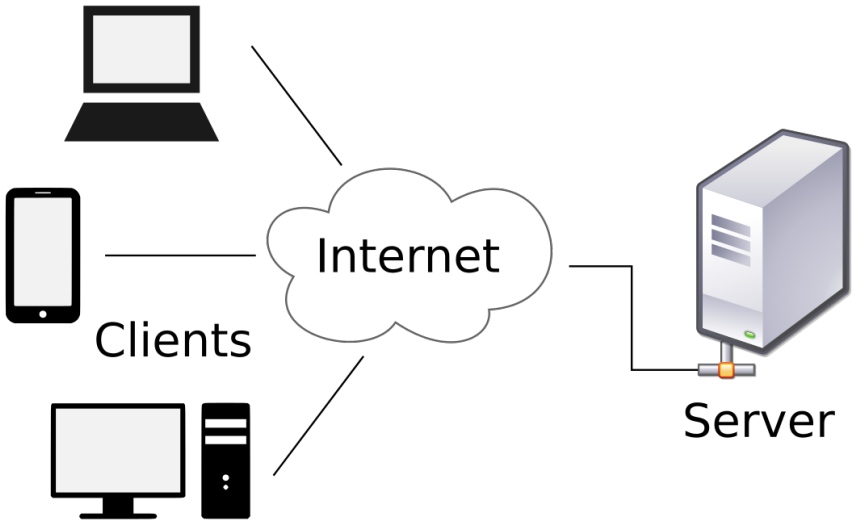
If you want to read ahead before you get to the HTML unit then it is well worth checking out the Mozilla HTML elements reference to see just what is available: `https://developer.mozilla.org/en-US/docs/Web/HTML/Element`

Let's just consider the journey we've taken so far. We've considered the various protocols involved in putting a document on a server on the Internet so that it can be requested and treated as a Web page using HTTP. This involves lots of lower level protocols that are concerned with moving information reliably around the Internet. It's not just about moving the document around though, it's also about being able to understand the structure of that document and present it in an appropriate way using HTML. The final thread here is that hyperlinks within an HTML document present a shortcut to initialising the request and retrieval of further documents, via HTTP, so there is a functional and symbiotic link between HTTP and HTML. The next few topics are now going to round things out by considering how clients and servers are related to each other and, more importantly, what we mean by a "web server" or "web client".

2.7 A Basic Web Architecture - The Client-Server Model

All of the topics that we've considered so far have lead us to this point. We now have all of the pieces that we need to consider what we mean by a "client-server architecture of the web". Understanding this is important to knowing what is happening when you develop a web page or site and have multiple resources. A good developer will understand where their resources are located, how they are transmitted to the client, and when that occurs. This will help us to build a useful conceptual model of where our code is, whether HTML, CSS, or JS, when it is at rest and when it is executed, and the difference between each.

In one sense the web architecture is very simple, there are web servers which listen for requests. They don't initiate communication but merely respond to incoming requests. Requests are made from clients. These can be browsers, or any other software, that speaks HTTP and can correctly formulate an HTTP request. Similarly a web, or HTTP, server is a server that speaks HTTP and can correctly respond to incoming HTTP requests. Clients make requests and servers respond.



Web clients are merely software that communicate to a Web server by making requests. The communication is governed by HTTP. Web clients are often also referred to as user agents. A web browser is a particular type of user agent which acts on behalf of its user, i.e. acts as an agent, to satisfy their requests. There are other user agents, for example, command line tools that retrieve web page or other web resources using HTTP, but that don't render the results to the screen. For example, a web scraper or web indexer, is a user agent which collects web resources but does something else with them like, perhaps, building a database with the results. The reason we talk about web "clients" as well as user agents is because of the link to the client server architecture.

To summarise, servers are software that responds to communications (HTTP requests) from clients (browsers).The response generally contains a document (HTML) that is transmitted back to the client as part of the response. Note that it can get (much) more complicated than that but this is a good starting place.

2.8 Servers

The term server is heavily overloaded in computing. It can refer to hardware and it can also refer to software. Generally though we consider a server, whether hardware or software, to be acting in a particular role. That role is "serving", providing, or distributing data. Server software may run on server hardware, or might run incidentally on other hardware, for example, your laptop during development. Generally though we'll ignore the idea of server hardware in this module and just assume that on some level, all of our software is running somewhere appropriate. Server software isn't limited just to Web contexts though. For example, many database management systems work by using server software which responds to client requests. However a database, rather than speaking HTTP and HTML might instead use a query language like SQL or SPARQL. That said, some database servers also speak HTTP, an example of this is CouchDB which provides an API which can be accessed via a web browser. Whilst it's a little outside the scope of this module, it is worth considering just how many pieces of hardware out there could incorporate an HTTP server to provide a user interface to any browser rather than needing

a specialised application. This is quite a democratising way to consider things and is especially exciting when we consider the idea of Internet of Things (IoT) and the notion that not only can the world provide data about what is happening to us, but things in the world can give us Web-based ways to interact with them.

A server is, on the simplest interpretation, merely a piece of software that runs on a computer. It listens for messages and determines the right response to make (where the determination of request and response pairing is defined by the protocol).; well l assume a server running on an Internet connected machine (so using TCP/ IP as lower level protocols). A web server is thus a server that listens for messages that are sent using web protocols (HTTP). By extension an email server is a server that listens for messages sent using email protocols (such as SMTP, IMAP, or POP).

We've established that servers are responsive, they respond to something speaking to them by sending a message. If a server is listening then what is doing the speaking?

2.9 Web Clients & User Agents

In the context of the Web, the software doing the speaking is the web client (or user agent). This is just another piece of software (nothing particularly special). Again, a web client just happens to speak HTTP but can initiate requests rather than merely waiting for them to arrive from other web clients. Other than this, a web client doesn't have to do anything more. It is useful to do something with the result of a request though. The result, the response, could just be logged, for example, the server responds with 200 OK. This would be enough if we were creating a web client that merely checked whether an HTTP server was alive and running correctly. A web spider (or search indexing agent) might retrieve pages from a site, then programmatically follow all the links to retrieve the pages that it links to and so on until it runs out of links to follow. Note that this is only a naive description of how a web spider might work. The spider would then do something else with the retrieved pages, perhaps building a database or analysing the page content for particular features. By contrast a web browser usually retrieves a single page then interprets the HTML content of the page to construct an internal, programmatic, representation of the page. This internal representation is the Document Object Model (DOM) which we will see soon. This DOM representation is then turned into, usually, graphical representation of the document. However, there are also text based browsers which manipulate the retrieved HTML document in a different way because they aren't visually based but are text oriented. Similarly a browser for the visually impaired might present an audio version of the page. The message to take away is that there is a lot more to the functionality of a web browser than merely retrieving pages and showing them to the user like a form of Internet PDF reader. Mobile apps are an increasingly popular web client. Frequently they will be retrieving data from a remote HTTP server (often as JSON documents rather than HTML documents) and then turning that data into a native application specific to the data retrieved rather than using a generic web browser. Essentially though, it is very easy for almost any piece of software to become a web client, it just needs to access, consume, or display web content, e.g. have the capability to talk usefully to an HTTP server.

Let's now consider a specific type of web client, the web browser, possibly the most visible item of "web software" in our next topic.

2.10 Web Browsers

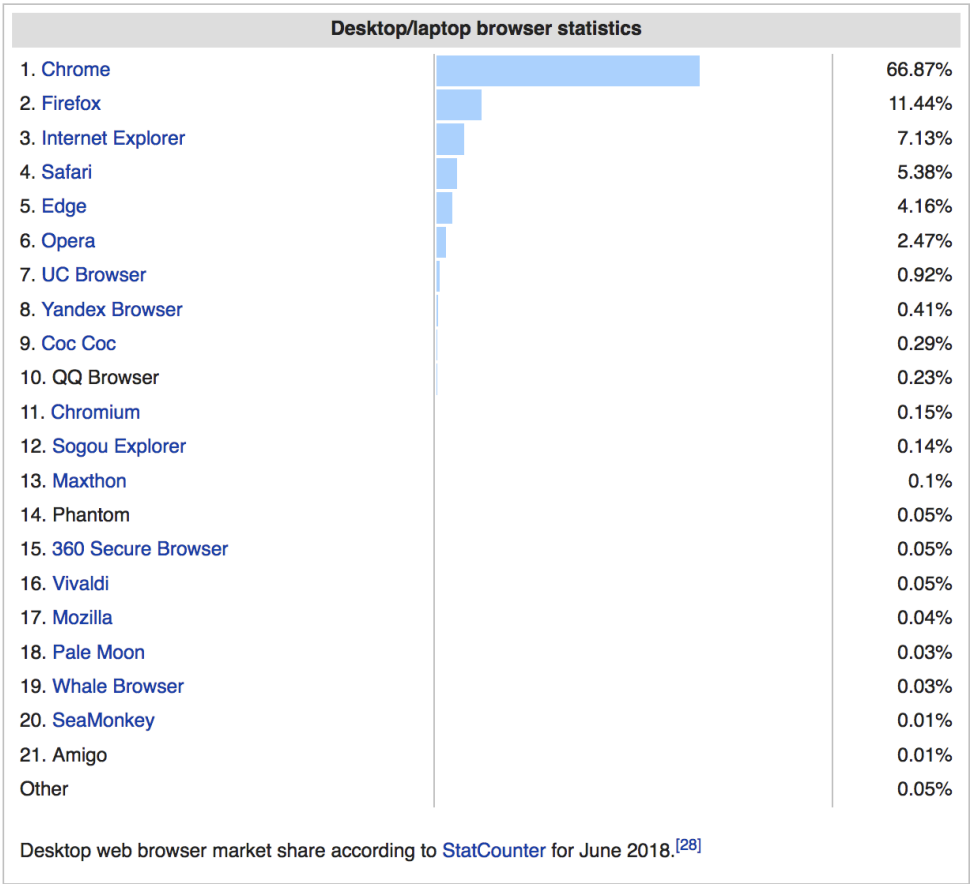
Web browsers are software that not only speak HTTP so that they can retrieve HTML and other related documents from a web server but usually also contain some form of a layout engine that renders the web pages (HTML) into an appropriate form. This appropriate form is usually but not limited to the graphical web pages that we are used to from everyday web usage.

The web browser is yet another invention from Berners-Lee. Having defined HTTP and HTML there was obviously a next step to create a user agent that could retrieve HTML from HTTP and then display it appropriately.

Browsers are generally used to interactively navigate the public web but are also frequently used to access private networks, IoT interfaces, local file systems, and much more.

Browsers have also become a default cross-platform environment. Many items of software that might previously have been written as standalone desktop clients are now implemented as "web apps" that are generally accessible from any given web browser. This can solve many problems involved in the traditional development and distribution of software. For example, a new version of the software only has to be published to the web server and when the browser refreshes the page the new software is deployed. No more downloading, double clicking and installing or package managers required. However this is at the expense of different behaviours between various browsers which can affect user experience. There are also different performance characteristics when dealing with browsers compared with native applications. Also, browsers have been designed primarily for displaying web pages, which are essentially documents, with a particular flow of information and internal structure. This is not the same as a general graphical interface toolkit as might be used to develop for the desktop.

There are many different producers of web browsers and the illustration gives you some idea of the prevalence and popularity of the more common of the various browsers. Note that this isn't entirely accurate because many browsers can identify as different user-agents, e.g. Firefox describing itself as Chrome, usually for compatibility. This is behaviour that is user-controlled and leads to the statistics on browser usage being a little uncertain.



2.11 The Document Object Model (DOM)

We've made great progress now and have examined a lot of the technologies that are involved in retrieving a document from a web server and getting it into the browser. But what happens next?

The HTML obviously needs to be altered from its default text based .html file into something that can be translated into the graphical display we are used to seeing in our web browsers. This is achieved by creating an intermediate model, named the Document Object Model (DOM).

The DOM is a cross-platform, language independent Application Programming Interface (API). It parses the incoming HTML document, treating it as a tree data structure internally within your browser. The tree is formed due to the hierarchical and encapsulated nature of HTML tags where everything is enclosed within ¡html¿ tags at the top level. The ¡html¿ level encapsulates ¡head¿ and ¡body¿ tags which in turn encapsulates other tags, and so on, until everything is in place. This is heavily dependent upon the specific tags found in any given HTML document. HTML is parsed into this data structure to construct the DOM (for that document). Each part of this tree is represented internally to the browser as an object that can have various attributes and methods. These methods can then be called to manipulate the model, a process often referred to as manipulating the DOM.

The methods associated with the DOM provide an API that JavaScript can interact with to manipulate the current page. For example, Objects can be manipulated programmatically, e.g. using Javascript, and the results displayed in the view pane of the user agent (browser). The browser also provides other APIs for interacting with the environment from JavaScript but the DOM is a good place to start.

It is also worth noting that the DOM and the HTML sources are not the same. You cannot rely on the HTML that you loaded initially being a good representation of what you see in the browser or what is represented internally in the DOM. When the HTML is parsed it is used to construct the DOM, but this model can then be manipulated by any linked JavaScript and also manipulated visually, or even hidden, through CSS. If you right click on a page in your web browser and select "view source" then you will see the original HTML that was retrieved from the web server. If instead of view source, you load the browser development tools then you can use these to inspect and manipulate the DOM, if you compare the two then you might notice that the DOM is not always identical to the source HTML.

You can find a little more about using the Chrome dev tools to inspect the DOM here: `https://developers.google.com/web/tools/chrome-devtools/dom/`

We will return to consideration of the DOM throughout the HTML, CSS, and JS topics in subsequent units. An awful lot of client-side javascript involves manipulating the DOM.

2.12 Browser Developer Tools

Our final topic for this unit is to consider turning all of this theory into practice. We only really need a good editor and a good browser to get started with client-side Web development - Web IDEs are often overkill for simple development.

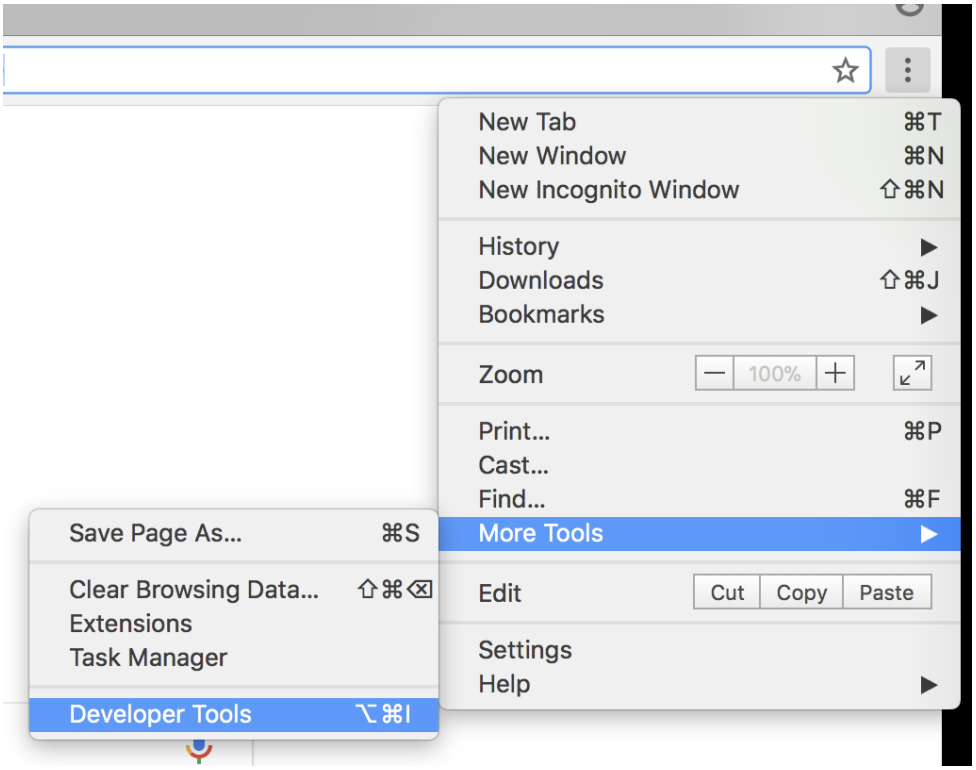
There are many browsers: Chrome, Chromium, Firefox, Safari, Opera, IE/Edge - but for this module we will focus on Chrome so that we all have a consistent development and learning environment. I'd still suggest having other browsers installed though so that you can compare behaviour between them.

All newer browsers have support for some set of developer tools. Common Features of Developer Tools include the following:

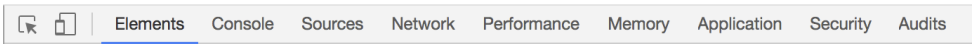
- HTML & DOM viewers & editors
- Web page assets, resources, network information
- Profiling & Auditing
- JavaScript Debugging & Console

Chrome is currently very popular but it also pioneers many new ideas and directions for the future of the Web.

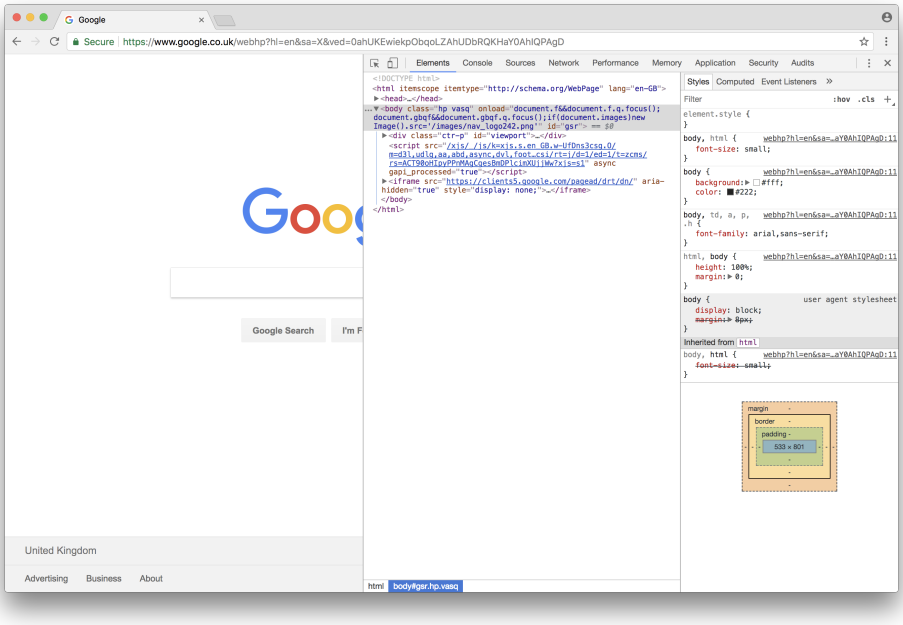
It is fast, stable, and feature rich. It also has many tools to support developers - get access to the surface (the web page) but also the internals of the browser and the web page/application. We can launch the developer tools by following the procedure in this screenshot:



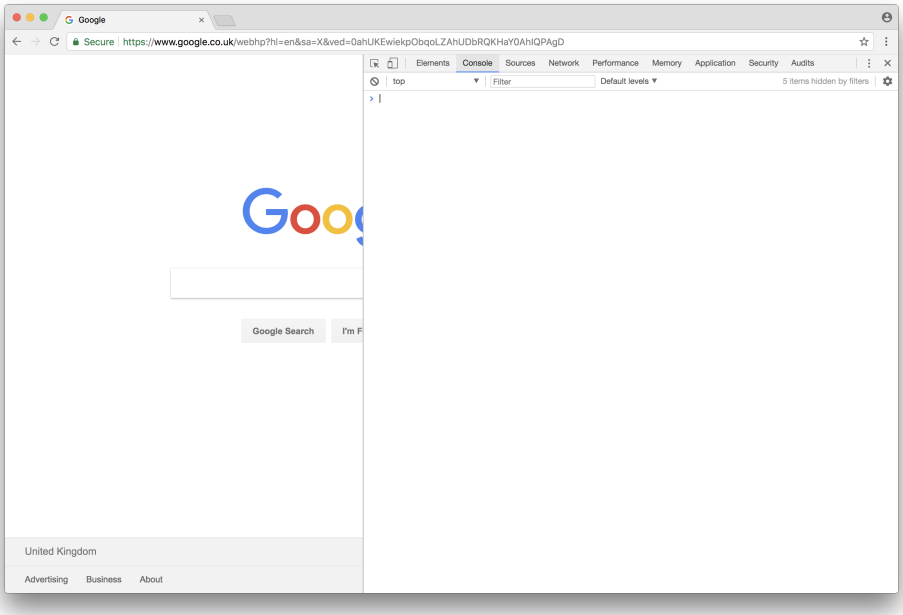
This gives us the following new set of tabs to interact with:



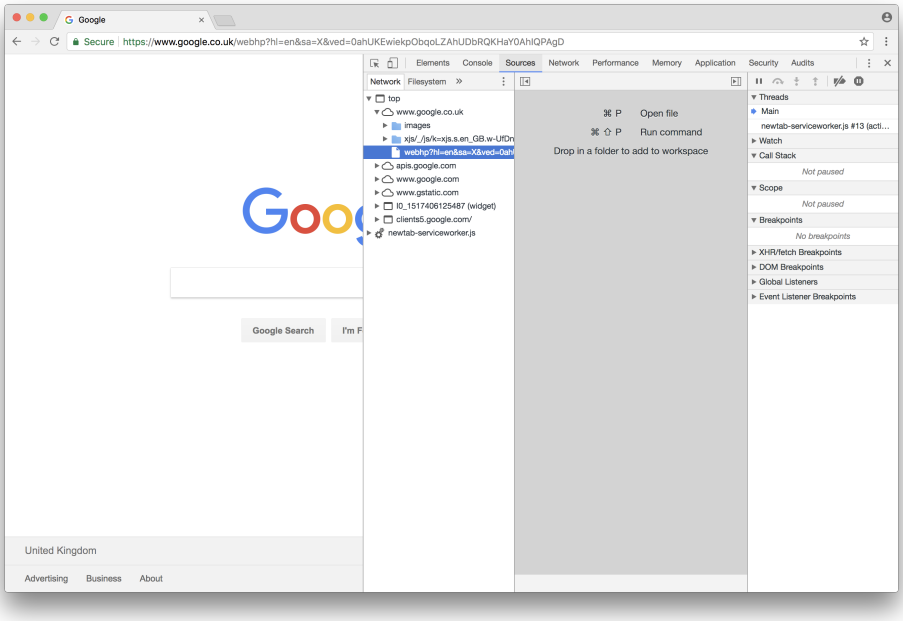
For each tab we get a different view on aspects of the page that is loaded, for example the "elements" that is the DOM representation of the underlying HTML that the page is made up from. Note that this can become quite complicated because any associated CSS and Javascript can manipulate the HTML elements. As a result the HTML that is rendered in the browser for any given web page that you are viewing can be significantly different to the original HTML source file that was loaded. Remember this, CSS and JS give you great powers to manipulate a page, you can even discard all of the original HTML and completely generate a new replacement page.



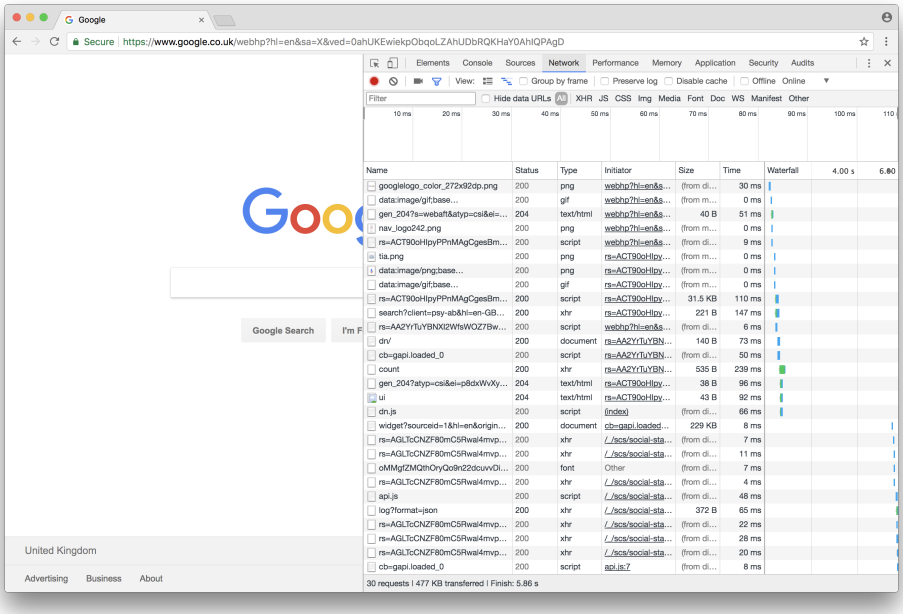
The next tab is for the console. This is a place that you will need to look at frequently as you develop your sites as it is the location where errors, warnings, and messages are printed when things go wrong. You can also use it as a location to execute your own javascript. Just open the console and write some code then when you press return it will be executed. When you write more extensive JS in subsequent units you will find the console is a useful and important tool to have ready in your toolbox.



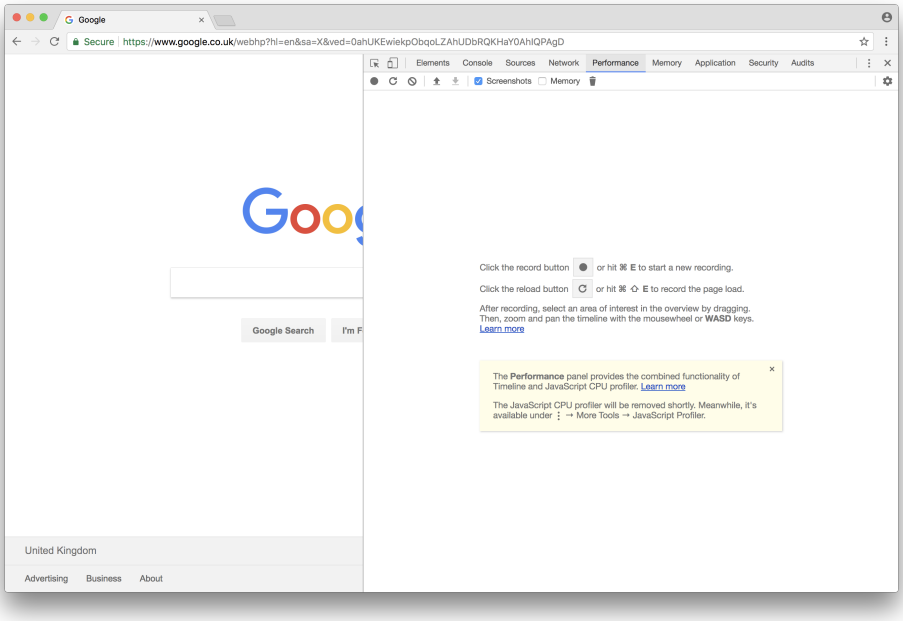
Next is the sources tab which will come in useful when we are dealing with deployed sites as it gives us a way to track all of the different resources that our pages use:



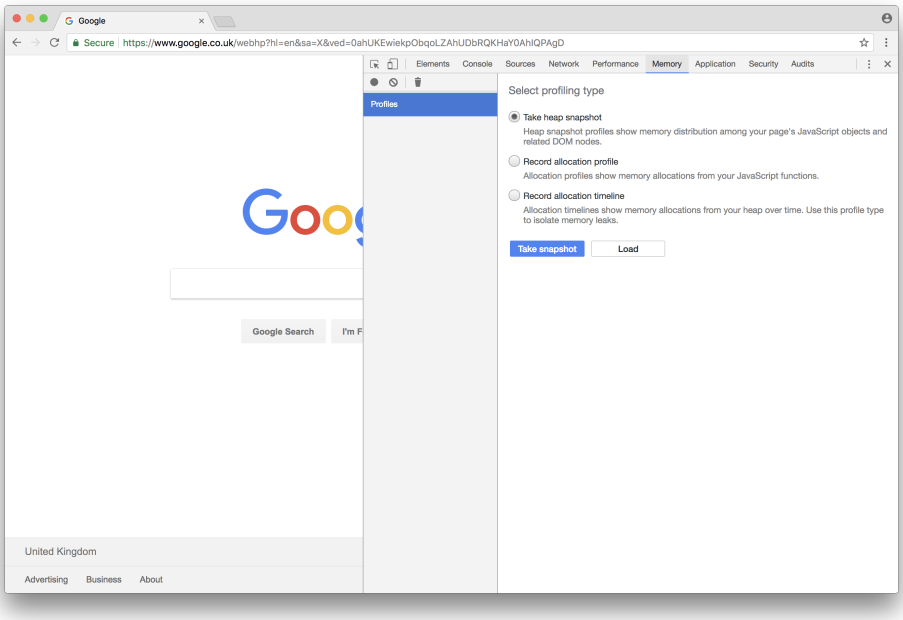
Our next tab, the network tab, gives us information about data and files being moved over the network/Internet/Web between your browser tab and all of the resources that the current web page is built from. This can be very useful when we build more complicated pages and need to debug why something isn't quite right.



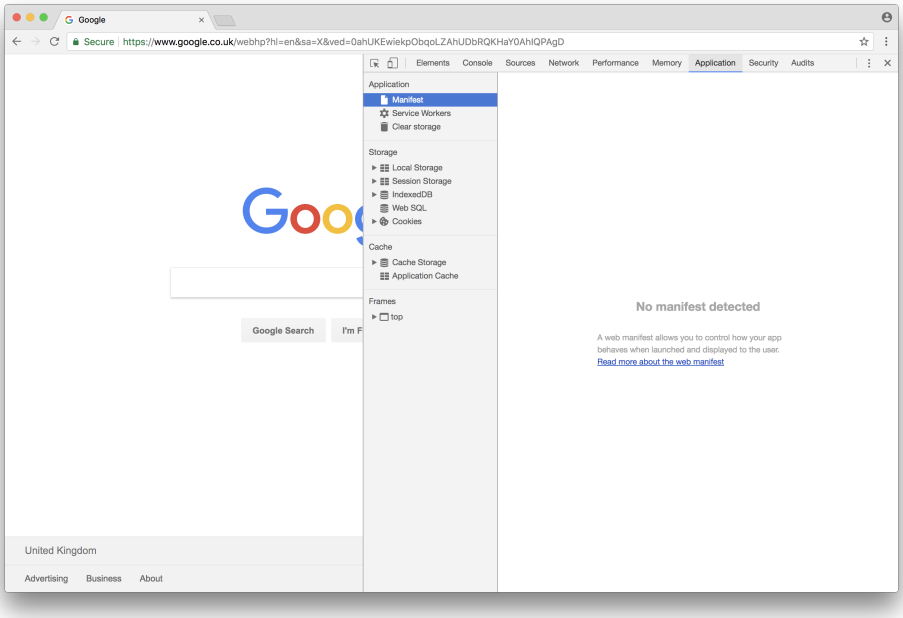
Next we have the performance monitor which we can use to inspect how well a page is working:



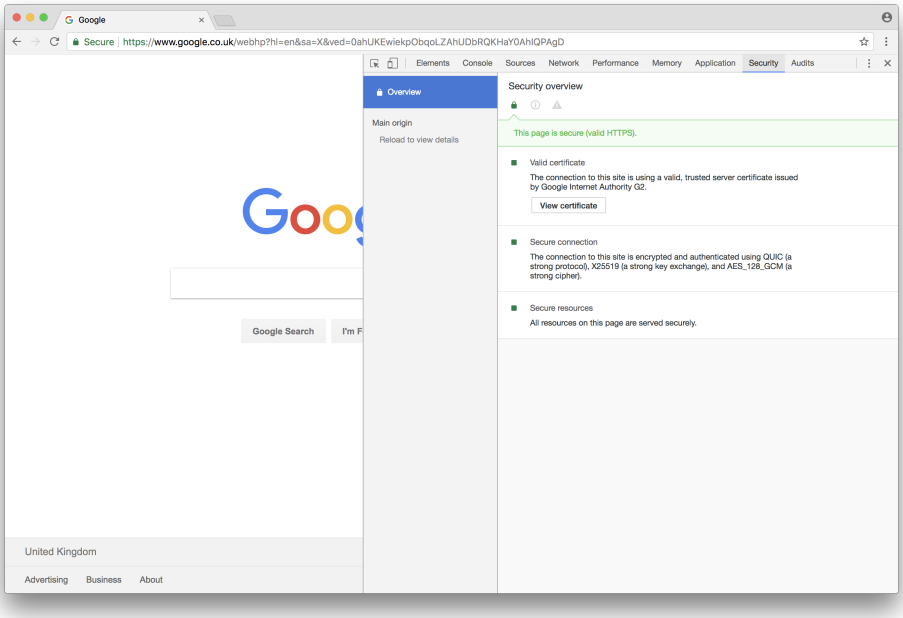
We can also profile memory usage which can be useful as start to write more extensive Javascript:



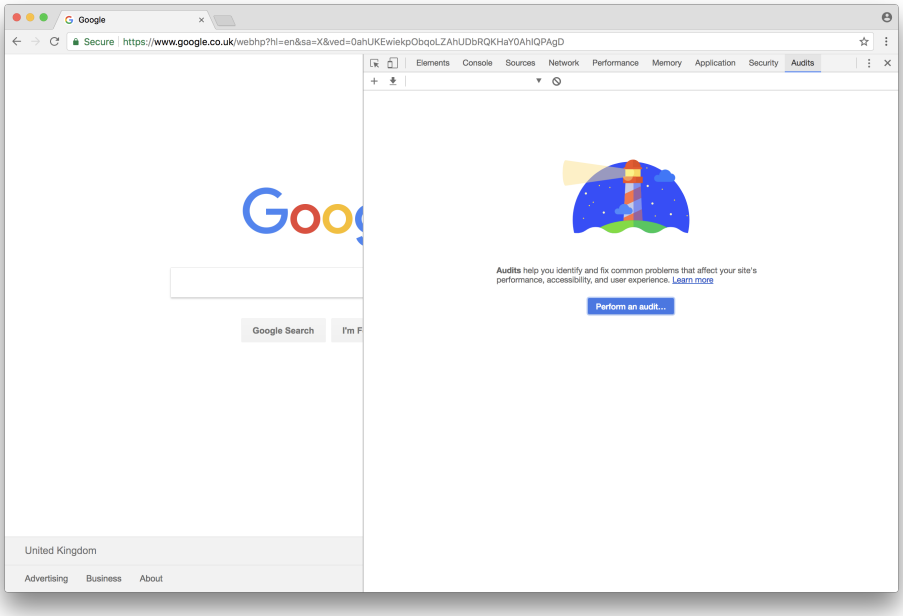
We can inspect aspects of the current web application, e.g. how local storage is being used, and which information is being cached:



We can also inspect browser related security features:



Finally, but no less important, Chrome supports an auditing feature using the Lighthouse project. This is a really useful tool for improving the quality of your pages. It audits against metrics for performance, accessibility, progressive web apps, search engine optimisation amongst many other aspects.



Throughout the remainder of this module, the browser tools will provide you with extrememly important insights into what your browser is doing at any given time. As your sites start to become more complex, you'll increasingly rely upon these tools to help you to track down the sources of bugs.

2.13 Summary

Basically, before we can go anywhere, we should really know how we got to our current location. In this unit we've surveyed the following:

- A potted history of the Internet, Web, Hypertext, DOM, and Basic Web Architecture (Servers and Clients)
- What happened technologically and socially, that lead to the situation we are in now
- Overview of the basic things we need to know to understand how it all fits together

We've also looked at the Chrome Browser features that we can exploit as a "development and learning environment" for this module.

It may well be the case that you already knew about clients and servers, or HTML, CSS, and JS, or any of the other things that we've introduced in this unit. However, you should remember that this is only scratching the surface and is meant to bring us all to the same starting line, ready for the rest of the module. In all honesty, the complexity of the Web and its associated technologies, and the rapidity with which they are developing, means that there is far more to discover than it is possible for most people to know. Hopefully you've at least been introduced to some things that you hadn't thought about before. But if you are aware of all of this, then that is your cue to delve even deeper and to develop a thorough professional understanding.

In summary, having read through all of the materials so far, and completed the practical work you should now have some idea of just how many different technologies interact to give us the Web that we know and love. You should also have some idea of the reasons why the Web that we have is the way that it is because of the incremental historical development of Web technologies over time, responding to user need. Finally, you should now have the minimal development environment that we'll need to progress through the rest of the units. Most important are the Chrome Browser, so that we have a default learning environment for everyone involved in the Module (which importantly incorporates useful debugging and development tools), an editor that can save plain text files (with .html, .css, and .js endings) so that we can write the source files that will make up our web pages as we progress.

Chapter 3

Hypertext + Markup + Information + Semantics = HTML

In this chapter we will study our first web language, the HyperText Markup Language (HTML). During our study we will consider the various historical versions of HTML and their impact upon the modern web, then focus on HTML 5 and the production of modern semantic HTML as the core of our websites.

HTML is one of the three core technologies of the Web,the other technologies being Cascading Style Sheets (CSS) and JavaScript (JS). We will cover all of these technologies during this module, but need to start somewhere. All three technologies are languages but they each target a different aspect of the Web, they each work together in different ways, and they each have different powers in terms of what they can be used to do and what their limits are.

Of the three technologies, the best place to start is with HTML. This is because every site has some underling HTML, even if this is subsequently discarded and entirely replaced with a site generated on-the-fly using JavaScript. There is always a minimal page that hosts the other languages and gives the browser a starting place for deciding what to display. For example, open a new, empty, browser window and then use the browser developer tools to inspect the page that is displayed. The browser gives you a default empty page. This empty page is described using HTML. We can conclude therefore that HTML is both critical and central to the functionality of the Web.

In the last unit we talked about the idea of HyperText and how this facilitates links between pages. In addition to this we also need to consider that links without content don't give us much, our pages need to have some content to display that the links can depict relationships between. One way to describe page content is through markup, the use of tags to "mark-up" or delineate sections of the page, such as a word or phrase or other string of text, that should be treated differently from the rest of the page. What is it that is marked-up? It is content, or information. Finally what is it we are trying to convey through web pages. We are not merely distributing information, there are other formats we could use if it was just about sharing information, e.g. database formats and file formats, but web pages try to convey content that has meaning, content that has meaning in the context of the other content that surrounds it. When we talk about meaning we often use the term semantics.

So if we put these four aspects together, we get the title of this unit:
Hypertext + Markup + Information + semantics = HTML

3.1 HTML & The Web

HTML is generally stored on Web servers and processed by Web browsers. You should consider this in the context of our overview of the Web and networking concepts in the last unit. This is not the only way to use HTML, but it is probably the most common at the moment.

The Web was designed for ease of publication - a non-programmer, such as a physicist or other research scientist should be able to develop and deploy Web-sites and Web-pages without needing to write (much) code. Technically, writing a Web page using HTML is a form of coding, you are using a language to tell a piece of software what to do. However, this language is so restricted and focused so carefully on its target, representation of documents, that it is usable by a wide section of society. Think about it, most literate people are able to identify a heading from other pieces of text, or to pick out a paragraph or other document-oriented organisational unit. This is all that people really do when they are making an HTML document.

It helps also that browsers are quite good at taking malformed HTML documents and repairing them, or at least doing the best they can to display the content. This can mean that the content doesn't quite appear the way the writer intended, but at least it can, usually, be displayed. To achieve this browsers have traditionally been very accommodating in what they will accept, not only in terms of malformed HTML, but also in terms of older versions of HTML.

Many sites are also very long lived. We dont want to break sites just because a newer version of HTML, CSS, JS, &c. is available. So browsers are usually very backwards compatible. They are generally capable of rendering very old sites. Note however that some technologies have lived and died along the way and some browsers, usually for security and stability reasons, have stopped supporting some technologies. Flash is a good example of such a "dead" technology. However the core languages, HTML, CSS, and JS are still, and probably always will be, supported.

As a result, you will see code from lots of versions of the Web and should be in a position to handle it (HTML in general) but we should aim to develop using the latest tools (e.g. HTML5) - particularly because HTML 5 (semantic HTML) added organisational elements that are distinct from previous versions. This is why we are covering bother HTML in general in this unit, but also HTML5. We want to be able to cope, as professionals, with older versions, but also to create modern new sites.

3.2 HTML

Having considered the relationship between HTML and the Web we can now perhaps concentrate on HTML itself. We've already said that HTML is the HyperText Markup

Language. That is, a language for turning text into hypertext by using markup. Note though, that in the grand scheme of things, this is just one way of creating hypertext and not the only way to create hypertext. It is perhaps the most dominant although you might have also experienced alternatives like markdown. In older word processors you used to be able to turn on the display of markup because word processor documents are similarly marked up, albeit internally, to indicate the different parts of the document. In some ways a word processor is very similar, in some contexts, to a Web browser. The underlying format for Microsoft Word documents is an XML document. XML is the eXtensible Markup Language so you'll find markup in lots of places once you know what to look for.

HTML is the standard markup language for creating web pages. It is not a programming language though. This is because it doesn't have any support for programming constructs like expressions, looping, or iteration, i.e. you can't write general purpose programs in HTML. What you can do though is define a Web page.

HTML is just one part of the triad of foundational web technologies (alongside CSS and Javascript). We use it to describe the semantic structure of the data, which CSS in turn presents, usually visually, to the user, and which Javascript can manipulate.

Generally HTML arrives in the browser via one of two routes, either from local storage or from a server. From a server is the most usual. Opening a document from local storage is what happens when you open an HTML file on your local computer, for example, by double clicking an HTML file. A file opened from local storage will run with reduced capability than one requested from a Web server. This is due to security policies associated with web browsers. You don't want your browser to have unrestricted access to your local file system when running arbitrary code, such as HTML which might have embedded or linked JS. The other route, requesting a page from a web server is more usual and is likely the most dominant method for getting a page to display in a browser. This happens after a successful request-response cycle between the browser and the web server.

Once the browser receives an HTML document from either server or storage it then renders that document visually. Note that other user agents can retrieve HTML may use the returned HTML in other ways, for example, rendering the content as spoken text or via a braille display for visually impaired users.

3.3 What dos AN HTML Document look like?

We've talked a bit about these HTML documents and how to retrieve them, but what do they actually look like?

Well, something like this (depending upon the content that is marked up, which is basically unlimited):

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>My first HTML 5 document</title>
```

```
5     </head>
6     <body>
7       <p>Hello World from HTML 5</p>
8     </body>
9   </html>
```

Remember that HTML is plain text. This means that you can write HTML in any text editor and only need to save it as a .html document (which can then be opened in a browser). Most operating systems will take care of the details of opening the file correctly. Depending upon your setup though you might need to specifically tell your machine to open an HTML document either in your browser or your editor, depending upon which you do most often. I find myself that HTML files on my local storage are usually there for editing so I have my machine set up to open files ending in .html (and .css and .js for that matter) in my favourite text editor rather than in my default web browser. Note that this doesn't affect your browsers ability to open and render HTML from web servers, or any file that is explicitly opened in the browser, i.e. via the File > Open File menu item or a dragged and dropped file.

3.4 Classical HTML to Modern HTML

We are currently on version 5 of HTML, known as HTML5, but there have been multiple major and sub-versions of HTML over the last thirty years or so. A lot of web sites are still written, and are still being written, using earlier versions of HTML. So it is worth your while to be aware of earlier versions and how and why they changed. Also, remember that HTML has developed over the years, usually in response to the desires of users and designers who are generally trying to make the Web better.

Early versions of HTML, at least until version 4.01 were primarily concerned with defining the visual presentation of a web page. This mixed both structure and presentation. That is the structure of a page, such as whether a section of text is a heading or a paragraph, but also how that section of text should look to the user, e.g. the font face, font size, font colour, etc.

If we think about how an HTML page can be used in different ways, either viewed in a browser, or processed for use in other tools, or in a mobile app, or an accessibility device like a Braille reader, or rendered differently for printing to paper or saving to a PDF, then it seems silly to dictate in the HTML how the page should look, because that depends on how it is being used, and that decision, no matter how much a designer might want to control things, is a decision for the end user to make. As a result, over time the HTML language has been revised to remove the aspects that deal with presentation and concentrate solely on structure and meaning. Presentational aspects have, instead, become the responsibility of a language designed for it, namely CSS.

So modern HTML describes the content, its structure, and its relation to other content and visual presentation of those things is delegated to CSS.

3.5 HTML ELEMENTS

HTML documents are constructed from HTML Elements. Let's consider our example HTML document from earlier:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML 5 document</title>
  </head>
  <body>
    <p>Hello World from HTML 5</p>
  </body>
</html>
```

The whole thing is an HTML document, but it is made from individual HTML elements. Elements are keywords that are encapsulated within angle brackets, e.g. <html>.

Notice also that many of the elements come in pairs, in fact all of the elements in our example above. Elements are represented using opening and closing tags, e.g. <html></html> such that <html> is an opening tag and </html> is a closing tag. Most tags delineate the start and end of a portion of text or enclose other sets of tags so are often paired, one for each end of the section.

Some stand alone amongst the text, e.g.
 which is used to insert a line break within a document. Standalone tags aren't part of an opening and closing pair, so aren't capable of enclosing a section of text or any other element like pairs do.

Tags are combined to create structured documents by denoting structural semantics for the text (such as headings, paragraphs, lists, links, etc.). Notice how, in our example, everything is enclosed within the <html> pair. The next layer of tags is the <head> and <body> pairs. The <head> pair encapsulates the <title> pair, and the <body> pair encapsulates the <p> pair. This encapsulation results in a hierarchical structure, formally a tree structure. We can describe an HTML document as having a tree structure where the <html> tags are the root of the tree, and the, in our example, <p> and <title> tags are the leaves of the structure.

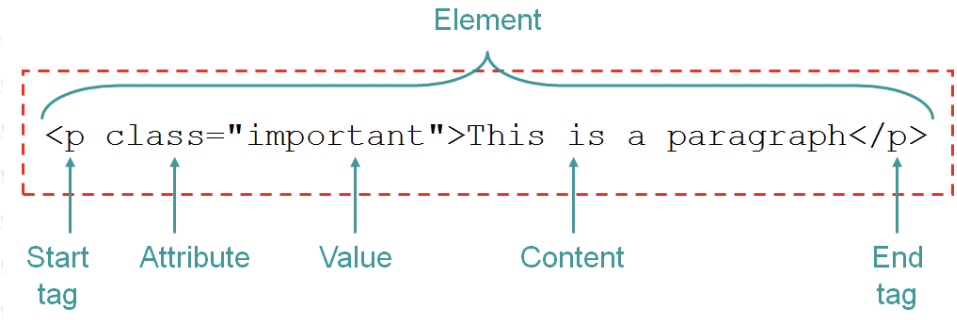
3.6 HTML Element Structure

HTML Elements are not just simple words with angle brackets around them like we saw in our example, e.g.

```
<p>Hello World from HTML 5</p>
```

Elements can do so much more than that, for example, they can have attributes and values which can give the browser additional information about how to deal with the tag. For example, we can assign individual tags a class so that they can be grouped together. This is useful when we use CSS later as we can then say something like treat all tags in class x in this way, e.g. embolden all tags which are assigned to the important class,

e.g. class="important". We can also give individual tags an ID so that they are uniquely identifiable and distinguishable from all other tags in the page. This is again useful when applying CSS and allows us to say, render this specific element, identified by ID, uniquely from everything else. Attributes and values are also used to attach javascript to individual elements so they are really important and give us one way to link between the various core web technologies and enable them to interact.



We'll see later that there are other methods we can exploit, but this is the core way at least from the perspective of HTML.

3.7 HTML Versions

We mentioned earlier that there are various versions of HTML that you will come across. As you investigate various web pages you will notice considerable variation amongst versions of HTML . Having now considered the things that make up an HTML web page such as the tags, elements, and attributes, we can now inspect a simple comparison between HTML4.01 and HTML5.

HTML4.01 looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01// E N
  "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
  <HEAD>
          <TITLE>My first HTML 4.01 document</TITLE>
    </HEAD>
    <BODY>
          <P>Hello World from HTML 4.01</P>
    </BODY>
</HTML>
```

A very similar document using HTML5:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first HTML 5 document</title>
  </head>
  <body>
    <p>Hello World from HTML 5</p>
  </body>
</html>
```

A few things to note, HTML5 uses lowercase for tags which makes things generally easier to read. The DOCTYPE declaration is also much simpler and easier to remember than in HTML4.01. Otherwise things are fairly similar. The larger differences are in terms of the exact range of available tags available for each version. HTML5 deprecates certain tags that relate to styling, strengthens the semantic interpretation of other tags, and introduces a set of useful organisation tags that are purely concerned with the semantic relationship between tags and how those tags are grouped together. Over the new few sections we'll look at the rich range of HTML4.01 tags and then concentrate, towards the end of the unit, on the semantic aspects of HTML5.

In summary, we should write HTML to the current version, e.g. HTML 5, but we should be aware of what earlier versions looked like so that we can handle them when we encounter them in the wild.

3.8 Validity

Because HTML is a language we can talk about whether something written in that language is correct or not. If it is correct then it is valid and if it is incorrect then it is invalid. Think of this a bit like writing a sentence in English, or any other language, but we'll stick with English for now. We can write things in English that are not grammatically correct and this can lead people to not understand what we are trying to communicate to them. It is even more complex with most programming languages because people can be quite flexible in understanding each other but computers must be very precise in their use of language. Browsers are quite robust in their treatment of malformed HTML, but despite this, it is still better if our HTML documents are valid.

There are tools to automatically verify that a given HTML document is correct (or otherwise) One example is the W3C validator tool[1]:

[1] `https://validator.w3.org/`

It is well worth testing pages that you write using this service, especially when you are first starting out with HTML.

3.9 HTML Tags

We've mentioned HTML tags but haven't really examined the sheer range of tags that HTML has available. There are a large number of tags in HTML. This unit will survey many of them, but it is worth using the following resources to explore them in detail:

- MDN HTML Reference[2]
- W3Schools HTML Examples[3]

Let's try to take a structured approach to exploring the range of tags. Starting at the highest level we have an HTML document that contains a definition tag and structural tags. The structural tags can then contain other tags, frequently in combination and encapsulated within each other. Let's now survey the range of standard HTML tags.

Document Type Definition: <doctype>

Document Structure: <html><head><body>

[2] `https://developer.mozilla.org/en-US/docs/Web/HTML/Element`
[3] `https://www.w3schools.com/tags/`

Within the <head> section: <title>, <base>, <meta>, <style>, <link>

Tags for text blocks: <address>, <blockquote>, , <h1>...<h6>, <p>, <pre>, <xmp>

Tags that define lists: <dir>, <dl>, <dt>, <dd>, <menu>, <ol>, <ul>,

Tags that define text format: <b>, <basefont>, <big>, <cite>, <code>, , <font>, <i>, <kbd>, <strike>, <sup>, <tt>, <u>, <var>

Tags that define anchors and links: <a>

Tags that define images and image maps: <img>, <area>, <map>

Tags that define tables: <table>, <caption>, <thead>, <tbody>, <tfoot>, <tr>, <th>, <td>

Tags that define forms: <form>, <fieldset>, <input>, <select>, <option>, <textarea>, <label>, <legend>, <isindex>

Tags that define frames: <frame>, <frameset>, <iframe>

Tags that define scripts: <script>, <noscript>

Tags that define applets & plug-ins: <applet>, <param>, <object> (<embed> not standard)

Tags that adjust text: <br¿, <center¿, <hr¿

Note that HTML5 incorporates additional tags that we'll look at later. Over the next few sections we'll survey some of these in more detail.

3.10 Text Formatting Tags

These tags should be reasonably straightforward and understandable as they usually have equivalent concepts in word processing which most of us should be familiar with. We have:

Headings: <h1>, ..., <h6> - various level from the largest <h1> through to the smallest <h6>

Physical Styles: <b>, <i> - for bold and italic. As these are presentational they are slowly being phased out in preference of for "emphasis"

Logical Styles: <cite>, <code>, , - used to delineate blocks of text that should be treated differently.

Note that most browsers will also still let you do things like the following:

```
1 <font face=‘‘’’ size=‘‘’’>
```

You might see this in the wild, especially in older pages. You can do this but shouldn't. EVER. This is mixing presentation and structure. You should always use CSS for presentational aspects of typography if you want your page to have a given look. You can use styles like to hint to CSS that you want something to be presented differently, but the actual presentation, how the user experiences the page, is dependent upon the users context so shouldn't spelt out explicitly in the HTML. Support for all presentational aspects within HTML is slowly being phased out as CSS is both a better and more powerful way to do things, but also allows a much better "separation of concerns" between structure and presentation.

3.11 Lists

HTML gives us three methods for making lists of things, probably because lists are one of the most prevalent ways to organise information.

Ordered Lists are itemised and numbered from 1 onwards. Use this if the members of your lists have a particular order or if you want each element to have its own number.

```
1 <ol>, <li>
```

Unordered lists are essentially a list made of bullet pointed items. Use this if you don't care about the order of the members of the list.

```
1 <ul>, <li>
```

Definition Lists are like dictionary entries where there is an identifier and then something elaborating on that identifier, for example, a word and its definition in a dictionary. Use this if you don't want either a number or a bullet but want to use a piece of text to define each entry in the list.

```
1 <dl>, <dt>, <dd>
```

3.12 Links

I've belaboured the hypertext aspect of HTML so far, because it is a critically important aspect of the Web. Without hypertext there wouldn't be a web, just a collection of individual pages. Hypertext is what turns a page from a single, isolated item of information into an element of the Web. By following links we explore and discover. So from the hypertext perspective links are the most important element of HTML. Hyperlinks in HTML turn text into hypertext using two types of link. These are internal links and external links.

Internal Links are links within the same page that you are viewing. These can be navigational, to help you efficiently explore longer, or more complex pages, for example, by linking from a contents section to the specific sub-section, or purely for user experience .e.g a link to the top or the bottom of the page. To use an internal link we need to define a target or destination that the link points to. We call this an anchor, e.g.

```
<a name= name >...</a>
```

We then point to the anchor using an ¡a¿ tag where the value of the href attribute contains the name that you used in the anchor, e.g.

```
<a href= #name > ... </a>
```

External links are very similar except that they don't need an anchor link as they point to another page. So you just need the web address of that remote page. Note that we also count separate pages within the same site as "remote pages". There are three patterns of remote link that you should be aware of:

We can link to another document in the same site using, e.g.

```
<a href= page.html > </a>
```

We can also link to an anchor within another document (if that document already contains an anchor), e.g.

```
<a href= page.html#name > ... </a>
```

Finally we can link to another site entirely by using a full Web address, e.g.

```
<a href= http://www.simonwells.org > ... </a>
```

3.13 Tables

After lists of data, tables are probably the next most prevalent organisational element. A table is merely a rectilinear layout of data into rows and columns. The top row of a table is, optionally, a header row, and each row after that is data to fill out the table. HTML tables are implemented along the same lines. A table is defined using the <table> tag. A series of rows are then defined using the <tr> tags. Within a row, a set of column headings can be defined using the <th> tags or else a set of columns, making up the contents of the row, using the table data, <td>, tags.

For example:

```
<table>
  <tr>
    <th>Heading 1</th>
    <th>Heading 2</th>
  </tr>
  <tr>
    <td>data 1</td>
    <td>data 2</td>
  </tr>
</table>
```

There are also additional table related tags that you can use, e.g. <thead>, <tbody>, <tfoot>, <caption>, which enable you to define more semantic structure for the table. As for all of the other tags, refer to the MDN HTML Reference for more details.

Note that whilst it is tempting to use tables for presentation and layout they shouldn't be used for this. There are better CSS features for doing visual layout. Instead, use tables only for data representation in circumstances where you either have tabular data or where the data is usefully grouped into rows and columns for better comprehension.

3.14 Images

Lots of text with no images can be a little boring. If we recall the original idea for the Web, as a way to share, distribute, and publish science data and experimental reports, then it seems natural to want to be able to incorporate images for figures and illustrations. Even if we extend this to newspaper and magazine articles, the use of images is even more common. So it seems natural to have a way to incorporate images, and that's what the ¡img¿ tag is for. This tag has mandatory attributes, "src" which we use to indicate the image file that should be inserted into that location of the HTML document once it is rendered. For accessibility, alternative descriptive text is required to be provided using the "alt" attribute. This way, for example, text based browsers or braille displays can describe what the image would have contained. Without "alt" text there is a gap in the content of the page which can hugely affect usability and user experience.

Image tags also have a range of optional attributes, e.g. width, height, longdesc , which can be used to give the dimensions of the image so that when rendering the page, if the image hasn't yet downloaded, because it might be very large, space can be left for it and the rest of the page displayed. The 'longdesc" attribute enables a long alternative description of the contents of the image to be retrieved and used. Note that the status of longdesc into the future is uncertain and may be deprecated from future standards.

The image type that an <img> tag can refer to include: GIF, JPG, PNG but most modern browser support is so good that we dont consider the image type so critically anymore. That said though, when putting together your own sites it is worth paying some attention to the specific images types you are using.

3.15 Forms

Everything so far has been about retrieving HTML pages from the Web server. By default this is, implicitly, using the HTTP GET method behind the scenes. What if we instead wanted to send data to the web server instead. We have a variety of methods to do that, especially once we get to using JS, but for the moment, the easiest way to send data to a server is to create a form to collect the data to send, then to use a button to "post" the data to the server. This uses the HTTP POST method. For example:

```
1 <form name="name" action="page.html" method="method">
2     ... various controls ...
3 </form>
```

To create a form we use the ¡form¿ tag and set various attributes telling the form what its name is, where the form should post to when its action is triggered, and what method to execute when the action is triggered, e.g.

```
<form name="registration" action="/my-registration-handling-page" method
    ="post">
    ... various controls ...
</form>
```

Notice that I've only mentioned the things that make up a form in passing and made reference to them as "...various controls ..." in the examples. This is because there are quite a few form controls that we can use to get user input and these are best discussed in the next section.

3.16 Form Controls

Forms can be built from a whole heap of different controls. These include buttons, checkboxes, radio buttons, text boxes, password input text boxes, hidden fields, file upload fields, selection lists, text areas, labels, and various mechanisms for grouping things together. Most of these should be familiar to you in principle from the kinds of data entry mechanisms that we are used to in most desktop and mobile applications. Similarly, if you want to see an example of a form then look at any signup or login page on any random website. An awful lot of the functionality that we are used to using on the modern web involves form elements. Note that these can all be styled using CSS so sometimes you might not immediately recognise the exact underlying element that is being used on any given page.

Buttons: <input type="submit">, <input type="reset">, <input type="button">, <input type=image">

Check boxes: <input type=checkbox>

Radio buttons: <input type=radio>

Text boxes: <input type=text>

Password textboxes: <input type=password>

Hidden fields: <input type=hidden>

File Upload: <input type=file>

Selection Lists : <select>, <option>, <optgroup>

Text Areas : <textarea>

Label (for a control) : <label>

Group of Controls : <fieldset>, <legend>

3.17 HTML, HTML5, & Semantic HTML

So far we've considered HTML in the historical context, in general terms of all of the tags that have accumulated up until now with the only proviso in their usage being a general rule to not use HTML for visual presentation where CSS could be used instead. Up until HTML5, earlier versions concentrated on document markup and there have been a number of false starts over the years, for example, including tags that were purely presentational. Since HTML5 there has been a focus on distinguishing semantic meaning within a page and then allowing these semantic elements to be handled by the user agent as appropriate for the user, with the default, and most common, being to render the page visually.

What do we mean by semantic? Perhaps we should turn to the dictionary definition first:

```
semantic | si'mantik
adjective
relating to meaning in language or logic.

semantics | smantks plural noun [usually treated as singular]

    the branch of linguistics and logic concerned with meaning.
    The two main areas are logical semantics, concerned with matters
    such as sense and reference and presupposition and implication,
    and lexical semantics, concerned with the analysis of word
    meanings and relations between them.

    the meaning of a word, phrase, or text: such quibbling over
    semantics may seem petty stuff.
```

So the message to take away is that modern HTML, from version 5 onwards should be concerned primarily with cleanly and unambiguously communicating the meaning of the content of a document to its users. The meaning can refer to the specific elements, e.g. the semantic status of a given heading, or to relationships between elements or groups of elements. We'll see very soon some new tags from HTML5 that are designed to enable us to describe the semantic status of various different sections of a page so that they can be handled as a collection, for example, a collection of links that might want to be used for navigation should be grouped together perhaps using a pair of ¡nav¿ tags. We'll see this a little later, but for now, let's explore why the move towards semantic markup has happened.

3.18 Misusing tags

HTML tags can often be used in different ways to get different effects. For example, for many years there wasn't a good way to control the layout of elements on screen. Designers realised that they could misuse table tags to at least get some control over placement of elements, e.g. that something was to the left or right of, or above, or below another element. However this presentational use of ¡table¿ was wrong and would break

the semantics of the page's organisation. Why? Because a table is a single entity in which the assumption is that the individual parts that make up that entity are related to each other. In a well formed table, items are related to each other by column and by row, and we assume things about the relationships between elements depend upon the meanings of those rows and columns as defined by the column headers. When a table is used for layout it breaks this relationship of meaning between elements. So, if we misuse tags, we might get the desired effect but we lose clarity in communication. This is important because the consuming software, the browser, then doesn't know how to handle the contents of those tags and cannot make assumptions based upon the proper use of those tags.

Let's consider another example. The and tags are used to mark up a paragraph. This has meaning because people usually know what paragraphs are and browsers know how to handle paragraphs because they are programmed that way. So paragraph tags can be considered to be semantic markup; they convey information about the meaning of their contents which is in addition to those specific contents. We can contrast this to tags like ¡b¿ or ¡i¿ which don't convey anything about the content that has been marked as bold or italic, and just communicate how it should look. We can also do things the other way around, for example, when we see this:

```
1 <h1>This is a top level heading</h1>
```

We know that it is telling us that the enclosed text is a level 1 heading, however it is represented. However this:

```
1 <span style="font-size: 32px; margin: 21px 0;">Is this a top level
    heading?</span>
```

will look very similar to a default <h1>but it is no longer clear that the enclosed text is a level 1 heading or merely some text that is a little bigger and heavier than the rest. So HTML tags are useful in communicating information about the text that they enclose.

A few other common misuses of HTML tags to achieve specific visual representations that might break the semantics of the HTML tags include:

blockquote - used to indent text because default presentation of a blockquote is indented. This is instead of using CSS margins to create the indent which is a purely visual presentational consideration. paragraph - used to add space between page elements instead of defining actual paragraphs. This is instead of using margin and padding style properties.

Unordered List - used to indent text but without list elements the HTML is invalid as well as being semantically incorrect (margin or padding styles) but most browsers will still render it as visually intended.

Heading - used to make text bigger and bolder. Semantically incorrect if the text isnt actually a heading (font-weight and font-size CSS properties)

Note that we could take the misuse of tags to an extreme. We could replace all body content markup tags with ¡div¿ tags and then style them all individually using CSS.

What do we gain or lose? We'd possibly have a page that looked exactly the same (due to the use of CSS) but the HTML would lose its communicative aspect. This can lead to so-called "Div Soup" which was very prevalent in many sites over the last decade. Div soup occurred because the tag is extremely flexible in terms of how its contents are rendered. Until HTML5 added support for things like headers, footers, and other organisational elements, web developers continuously invented their own novel ways to do the same groupings, usually using divs. The result is illustrated in Figure **??** which shows the ambiguous div soup versus structure semantic markup.

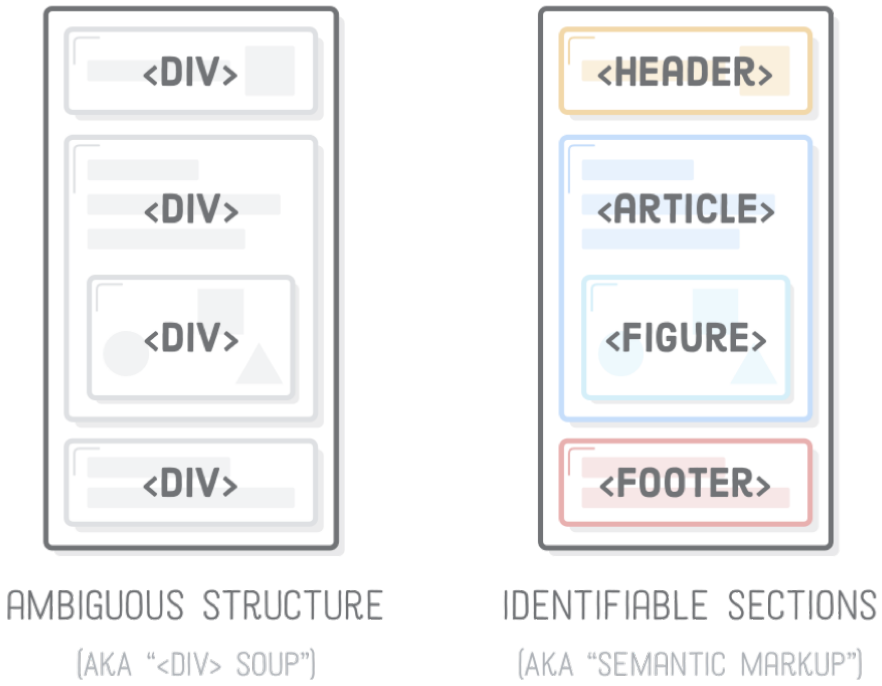


Figure 3.1: An illustration of the ambiguities that can arise when using Div elements to impose structure instead of HTML5 semantic elements.

3.19 Structure & Meaning

Let's take a slight detour and consider the relationship between the structure of a web page and the meaning of its content. If we consider a search engine, indexing all of the pages that it can find, so that it can show other people to it as necessary. A search engines doesnt see the style and presentation of your page. It only sees the content. For efficiency, many search engines will ignore the CSS because the CSS isn't meant to convey meaning, only visual style and presentation, so can be safely ignored. Instead the HTML is interrogated, because the HTML is meant to convey meaning. The search engine will only care about the HTML because it is designed to attempt to "understand" the content and it is the HTML that holds the content.

Where we place content on a page, and how we mark it up, can alter how that content is dealt with. This is not from a visual presentation perspective, but from the internal structure and semantic meaning perspective. For example, content in an <h1> tag is

more likely to be weighted as important to interpreting the overall structure of a page, and as a result contributes to the search engine analysis of a page in a different way to other content marked up using the <p> tag.

3.20 Semantic Markup

Semantic markup stemmed from the drive towards marking up meaning in addition to the typographic elements that make up any given document. This happened in response to designers attempting to bring semantic markup to HTML. A good example is the use of and tags to group elements of a page together as we saw earlier when considering div soup. The main problem is that these and themselves dont actually mean anything. They don't have any inherent meaning. If you are presented with a then it is difficult to infer exactly how the content of the ¡div¿ should be treated. For example, one could be a navigational collection, another could contain core content. How does a machine separate them so that it can treat each appropriately?

By being explicit and including tags in HTML5 to support these goals we can use appropriate tags to delineate between, e.g. core content and navigation. The browser can then treat each differently as appropriate. This is why HTML5 includes tags like the <nav> tag, to communicate that its content should be treated as navigational content and not as body content or any other kind of content.

Over the years, designers defined their own IDs and class names but this was ad hoc; people took their own approaches and so they were naturally inconsistent. This made pages difficult to process. For example, search engines detecting useful parts of a given page, or braille readers being able to ignore navigational elements when reading page content. In response HTML was extended and clarified to include tags that communicate more clearly defined meaning about their content. This is, in time, leading towards better structure for pages, as well as increased accessibility, and more reliable automated processing, maintainability, and reuse of web resources.

3.21 HTML5

Semantic meaning has always been present in HTML but is now, since HTML5, emphasised. HTML5 attempts to address this through addition of meaningful grouping tags so that the various parts of a page can be delineated and communicated more easily and reliably. These could be considered as "sectioning elements" or even as divs but with added meaning.

HTML5 includes support for the following semantic markup:

- <section> - a thematic grouping of content, typically with a heading
- <article> - independent, self-contained content (suitable for syndication or reuse)
- <header> - container for introductory content
- <footer> - information about its containing element (author, copyright, &c.)

- <nav> - major blocks of navigational links
- <aside> - related additional content
- <figure> & <figcaption> - visually explain an image
- <main> - the core content of the document
- <mark> - highlighted/emphasised sections
- <details> - additional information that can be hidden/shown
- <summary> - visible heading associated with <details>
- <time> - date/time information

Most of these tags should have names that enable you to reasonably easily infer their intent. This is what makes them semantic tags. These new tags can be used to communicate the various parts of a page as shown in the following figure.

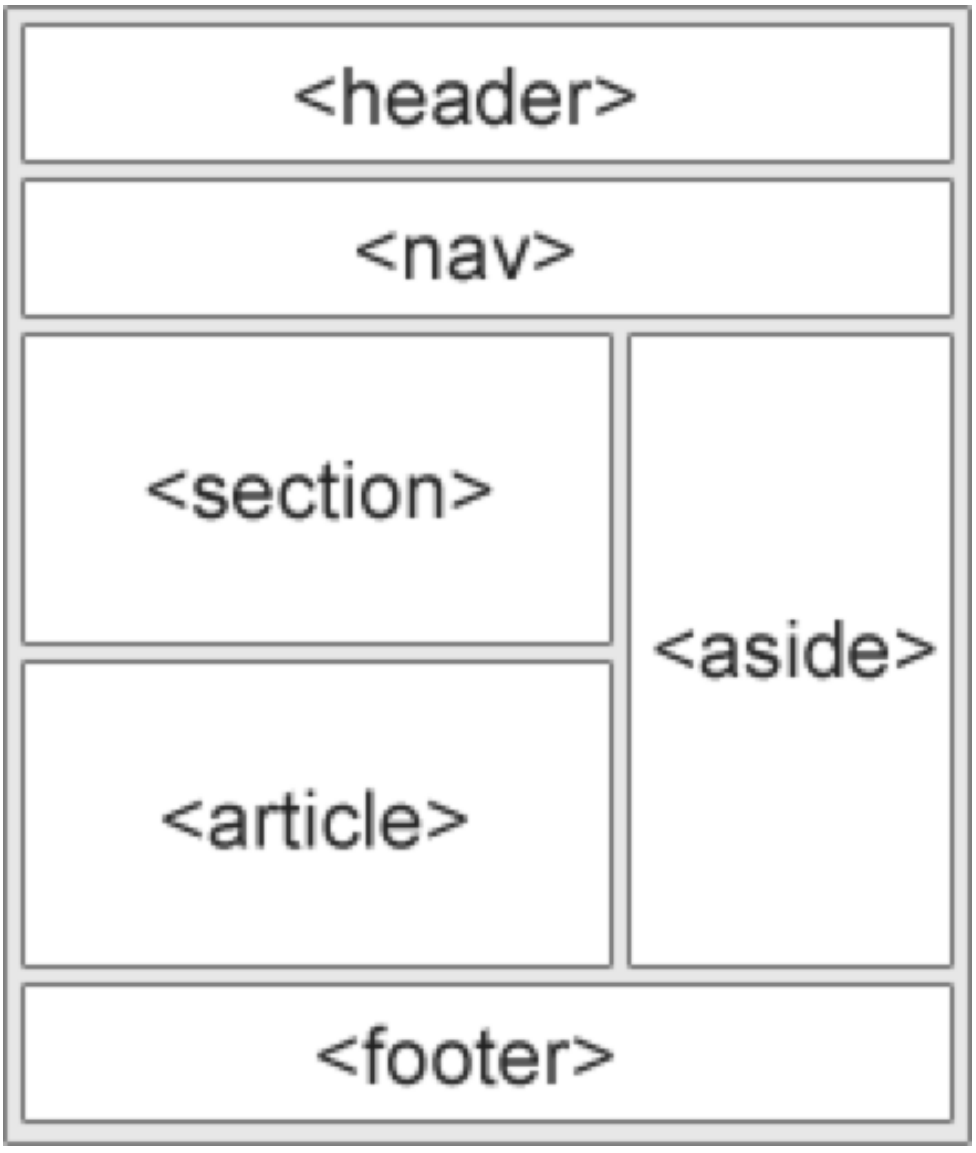


Figure 3.2

3.22 Semantic Meaning in Inherited Tags

The move to HTML5 involved some inherited pre-existing tags being redefined to have semantic meaning where previously they did not. This is summarised in the following table. You should notice the absence of the ¡b¿ and ¡i¿ tags - these have presentational meaning only so are not considered to be semantic HTML.

Tag	Meaning	Tag	Meaning
<abbr>	Abbreviation	<h1>	First-level headline
<acronym>	Acronym	<h2>	Second-level headline
<blockquote>	Long quotation	<h3>	Third-level headline
<dfn>	Definition	<h4>	Fourth-level headline
<address>	Address for author(s) of the document	<h5>	Fifth-level headline
<cite>	Citation	<h6>	Sixth-level headline
<code>	Code reference	<hr>	Thematic break
<tt>	Teletype text	<kbd>	Text to be entered by the user
	Logical division	<pre>	Pre-formatted text
	Generic inline style container	<q>	Short inline quotation
<del>	Deleted text	<samp>	Sample output
<ins>	Inserted text	<sub>	Subscript
	Emphasis	<sup>	Superscript
	Strong emphasis	<var>	Variable or user defined text

Figure 3.3

3.23 Summary

We've covered a lot of ground again in this unit. You should now:

- understand how HTML has developed and why it works the way it does
- be aware of the range of tags supported by HTML
- be able to assemble basic HTML documents
- understand the difference between syntax and semantics
- be aware of the need for semantic representation within markup
- know about the range of semantic tags within HTML5

There is obviously much more to effective HTML use than we can cover in one unit, but we can develop effective skills through practise. This isnt the end of the story about HTML, semantics, and the representation and use of meaning on the Web, well see HTML much more as we style and interact (programmatically) with it.

For a diversion you could consider investigating the “Semantic Web”. This is the “web of meaning” which is finding new application alongside AI technologies if you want to get an idea about one possible future of the Web.

Chapter 4

CSS Intro & Overview

In this chapter we begin to look at a second Web language, Cascading Style Sheets (CSS). This language is frequently considered to be a design or presentation language, and we'll start to look at how we can use CSS to style the elements of our HTML documents to make them look more visually appealing. HTML alone is really great, but pretty ugly. CSS gives us a way to improve how the elements of our pages are presented.

4.1 HTML & CSS

As we've seen in the last unit, HTML gives us a language for describing structure. This is essential, without structure, our text loses its hypertext abilities, and the advantages gained from markup. However, we've also seen that, over time, HTML has had its abilities to define style, the presentation of the content, curtailed and in some cases removed. At the very least, using HTML's residual style capabilities to determine how a page should look is considered, at best, old fashioned and to be frowned upon.

Instead, we now use a style language to determine how our content should appear. It is Cascading Style Sheets (CSS) that enable us to control presentation. This is not just about making things look pretty however, but is very important to usability and accessibility. Partly this is for the developers, instead of having content and presentation mixed they can be handled separately, perhaps by different members of the team. It also enables more efficient representation, for example, a style can be defined once and then applied wherever it is needed.

In addition, multiple different presentations of a single set of content can be defined and then used depending upon the context. This is how things like a print layout and a screen layout can be defined differently, or perhaps different screen layouts for different sizes of screen.

In this unit we will concentrate on presentation from a technical perspective and specifically on how to style the individual elements of pages. Well then look at aspects of design over the next couple of units to try to apply this to produce cohesive designs. Briefly, styling individual elements (this unit), arranging groups of elements (next unit), and bringing everything together to create cohesive designs (next unit but one).

4.2 HTML Content & Presentation

Early versions of HTML mixed content and presentation. This was fine, it worked, but as the Web grew in capability and utility, the ways that the Web was used changed and this put pressures on design decisions made early on that governed the production of early websites. Initially every element needed font, colour, style, alignment, border, size, etc. to be explicitly described, often repeatedly so, throughout the HTML document. Anything that wasn't specified would revert to the default presentation.

Why does it default and where does the default style come from? To answer this it's worth noting that browsers have a built in style sheet that applies a default style to pages when no other style information is supplied. All style information, whether from HTML, or CSS, will override this internal browser stylesheet but in certain circumstances an end user will want to apply their own stylesheets to your site, regardless of your design. This is because end users might have their own accessibility requirements which must always take precedence over the web designers desires. The consequence is that you cannot rely on your carefully implemented design being what your user eventually sees. It is still worth caring that your default site is good and usable, but you should also ensure that the content is complete and well structured and not dependent upon the design for it to be usable.

Over time, HTML has had its capabilities to describe style reduced. Moving style declarations, initially into their own separate blocks within the HTML document, and then later to external files made things both more simple, more efficient, and more flexible. The style declarations that were initially applied to typographical elements were eventually developed into a complete language for describing how HTML should be presented visually. These style declarations can now, with some caveats, be applied fairly uniformly across most HTML elements, not just applied to typographical elements. This allows not only a section of text to be styled, but also the block that it is grouped in, and every other element that makes up a given page.

Overall, removing style declarations gives simpler HTML which can focus on content representation, structure, and semantics, and makes for more manageable designs. This is important once our sites begin to grow and become more complex.

4.3 Content & Presentation: A Separation of Concerns

Before getting into the technical aspects of CSS, let's consider a few important design related topics first, namely, the relationship between content and presentation. When we have HTML and CSS, and when applied properly, the visual and design aspects of a particular view, rendering, or usage of a document are separated from the core content and structure of that document.

It's worth considering some other perspectives on the separation of content and presentation. Think of the human body. We have a skeleton which gives us structure but it is our flesh that gives us our appearance. Given a body without flesh, i.e. a skeleton, it is quite

difficult for most people to distinguish this from any other skeleton, unless perhaps you are an anthropologist or physiologist. Similarly flesh without a skeleton would look and act very different. The two need to work together to give the individually distinguishable and capable humans around us, different complex systems working in concert. The same relationship holds between content and presentation, and between HTML (content) and CSS (presentation). Either, without the other, is less good than both working together.

The nature of many modern, complex, computational systems means that a team of people work together on them. There are many tools for supporting co-operative work, but the separation of concerns, in this case the separation of content and presentation is of particular importance. By separating responsibilities it enables people to work in parallel, each working on their area separately to make progress. This doesn't mean working in isolation, just that the better separated the concerns, the more work can be done on each before progress on one is hindered by the other.

This is not a rigid rule though, but more of a best practice. Different members of a team can work on each aspect and content can be presented in different ways.

This approach also has benefits after a site is deployed. It enables, for example, people with visual impairments to provide their own stylesheets to the browser to override the website developers decisions. It enables screen and print versions of pages to be styled differently. Content can be tailored to the medium of consumption.

As I mentioned earlier, this is not a rigid rule, inline and embedded styles will work in most browsers, and you might not be able to tell the difference, and this is often useful for quick hacks when developing and trying to get things right. However this can lead to future problems. Separation of concerns leads to simplified change management, increased flexibility, and better usability.

4.4 Design First or Content First?

This separation of concerns does raise a question though, should we design how our page will look first or create and structure the content first?

Often you'll have a general idea for the overall look of a site before you start building it. At least you will usually have some idea of the feel that you are aiming for. Frequently this will be out of your control though, especially when building sites for others. Generally though you cant really design something well until you have some idea of what the design is working with. You need some content or at least an idea of the structure of the content to get started with a design. Without content, the design might include things that won't appear in the site. At best, this is wasted effort, at worst, it can guide the design process in a direction that is wrong for the site you are building.

We should recognise that content often drives the design process. Even without a design, we can take out content, structure and mark it up using HTML, and view this in a browser. We can have a navigable site without any design. It will certainly look quite poor though and will almost definitely not be anything like you would want to build, but

it is a starting place. From this we can conclude that, if in doubt, we can start building the HTML, ensuring that it is well structured and captures all of the available content, before we start designing.

Note that we can also start by mocking up the content. We'll see some techniques for doing this in the design for hackers unit. This is a great way to give you something that you can then start applying designs to. However, it is very likely that mocked up content will diverge widely from the final content which gives an opportunity for the design process to again head in the wrong direction. The need for mocked up content can occur in real world projects for many reasons though, often related to timing and imperfect alignment between business units. If the content writers are not yet ready to provide content, but the design and implementation teams are ready to start then it can be better to have them producing potential solutions that can later be verified using real content.

We can relate this idea of a content first design process to other design ideas. For example, it is often said that "Form follows function". How something is structured stems primarily from the underlying engineering requirements and a desirable, often elegant, design is often the one that supports the needed functionality and nothing more where every extraneous element that is merely there for decoration has been pared away. Another way to think about content and design is to consider that we should not conform structure of content for design reasons if it then leads to a compromised system. If the design, for purely design reasons, then leads to a buggy, inefficient, unreliable, or unusable site, then something has gone wrong in the process. That said, we can, and indeed should, let structure inform the design process. Many elegant solutions stem from the design exposing underlying structure.

If in doubt though, the default should be to start with content, write HTML, and think about your data before getting to design and CSS.

4.5 An Overview of Cascading Style Sheets (CSS)

Having considered some important design considerations lets now turn our attention more fully to the idea of CSS as a technology, before looking at the technical link between HTML and CSS. We'll then finish by looking in detail at CSS as a language.

CSS is a simple, text-based, page appearance description language. In more formal terms we could also call CSS a declarative, domain-specific, programming language. These two descriptions give us lots of ways to consider what CSS is. We'll perhaps skip over the simple part for now. CSS documents can become quite complex if you write them that way, and you'll see lots of complex CSS in the real world, but there is no need for your CSS code to be complex. It can be simple and straightforward, but like most code, this property is mostly on the shoulders of the programmer. Text based indicates that, like HTML, it can be written using a standard text editor. No special tools are required. This contributes to the democratic nature of web technologies in general, most people can create the basic files that make up a web page without undue expense or effort. The "page appearance description language" phrase is useful though. It kind of sums up much of what CSS is about. It is a programming language for describing how a page should look.

For our purposes the page is written using HTML so CSS is describing how an HTML page should appear when rendered.

In the more formal description we had the term "declarative". This term implies that control flow is implicit. We are saying what we want and the computer then tries to achieve that. We don't say exactly how the end result should be achieved we just declare what we want. This leads us to thinking of CSS as a constraint language. The browser, when rendering, can do what it likes to our HTML page. It generally doesn't, but it can. By using CSS we are declaring what we want the browser to try to achieve by imposing constraints on the elements that make up the page. For example, a declaration that a section of text should be red is also imposing a constraint on the range of valid colours for that section of text. This is also where some of the potential complexity of CSS arises. Some constraints can be incompatible, if we declare that we want to make a section of text bigger, then the container in which the text is held must grow to still be able to contain it. If we also constrain the size of the container then there will be conflict and the effect as a result can be different, for example, either one of the declarations is overridden, the container can grow to enclose the text, or else the text can spill out of the container or be clipped at the edge of the container. This is where some of the perceived complexity of CSS can stem from.

However HTML and CSS are resilient. If a browser cant do what you declare in the CSS then the site will generally continue to function. It is rare for the content to fail to be displayed at all. But it might be unlike the designer intended.

CSS permits almost every HTML tag to be arbitrarily scaled, positioned, and decorated. This overcomes some limitations of styling associated with traditional HTML because HTML limits the application of styles to particular, usually typographical, elements. However, we shouldn't think of CSS as a pixel-perfect user interface design tool like we might use for building desktop or mobile apps. It is not. It is better considered as a tool that enables flexible and robust user interfaces to be implemented that will reflow to work well with screens of different sizes across many platforms.

4.6 The Slow Detachment of Style and Presentation from Content

Let's now quickly survey how the detachment of presentation from content has occurred in HTML. We've moved from the pre-CSS HTML presentational attributes, for example,

```
1 <h1><font color=  r e d  >The Quick Brown Fox</font></h1>
```

to the use of style parameters such as,

```
1 <h1 style="color: red;  >The Quick Brown Fox</h1>
```

Because style parameters only apply to the elements to which they are attached, meaning that declarations with similar parameters might need to be repeated in many places in the same file, a more efficient solution was required. We moved to style blocks, collecting style declarations in their own area of the HTML document, separated from the HTML elements to which they are applied. For example,

```
<!DOCTYPE html>
<html>
  <head>
    <style type= text/css >
      h1 {color: red;}
    </style>
  </head>
  <body>
    <h1>The Quick Brown Fox</h1>
  </body>
</html>
```

The limitation of style blocks is that a given block only applies to that one page that it appears to. However we likely want our entire site to have a similar and consistent style across all pages. So why not collect styles into an external, separate file, and then include it wherever it needs to be used. This leads to external style sheets. Here is an example of the line that would be included in an HTML file to tell it where to find a CSS file for use:

```
<link href="path/to/file.css" rel="stylesheet" type="text/css">
```

Best practice is to use an external stylesheet.

4.7 Using CSS & HTML Together

In the modern web, CSS can be used with HTML in 3 primary ways which are very similar to what we've just seen. Considering the previous topic, the only approach that is no longer valid is the earliest, using presentational attributes. All three others are valid in the modern web but have various advantages and disadvantages. We can

1. Attach CSS to a specific HTML element using the style parameter.
2. Inline CSS globally by collecting it together using a ¡style¿ block
3. Retrieve CSS from an external URL using directives like this:

```
<link rel="stylesheet" type="text/css" href= theme.css >
```

We'll look at each of these in turn over the next three topics and consider the advantages and disadvantages of each.

4.8 Using the Style Parameter

The style parameter is used "inline" as an attribute of the HTML Element to which it is applied

```
<p style= color:red >
```

An advantage of this is that you can put your style declarations right where youre using them. You don't have to scroll to a different part of the file or look in a separate file, it is right there. However, this can lead to lots of repetition. You have to specify everything exactly where you want it. For example, every paragraph that you want to colour red must have an inline declaration which can seriously bloat your site. This is also an extreme mixture of content and presentation. There is no real separation between the two when taking this approach.

4.9 Using a <style> block

A style block enables us to collect all of the style declarations together, for example, those that we might have attached otherwise to individual HTML elements. This means that you can make a single declaration, e.g. color: red; and then list all of the places that it should be applied to. For example,

```
<!DOCTYPE html>
<html>
  <head>
    <style type= text/css >
      h1 {color: red;}
    </style>
  </head>
  <body>
    <h1>The Quick Brown Fox</h1>
  </body>
</html>
```

Here we have a style block in the head section of the HTML document. There is a single declaration, color:red; and this is applied to a single HTML element, the <h1> tag.

This begins to separate presentation from content. The content is all in the ¡body¿ section of the document and the style is now separated into the style block in the ¡head¿ section.

However, this only styles individual pages and not sites. To style all pages in a site using this approach requires the style block to be repeated in every page. If we then make a change to the design, we must edit all style blocks in all pages to ensure consistency across the site.

4.10 External Stylesheets

External stylesheets give us the ultimate, so far, in separation of content from presentation. Content is now stored in the relevant HTML files and styles are stored in CSS files. Multiple HTML files reference the CSS file and hence the style for the site only needs to be placed in a single place, and only needs to be updated in a single place. In other words, we can reuse the same .css file in multiple .html files. This enables us to style an entire site (multiple HTML files) with all the CSS declarations for the visual presentation of the site in one place. This is reasonably easy to manage and update the design without touching the content. We can also add new content and take advantage of a ready made style. Let's look at how this works in practice. Here are the contents of two files, index.html and style.css which work together to style the content of a simple page:

index.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <link href= style.css" rel="stylesheet" type= text/css" media=
          screen  >
5   </head>
6   <body>
7     <h1>The Quick Brown Fox</h1>
8   </body>
9 </html>
```

and the CSS (style.css):

```
1 body {background-color:black}
2 h1 {color:red}
3 p {color:blue}
```

Notice that the <link> element in the HTML document references the name of the stylesheet. This is how we tell an HTML page which stylesheet to use. Within the stylesheet, notice that it references HTML elements that exist in the HTML file. So these two files aren't completely separate. The CSS must declare styles for elements that exist in the HTML document else there is nothing to style. Similarly, to take advantage of a style, the HTML document must include elements that are declared in the CSS.

It is this that enables us to take advantage of ready made CSS designs. There are lots of ready made style collections available, under a variety of licenses that can be included into your HTML. This can mean that you can avoid writing any CSS if you are happy to adopt an existing design and to perhaps have a site whose look and feel is shared with other sites. We'll consider this in more detail in the design for hackers unit.

We should also note that another advantage of external stylesheets is that a page can specify multiple different stylesheets which can be applied in different contexts. A common one that you might have experienced is when printing a page you might notice that the printed layout can differ from the on-screen layout. Similarly, many sites now have dark and light modes which switch depending upon ambient light conditions, or user preference.

These are all controlled and defined through the provision of stylesheets for each context. The HTML media attribute lets you specify different style sheets for different contexts, e.g. print, projection, aural, braille, tty, tv, etc.

4.11 Basic CSS Syntax

So far we've looked at how CSS is related to HTML, how CSS expressions can be made available to an HTML document through inlining, blocks, or external files. However we've not really looked at the language itself yet. Let's remedy that.

CSS is used to write stylesheets and stylesheets are merely a list of rules that declare how we want particular HTML elements to be styled. The rules are constructed from selectors and declarations.

Selectors are used to find relevant HTML elements to apply the declaration to. The declaration is a list of simple pairs of property and value. Each element of the list is separated by semi-colons and properties are delineated from values using colons. This is illustrated in the following figure:

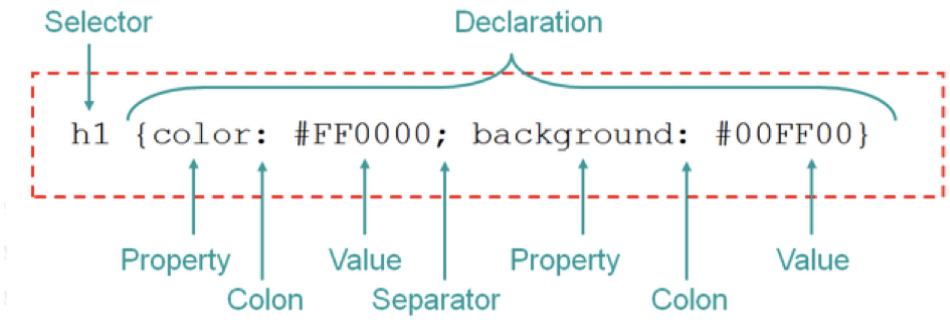


Figure 4.1

CSS rules use a simple syntax and are fairly easy to understand. Essentially one or more selectors identify which html elements to target then a declaration specified how to style those elements.

4.12 Selectors

We use selectors in CSS to tell the browser which parts of the markup to apply a style to. For example we can tell the browser to apply a style to all instances of a given HTML element within a document, e.g. <h1> or <img>.

Sometimes though we will want to treat different instances of the same element in different ways, for example, style the first paragraph in one way and other paragraphs in another way. To support this CSS has a few ways to target different groups of elements or even individual elements. Recall that we can give an HTML element a unique ID attribute to enable us to differentiate that single element from all of the others. In CSS we target elements with a specific ID attribute, e.g. <p id="para-123">, using a selector that matches the ID, e.g. #para-123.

Sometimes though we don't want to target just an individual, and we don't want to target just a given element, what we want to do is to target an arbitrary group of elements. We can do this using the class attribute. If the element has the matching class attribute defined, then it will be matched by the selector, e.g. We can use the .photos selector to match elements in our HTML document that have the same class, e.g. <p class="photos">.

Note that there are also so-called pseudo-classes which refer to special states of selected elements, e.g. hover, visited, active, &c. Pseudo-classes enable elements to be styled in relation to things outside the DOM, e.g. the history of navigation or whether a link has previously been interacted with or followed. We can combine and join selectors in many ways to specify elements by location, element type, id, class (or any combination thereof).

4.13 Declarations

As we saw in the CSS syntax topic, declarations work with selectors. One to tell the browser which elements to target and the other to tell the browser the set of properties to apply to the elements so selected.

Individual declarations are relatively simple. They are designed to give a specific value to a property. This is what is known more widely as a key:value pair. A declaration comprises a property, a colon (:), and a value, e.g. *color: red*.

Properties are defined in the CSS standard and each property has a given set, or range, of legal values. These can include keywords like "center" or values such as numeric values, percentages or others. For example, colour can be specified with keywords, e.g. "red;;, or Hex values like #FF0000 or RGB values like rgb(255, 0, 0).

It is worth getting used to consulting documentation when it comes to CSS properties and values. Some CSS properties can affect any type of element whereas other properties can apply only to particular groups of elements.

Because we are likely to want to set multiple properties efficiently, a declaration is really a semi-colon separated list of multiple individual declarations (key-value pairs). For example, here are two declarations separated by a semi-colon:

```
1 color: red; text-align: center;
```

When using declarations in a CSS file we enclose them into a declaration block, a list of declarations encapsulated between braces, like so:

```
1   h1 {color: red; text-align: center;}
```

The takeaway from this should be that CSS declarations are quite straightforward and don't have to be complex or unwieldy. The trick is for you, as the developer, to stay in control.

4.14 CSS Design Constraints

CSS applies styling and layout to HTML elements, it doesn't create them. We can use CSS to declare that we want to paint a border around an HTML element but we cannot create the basic element itself in pure CSS. All we are doing with CSS is declaring how the page should be displayed but the actual display of the page is the responsibility of the browser and its layout engine. The layout engine will attempt to consistently apply all of the constraints imposed by the CSS onto the elements defined by the HTML. If there are mutual incompatible constraints then these will be resolved by the rules written into the layout engine and this is highly dependent upon the order in which the constraints are imposed with the CSS file.

The idea is that CSS can be used to describe various layouts of content. When the browser layout engine renders this content, it takes account of the platform it is running on and adjusts things as a result. This is how web pages can adjust automatically to any screen/viewport/canvas on any platform. CSS, HTML, and browsers are designed to do this, which is one of the reasons why they are less accomplished when trying built pixel-perfect interfaces. For example, web pages give us text that scales independently of the viewport. Scaling a font, i.e. making text bigger, should lead to a reflow of the page content, shifting other things further down the page to make space, it shouldn't lead to horizontal scrolling. Many browsers also use progressive rendering. In any given page, the layout depends on previous content, i.e. content further up the page, but not future content. However, loading new content might force a reflow of previously rendered content. For example, consider if there is an image element tag part way down a page. If the element doesn't specify the dimensions of the image then space can't be left for the image until it is fully loaded, but this might hold up displaying the rest of the page. Many users would prefer to be able to see the content of a page as it loads, especially the words, rather than waiting for everything to be complete. So the browser will display what it has, then re-render the page if necessary as elements finish loading.

A reminder though that the user can override anything regardless of what the designer thinks or intends. Once your HTML and CSS reach the users browser, the user is in control. There are valid reasons for this as we explored earlier, mainly to do with accessibility and enabling the user to choose how to consume their data.

4.15 Inheritance

Inheritance, the idea that we can build on pre-existing CSS declarations is a key CSS feature. The idea is that we should be able to take advantage of the existence of hierarchical structure in HTML documents to make things more efficient.

Recall that HTML is parsed into the DOM, a model of the objects that make up a page. The DOM is a tree and is therefore hierarchical. It starts with the root <html> tags and all page content is encapsulated within that. When styles are applied to elements higher up the DOM hierarchy they can be inherited by elements further down the hierarchy. In this way nested descendants generally inherit text-related properties from parent elements that enclose them. This is efficient because we then dont have to declare the same properties repeatedly at each level of the DOM hierarchy. For example, given:

```
<h1>
   This is to $<$em>illustrate</em$>$ inheritance
</h1>
```

and the following style declaration:

```
h1 { color: purple; }
```

But no declaration for the colour of the element, the element would inherit the h1 property without us needing to explicitly declare that the word "illustrate" between the tags should also be coloured purple.

Admittedly, this is one of the areas of CSS that causes confusion in practice. It can be difficult to determine where exactly an inherited style comes from even when examining the DOM through the browser developer tools.

4.16 Colours

Colour is an easy place to start when thinking about styles. It has visual impact greater than most other aspects of style. We can control the colour of an element by using the "color" property. We've seen the color property used in several of the examples so far, e.g.

```
h1 { color: purple; }
```

to set the color property of the h1 element to purple. There are however 140 named colours in CSS which we can use directly just by supplying the name as the value to the color property. These include the following: *lightcoral, rosybrown, indianred, red, firebrick, brown, darkred, maroon, mistyrose, salmon, tomato, darksalmon, coral, orangered, lightsalmon, sienna, seashell, chocolate, saddlebrown, sandybrown, peachpuff, peru, linen, bisque, darkorange, burlywood, antiquewhite, tan, navajowhite, blanchedalmond, papayawhip, moccasin, orange, wheat, oldlace, floralwhite, darkgoldenrod, goldenrod, cornsilk, gold, lemonchiffon, khaki, palegoldenrod, darkkhaki, ivory, lightyellow, beige, lightgoldenrodyellow, yellow, olive, olivedrab, yellowgreen, darkolivegreen, greenyellow, chartreuse, lawngreen, darkseagreen, honeydew, palegreen, lightgreen, lime, limegreen, forestgreen, green, darkgreen, seagreen, mediumseagreen, springgreen, mintcream, mediumspringgreen, mediumaquamarine, aquamarine, turquoise, lightseagreen, mediumturquoise, azure, lightcyan, paleturquoise, aqua, cyan, darkcyan, teal, darkslategray, darkturquoise, cadetblue, powderblue, lightblue, deepskyblue, skyblue, lightskyblue, steelblue, aliceblue, dodgerblue, lightslategray, slategray, lightsteelblue, cornflowerblue, royalblue, ghostwhite, lavender, blue, mediumblue, darkblue, midnightblue, navy, slateblue, darkslateblue, mediumslateblue, mediumpurple, blueviolet, indigo, darkorchid, darkviolet, mediumorchid, thistle, plum, violet, fuchsia, magenta, darkmagenta, purple, orchid, mediumvioletred, deeppink, hotpink, lavenderblush, palevioletred, crimson, pink, lightpink, white, snow, whitesmoke, gainsboro, lightgray, silver, darkgray, gray, dimgray, black.*

Colours can be referred to by name, at least for the 140 named colours, but also by hex code, RGB or HSL code. When using hex, RGB, or HSL codes then there are many more than 140 colours available and you have full access to a space of approximately 16M+ colours.

4.17 Background

to set the color property of the h1 element to purple. There are however 140 named colours in CSS whiAfter colour, interacting with the background of the web page is probably the most prevalent way to alter the stylistic character of a page. An easy way is through the background-color property. We can supply it with any of the named colours or colour codes covered in the last topic. Alternatively, we supply an image to be set as the value of the background-image property. This is a picture that is used behind the text and other elements that make up the page content. Background images can be manipulated in various ways, for example being made to repeat, e.g.

```
1 background-repeat \{repeat, repeat-x, repeat-y, no-repeat\}
```

to set the color property of the h1 element to purple. There are however 140 named colours in CSS whior for the image to be located in specific places using the background-position property:

```
1 background-position - specify two values for horizontal and vertical
    from \{top, bottom, left, right, center\}
```

4.18 Font Properties

After the page background, the typographical elements of the page content are the next most prevalent elements that are styled. We can style the following:

- font-family - which font to use
- font-size - how big it should be
- font-style - {normal, italic, oblique}
- font-weight - {normal, bold, bolder, lighter, +numeric values}

4.19 Text Properties

When dealing with typographical aspects of page content, it is not only the fonts that are important to consider, but also the properties of the text itself, regardless of the font and how the letters themselves are displayed. Think about when word processing a document, you will frequently use the text alignment to left or right align your text, as well as decorators, perhaps to underline certain sections of text. CSS enables us to style our page content in similar ways:

- text-align - justify blocks of text, e.g. {left, right, center, justify}
- text-decoration - {none, underline, overline, line-through, blink}
- text-transform - {non, capitalize, uppercase, lowercase}

4.20 Links

I've already belaboured the importance of links in the process of turning text into hypertext. It would be a huge omission if we couldn't also control how we want links to be displayed in our pages. CSS gives us this control and we can affect the color and text properties of links using the following selectors:

- link - the normal state
- visited - if the link has previously been followed
- hover - while pointer is over link
- active - whilst being clicked

These enable us to define how links should look in their various states, e.g.

```
1 a:link {color:red}
```

declares that we want to select the hyperlinks that are in their normal state, i.e. not previously visited or currently actively engaged with by the user. Having selected these links, we want to set their color property, e.g. the colour of their text, to red. Many implemented sites will choose a palette of link colours that work well with the design of the site.

4.21 The Box Model

You might be wondering at this point how the browser places all of these differently shaped elements onto a web page when it is rendering it. The secret is the CSS box model. This places every element of content into a set of boxes that affect spacing and placement in relation to all of the other elements of the page and their CSS declarations in turn. At this point you should be starting to appreciate just how complex the interactions between HTML, DOM, CSS and layout/rendering engine can become.

The box model defines the following:

- Padding - distance between edge of element and its content
- Border - frontier between padding and margin
- Margin - distance between edge of element and adjacent element

which can be used to control the space around an element in a number of contexts. These are illustrated in the following figure:

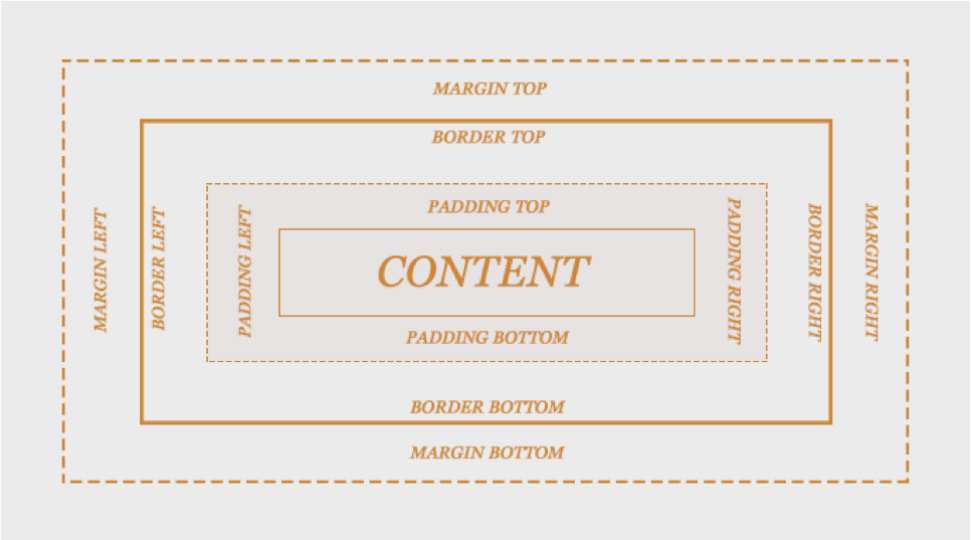


Figure 4.2

Notice that padding, border, and margin are divided into four edges, because they are boxes around the content element, the edges are top, bottom, left, and right. Applied to the border element gives us: border-left, border-right, border-top and border-bottom

The same naming pattern works for the margin and padding boxes. Boxes also have borders which can be visually styled. Often setting them to none is preferred, but borders can be applied to all edges or to individual edges, specifying border characteristics, e.g. solid, dotted, dashed, double as well as width and colour.

4.22 Browser Rendering & CSS

We've mentioned before some aspects of the job that the browser does in rendering the static HTML and CSS files into a visual representation on screen. Much of this occurs automatically according to the rules of the specific rendering engine used by the user's browser. However, some elements of the rendering process can also be manipulated through CSS. How the browser lays out HTML elements follows from the constraints declared in the CSS associated with the page.

However we can use the display property to affect the rendering of specific elements. For example,

- inline - Elements flow horizontally left to right as space allows. Line breaks happen when the edge of the container is reached, and the flow continues on the next line.
- block - Elements flow top down, one after another. The width of the block is what is available in the container, the height is defined by the content.
- table - Child elements are aligned both vertically and horizontally.
- flex - Child elements are placed on a single axis either vertically or horizontally, which specified alignment, adjustments and spacing.

We shall examine the flex and table declarations in the next unit as they are very useful if we want to build particular layouts for our pages.

4.23 Perceptions of CSS & Getting the most from it

CSS is often perceived as being hard or difficult, but this is often due to thinking of CSS as something that its not. It is not, for example, a pixel perfect layout tool like we might see in the graphic design industry for laying out printed pages. It is wrong to think this way. If you instead think of it as a flexible way to produce pages that work in lots of contexts, arranging page content in near optimal ways to make it easy to consume from the widest range of devices and screens, then it starts to make more sense. We can't easily specify an exact layout for every possible screen, but we can declare our desires and then allow the browser to make the best job of arranging things on screen according to those declarations. The complexity in CSS stems from trying to exercise too much control over how the browser renders the page, requiring increasingly complex rules to govern all of the different exceptional states that might occur. As these start to add up we start to realise where the reputation for difficulty comes from. CSS rules can combine and be inherited in a page, and the more rules, the more likelihood for unintended interactions between rules.

So perhaps a change in mindset is required when designing for the Web. We should want to develop sites that are of the web, not merely put on the web. Properties interact and can lead to unexpected results - set one and it combines with others (and defaults) so we perhaps should never be more explicit than necessary to get the effect we want. If we refrain from going too far down the page layout control route and trust our browser rendering engine to do its job, then HTML pages are responsive by default. So my advice is to work with this rather than against it and refrain from trying to force the rendering engine into different or overridden behaviour. Let your content determine either width or height (or both) instead of forcing dimensions and layouts.

Ultimately, as soon as you try to impose your will on how your content must be viewed in the users browser then you are setting up more work for yourself at the very least, and at worst causing problems for yourself where there need not be any.

4.24 summary

You should now be able to explain why structure and presentation are treated separately, understand how HTML and CSS are related, and have a basic grasp of the syntax and semantics of CSS. As with HTML (and later Javascript) we have only scratched the surface of what each technology can do. This is a foundation. Exploring what it can do for you is your responsibility.

Chapter 5

HTML Page Layout Using CSS

HTML alone is really great, but pretty ugly. CSS gives us a way to improve how our pages are presented. We can use CSS and semantic HTML to group and organise elements of the UI to give us reusable "blocks" or 'units" from which any given page is assembled.

The last chapter only scratched the surface of CSS and its interaction with, and applications to, Web design. We'll continue our study of CSS, but this time with a focus on the specific ways that CSS supports "laying out" or arranging the elements of a web page to achieve a particular design. You should contrast this with the last unit where we considered how individual typographical elements could be styled, such as the size or colour of lines of text.

Now, instead, we're thinking a little bigger, how all of the elements of a page work together to implement a design. Part of this story will be continued in the next unit when we consider some rules of thumb for designing pleasing user experiences.

For now though we'll consider how any given page can often be considered in terms of a number of blocks. Each block will gather together a number of html elements, for example, the header block, the footer block, the main content or "body" block, and the navigational block. When we start to think of our pages in terms of reusable blocks it can assist us both in designing our pages but also in terms of implementing them as well.

We've seen how HTML gives us essential structure for turning plain text documents into hypertext documents. To some degree HTML also gives us many of the structural elements that we are already used to in terms of word processing text documents. When we decide to make a line into a heading or add a break in our text to create a paragraph, these are similar processes whether we are using a word processor or writing HTML. However, these choices are concerned with capturing, or delineating, the different ways in which the parts of the document work together. Just marking something as a heading or a paragraph doesn't necessarily mean that it must be presented in any specific way. Instead it marks it out so that we can, if so desired, present it in a different way. This marking up process merely facilitates our being able to achieve this.

To actually make a difference to how our HTML document, and the elements that make up that document, are actually seen by a user, we need to consider presentation. We use

CSS to govern the visual presentation of our document, using the various elements of our HTML markup as cues to help us achieve this.

However, whilst CSS is concerned with visual presentation, this is not just making things look pretty. In addition, visual presentation via CSS is an important aspect of both usability and accessibility. By usability we mean all aspects of the design that make the resulting rendered page fit to address the task for which it was intended. Usability can range over many factors including safety, effectiveness, and efficiency. Accessibility is a related topic to usability which is concerned more with ensuring that the resulting design doesn't exclude users from effectively using it. This could include ensuring that a page is accessible to people with specific disabilities, for example, making a page accessible to those with visual impairment might require the underlying HTML to be effectively structured for an assistive technology such as a screen reader to use it. However not all visual impairments are equal, another view on the same aspect of accessibility might require that colour contrast be sufficient between background and foreground elements so that those with minor visual impairments can effectively use the site, whereas colour selection, the choice of colours for a site's colour scheme shouldn't exclude those, or reduce accessibility for those with colour vision deficiency or colour blindness. To root this in the real world consider that red-green colour blindness affects up to 8% of males of Northern European descent.

In the last unit we dealt with styling individual elements but now let's consider how we can group HTML elements together so that they can be managed, laid out, and presented to our users regardless of the device they use. By doing this kind of approach, we also start to gain some strategies for managing a site's design, rather than merely the design of a single page. By reusing groups of elements in similar ways across different pages, we can begin to produce designs that have consistency, and which, as a result, provide a better user experience.

Before we get deeper into this unit, let's also consider just how good the browser is at rendering web pages. Whilst CSS is useful for styling HTML and can make things look great, allowing the browser (user agent) to decide how to place HTML elements on screen is frequently a great idea. The main reason is that this is what the browser, or more correctly, the browser's page layout engine, is designed to do. You can however override this behaviour to get your pages to be laid out and rendered in other ways. CSS will influence how things look and where they are placed. You should recognise though that the more you override the browsers default approach to page layout, the more work you are making for yourself and the more likely you are to run into cross-browser inconsistencies that can lead to different behaviour on different platforms.

5.1 Rendering Pages

Our browser is designed to render web pages. That is, it retrieves the HTML for the page associated with the current web address, then retrieves any linked resources including CSS and JS as well as things like images. During this process the browser will parse the HTML and create an internal DOM representation of the page, modifying this

as necessary to account for any directives from the CSS or any modifications imposed by the JS.

Unless the CSS or JS overrides the default layout that the browser uses then most pages are laid out fairly simply. Assuming a left-to-right reading order, rendering begins at the top left corner and the page flows from there in the order in which it is defined in the HTML document. This is generally a straightforward process which, as a rule, is only hindered by images of unknown dimensions which take longer to download than the rendering engine takes to layout the page according to the known information. This is one reason why you might occasionally see pages that seem to change where things are located as images appear, because the dimensions weren't specified in the HTML and could only be finalised once the associated image file was fully retrieved.

Sometimes though, we really want to have more control over our layout. We want to be able to create a block, or grouping of items, that are placed in a specific location on screen, e.g. to the left, or right, or above, or below other items. Essentially we want to place our HTML elements in particular locations on screen where the normal layout engine, uninterrupted, wouldn't normally put them. This creates tension because browsers are built around layout engines which are optimised for presenting documents, particularly documents that have a quasi-academic publication style structure including, for example, an introduction, various sub-sections, and a conclusion.

Any other form of screen layout, such as a desktop application style user interface, will potentially cause issues for the layout engine. For example, requiring multiple passes to place things in their correct position, or even not being able to consistently and comprehensively satisfy all of the constraints that the design imposes. Remember also that the rendering engine is trying to, by default, create the best layout for the current combination of screen quality and window dimensions, given the content that has to be displayed, and, for the most part, the current generation of rendering engines do a really good job when designers respect this.

We should note that text is inherently responsive by default. It can keep wrapping to the next line until you reach the end of the content. Text resizes nicely to account for all devices, screens, viewports, and window sizes, resolutions, pixel densities, and dimensions. It just keeps flowing until the page is done. However when we try to use the same rendering engine for more exacting layouts, trying to implement desktop style interfaces for example, difficulties can, and frequently do, arise.

5.2 Layouts: HTML + CSS

Let's now look at a way to do layouts with CSS and HTML versions earlier that HTML5 that you will see in the wild, and which work, but that are no longer the preferred approaches. Note that this is for historical usage only; we'll look at some better ways to implement modern designs using HTML5 and CSS, but currently you also need to be aware of the approaches that are still out there. Because many sites have used these approaches, they will be around for a long time, and you might be called upon to maintain such a site in your career.

You can use HTML ¡span¿ and ¡div¿ elements to assemble page layouts. These elements work in different ways and introduce two useful notions to our design toolset. The first is the idea of an inline element, something that works within the context of, or inline with, another element. For example, given a paragraph of text, if you want to have a specific sub-string treated in a different way, such as coloured for emphasis, then an inline element is useful to mark up the start and end of the sub string. The span element is just such an inline element and that's exactly how we might use it. Contrast that with the div element which enables us to wrap parts of our page into a block, to group elements, so that we can treat the contents of the block differently from the rest of the page.

If we wrap these inline and block elements around the different groups of content of our page we can then target those groups with specific CSS. This is achieved by given each inline and block element a unique ID attribute so that they can be identified, referenced, and subsequently positioned and styled by CSS. Note that to do this, HTML requires IDs to be unique.

You'll see this pattern in lots of web sites built prior to HTML5. For quite a while this was an effective way to control the layout of a page but it had, and still has, drawbacks. The main drawback is that your choice of ID for each grouping might not have the same semantic communicative ability as the IDs that others are using. Also the sheer variety of ways in which we can say the same thing, e.g. referring to the navigation bar which we could call "navigation bar" or just "navigation" or "navbar" or "nav-bar" or merely "nav". If it is a panel down the side then we can add all the combinations of nav and panel to the list above. Instead it is best to use semantic HTML tags to achieve this kind of layout control when possible.

But what if you have groupings that dont fit to standard semantic tags? For example, for columns of content in a multi-column layout for ultra-wide monitors. At this point, the use of div blocks makes sense again, as a way to take us beyond the language that HTML 5 semantics define, but not as a starting place.

5.3 How Not To Lay Out Pages

Using div and span tags to organise a page layout isn't the only way that web developers have tried to produce pixel perfect layouts and desktop style user interfaces in the past. So let's quickly survey some of the approaches for how not to do it.

One method was to misuse HTML tables. Because tables arrange things semantically into columns and rows, it is an easy mis-step to get from there to the idea that this semantic use can extend to the visual presentational use. However tables are, just like most default HTML elements, inherently responsive, they will reflow to best fit the screen and window that they are rendering for. This means that changing the window dimensions can alter the layout of the page contents as the text reflows. A clever hack involved combining a table with single pixel transparent gif image files. As they are transparent, the user doesn't see the files, but by using many of them they can be used to push page content around so that it is laid out just as the designer intended for it to be seen visually. However, this is a horrible solution to the problem of getting HTML layout engines to

do something they werent designed to. The main problem being that this is not easily maintainable and can have lots of side-effects for accessibility because a page might now incorporate many thousands of instances of transparent gifs. Think about it, to indent a line of text by 300 pixels will need 300 single pixel gifs.

In comparison to this, the use of divs and spans to group elements then CSS to arrange them is much better; whilst this is still horrible and severely breaks semantic HTML the impact is less terrible but far from ideal.

5.4 Some Common Layouts

Before we get to actually using CSS to lay out our pages, let's just consider some basic layouts that are quite common.

The default for the web is as a single column of text which reflows according to the dimensions of the window that it is rendered to. This is inherently reliable, stable, and responsive. However, as the size of page content increases, this can become a little unwieldy, not necessarily unusable, but, for example, the amount of scrolling required to reach the top or bottom of the page can increase significantly. Similarly, such a page might require the user to find the exact hyperlink, embedded in the page, to enable them to navigate to other pages in the site.

This leads to the category of layouts that involve two parts, the content part, and the navigational part. By grouping all of the links to other pages in the site together, a consistent navigation experience can be designed and replicated relatively easily across all pages in the site so that your users know where to look to find the next place to navigate to. This is generally laid out in one of four ways, with navigation above, or below, or to the left of, or to the right of, the page content. Frequently this leads to a two column layout, with a navigation bar to the left, or right, and the content next to it. This might also include CSS directives to stop the navigation element from scrolling when the rest of the content is bigger than the screen.

Yet another, reasonably common layout, again with multiple possible variations is the three column layout, including a navigation bar and content, but now with the addition of some kind of sidebar which perhaps incorporates additional data or metadata about the current page.

You are only really limited by practicality in terms of potential layouts involving some combinations of multiple rows or columns. Note however that a common extension of these layouts is also to include a header and, or, footer area at the top and, or, bottom of the screen respectively. These might well also combine with the navigational aspects, depending upon your design goals for the page.

Additionally, consider that many sites will also incorporate some dynamic elements in their page layout, for example, enabling hide-able panels or modals that can be shown or hidden as necessary.

5.5 CSS-based Layout Solutions

Two recent solutions give better control over relative placement of HTML elements than using divs and spans and can enable us to realise some of the layouts we've considered whilst maintaining much of the basic flexibility and responsiveness of traditional HTML. These include the FlexBox and the Grid layout. Each is potentially applied to multiple collections of elements to place them relative to each other on screen during rendering.

The Flexbox is a way to lay out linear collections of elements either horizontally or vertically. That basically means placing collections next to each other, from the left edge of the screen until the right edge is met, then wrapping around to put the next collection below the others and continuing to place collections in this manner, wrapping around as needed, until everything is laid out. If we consider each collection to be a word in a sentence, which is really a collection of letters, then this is just like how a sentence is placed on a page, running from left margin to right, and wrapping around into another line each time it reaches the right margin.

The Grid Layout is similar, but instead of working in a single, linear dimension, it works in two dimensions, and is a way to lay out matrices of elements as rows and columns. In some ways this is just a like a table, but recall that an HTML table is for organising matrices of information semantically, a Grid layout is for presenting matrices of information visually. This means that if we want to enforce that a certain group of information is always vertically aligned above or below another group, or immediately to the left or right of another group, we now have a mechanism to do so.

Both the Flexbox and Grid are designed to use CSS to influence the visual placement of HTML elements so that we can avoid hacks like using the transparent gif when attempting to implement our desired layouts.

When considering the use of FlexBox and Grid layout we should take into account the following:

- How we build interfaces there are certain recurrent patterns of user interface layout as we considered earlier.
- How pages are rendered from the upper left corner of the screen to the lower right corner of the screen.
- That the screen has two dimensions the horizontal dimension and the vertical dimension.

So, in some cases we want to lay out things in one single dimension, either horizontally or vertically, but not both at the same time. This leads naturally to the idea of the FlexBox. A layout for a collection of items along a single dimension. Where the FlexBox works either horizontally or vertically, if we add the second, complementary, dimension, then we arrive at the Grid layout.

Let's now consider each approach in turn, starting with the FlexBox...

5.6 FlexBox

Imagine a box reaching horizontally from one side of the screen to the other. This box can contain HTML elements, either as individuals, or in groups. We want to be able to control how elements are spread across the box from one side of the screen to the other. For example, we might want to control the amount of space to either side of, and between, the elements in the box.

Because our computer screens are not infinitely wide or high, we can't have our box containing infinite elements. At some point, if we have sufficient content, we reach the edge of the screen, and the responsive nature of our browsers rendering engine will attempt to reflow the content to start on the next "line" below. If our screen is too narrow for the content we want to display then we might also want to be able to exercise a little control over how and when the box flows over into another row and arranges elements below.

Note that there are some minor differences between horizontal and vertical arrangements because we are habitually used to scrolling up and down a page, but less used to scrolling horizontally, so reflowing content that is too wide for the screen so that it is displayed below seems sensible, but for vertical arrangements, we expect to just keep scrolling down until we reach the end.

Flexbox lets us do this. It supports a range of CSS properties:

- flex
- flex-basis
- flex-direction
- flex-flow
- flex-grow
- flex-shrink
- flex-wrap

For each property you should investigate the documentation associated with it in the Mozilla developer Network CSS reference[1]. Let's now consider some examples to see what FlexBox looks like and how it can be used.

5.7 Flexbox Resizing Example

This example is meant to demonstrate a very simple collection of content and how FlexBox can manage the arrangement of that content when a window's dimensions are altered. We'll use FlexBox to style an unordered HTML list.

[1] `https://developer.mozilla.org/en-US/docs/Web/CSS/Reference`

First, some HTML (flex1.html) for our page. It's worth taking a look at how this is rendered without the CSS so that you get an idea of what it looks like in isolation without any CSS applied. This will let you see just how much functionality the CSS Flexbox declaration adds.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
      <title>CSS Flexbox</title>
      <link rel="stylesheet" href="flex1.css">
  </head>
  <body>
    <ul class="flex-container">
      <li class="flex-item">1</li>
      <li class="flex-item">2</li>
      <li class="flex-item">3</li>
      <li class="flex-item">4</li>
      <li class="flex-item">5</li>
      <li class="flex-item">6</li>
    </ul>
  </body>
</html>
```

Note that we just have a simple HTML document, with a link to an external CSS file. In the body of our document we merely have an unordered list whose contents are the numbers 1 through 6. This is merely to make it easy to see and compare how the contents of the FlexBox are placed and rendered. See that the ¡ul¿ tag has a flex-container class attribute and the individual items are given flex-item class attributes. Now let's get to the CSS (flex1.css).

```

.flex-container {
  padding: 0;
  margin: 0;
  list-style: none;
  display: flex;
  flex-flow: row wrap;
  justify-content: space-around;
}

.flex-item {
  background: tomato;
  padding: 5px;
  width: 200px;
  height: 150px;
  margin-top: 10px;

  line-height: 150px;
  color: white;
  font-weight: bold;
  font-size: 3em;
  text-align: center;
}
```

All we have done here is apply styling to the unordered list container, indicating that we want it treated as a flex container with the display:flex and flex:flow directives. We've also applied a style for individual items within the container. This is all standard CSS and not specific to the FlexBox container but merely changes colour and font settings, as well as adding padding and sizing information to the elements.

This should give us a set of 6 boxes which will reorganise themselves to best use the space available in the enclosing window. Here is what it should look like but you should really try it out for yourself and experiment with things to get a feel for how the FlexBox acts.

Figure 5.1

5.8 Flexbox Navbar

One useful application of the FlexBox is to help implement a navigation bar, or navbar, which will reflow to make the best use of the enclosing window whilst remaining enclosed and isolated from surrounding HTML elements in the same document. This is a common layout design that is worth having in your toolbox.

Let's see what a simple navbar would look like using the FlexBox. First some HTML (flex2.html):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
      <title>Flexbox Navbar</title>
      <link rel="stylesheet" href="flex2.css">
  </head>
  <body>
    <ul class="navigation">
        <li><a href="#">Home</a></li>
        <li><a href="#">About</a></li>
```

```
        <li><a href="#">Products</a></li>
        <li><a href="#">Contact</a></li>
    </ul>
  </body>
</html>
```

This is fairly straightforward HTML content. Like our last example, an unordered HTML list, but instead of merely numbers as content, we've named each item according to how it might be named for a simple online business, and turned each item into an empty hyperlink because the purpose of a navbar is to help us to navigate to different locations in the site. Now let's investigate the CSS (flex2.css) to style this as a FlexBox navbar. It's a bit more complicated than the simple demonstration earlier. If in doubt look up the CSS declarations that you are unfamiliar with in the MDN CSS reference.

```
.navigation {
    list-style: none;
    margin: 0;
    background: tomato;
    display: flex;
    flex-flow: row wrap;
    justify-content: flex-end;
}
.navigation a {
    text-decoration: none;
    display: block;
    padding: 1em;
    color: white;
}
.navigation a:hover {
    background: darken(tomato , 2%);
}
@media all and (max-width: 800px) {
    .navigation {
        justify-content: space-around;
    }
}
@media all and (max-width: 600px) {
    .navigation {
        flex-flow: column wrap;
        padding: 0;
    }
    .navigation a {
        text-align: center;
        padding: 10px;
        border-top: 1px solid rgba(255,255,255,0.3);
        border-bottom: 1px solid rgba(0,0,0,0.1);
    }
    .navigation li:last-of-type a {
        border-bottom: none;
    }
}
```

Part of this complexity is just to manage things like the display of hyperlinks in the navbar. Now let's see how that looks:

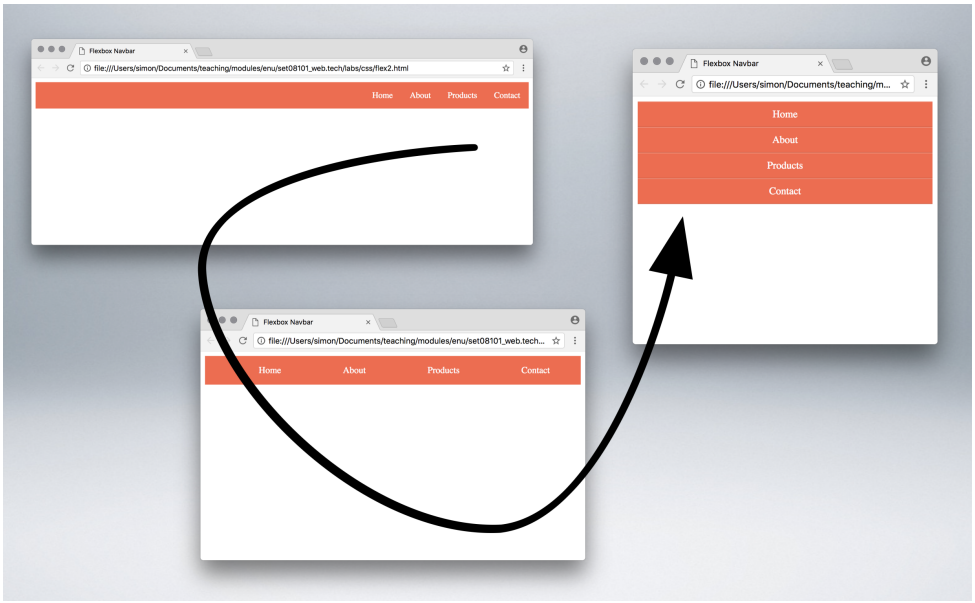


Figure 5.2

5.9 Flexbox Page Layouts

There's nothing to stop us going further than merely creating a flexible navigation bar. Instead we could use a flexbox to lay out our entire page. Let's give that a go. Note in the example how we are now using semantic HTML5 elements like the header, article, aside, and footer. Note that we're also wrapping everything in a div so that we can apply the display: flex declaration to all of the page's body content. First the HTML (flex3.html):

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
      <title>3-column layout + full-width header and footer</title>
      <link rel="stylesheet" href="flex3.css">
  </head>
  <body>
    <div class="wrapper">
      <header class="header">Yar Pirate Ipsum</header>
        <article class="main">
            <p>Deadlights jack lad schooner scallywag dance the hempen
               jig carouser broadside cable strike colors. Bring a
               spring upon her cable holystone blow the man down spanker
                Shiver me timbers to go on account lookout wherry
               doubloon chase. Belay yo-ho-ho keelhaul squiffy black
               spot yardarm spyglass sheet transom heave to.</p>
        </article>
        <aside class="aside aside-1">Aside 1</aside>
        <aside class="aside aside-2">Aside 2</aside>
        <footer class="footer">Footer</footer>
    </div>
  </body>
</html>
```

Note that we're using some "placeholder text" to fill out the various parts of the interface so that the layout engine in the browser actually has something to lay out. Now let's take a look at the CSS (flex3.css):

```
.wrapper {
  display: flex;
  flex-flow: row wrap;
  font-weight: bold;
  text-align: center;
}

.wrapper > * {
  padding: 10px;
  flex: 1 100%;
}

.header {
  background: #CC3F0C;
}

.footer {
  background: #9A6D38;
}

.main {
  text-align: left;
  background: #33673B;
}

.aside-1 {
  background: #D8CBC7;
}

.aside-2 {
  background: #D8CBC7;
}

@media all and (min-width: 600px) {
  .aside { flex: 1 0 0; }
}

@media all and (min-width: 800px) {
  .main    { flex: 4 0px; }
  .aside-1 { order: 2; }
  .main    { order: 3; }
  .aside-2 { order: 4; }
  .footer  { order: 5; }
  .header   {order: 1;}
}

body {
  padding: 2em;
}
```

Now, let's see how that looks. You should be able to recognise a three column layout with full width headers and footers. We should have a resize-able layout that reflows into

a sensible order when the window becomes too narrow to accommodate the horizontal layout of the two asides and the main content.

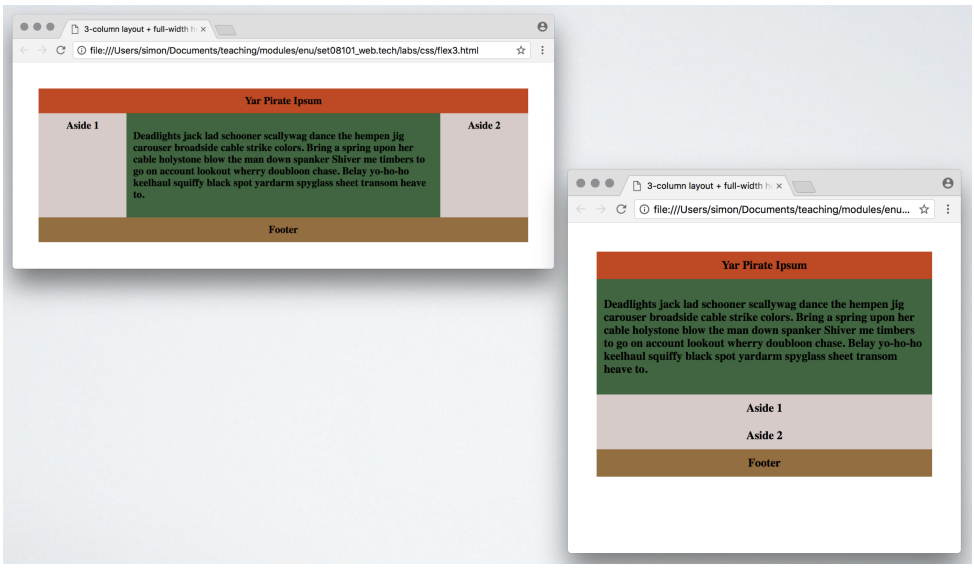


Figure 5.3

5.10 Flexbox User Interface

We can refine this layout easily. With the addition of a slightly nicer colour palette, a little differentiation in text sizes in the various sections, and using the first aside as a navigation panel, we now have something that is a little more visually pleasing. First some HTML (flexbox4.html):

```
<!DOCTYPE html>
<html lang="en">
  <head>
      <meta charset="UTF-8">
      <title>Three Column Layout</title>
      <link rel="stylesheet" href="flexbox4.css">
  </head>
  <body>
      <header>Header</header>
      <div class="container">
          <main>
              <h1>Main Content</h1>
          </main>
          <nav>
              <h4>Navigation</h4>
          </nav>
          <aside>
              <h4>Aside</h4>
          </aside>
      </div>
      <footer>Footer</footer>
  </body>
</html>
```

Compare this approach to the previous example, again to show you the flexibility of how we can build relatively similar interfaces with different combinations of HTML facilitating the implementation. Note how the header and footer elements are this time outside of the container div that manages the flex layout, i.e., the header and footer are still full width elements at the top and bottom of the screen but are no longer part of the flex layout itself, only the main, nav, and aside elements are part of the FlexBox. Let's take a look at the CSS (flexbox4.css):

```
body {
  margin: 0;
  padding: 0;
  max-width: inherit;
  background: #FFF9F9;
  color: #1C1C1C
}
header{
  font-size: xx-large;
  text-align: center;
  padding: 0.3em 0;
  background-color: #F2EDED;
  color: #1C1C1C;
}
footer {
  font-size: medium;
  text-align: center;
  padding: 0.3em 0;
  background-color: #F2EDED;
  color: #1C1C1C;
}
nav { background: #EDE6E6; }
main { background: #D8DDDE; }
aside { background: #EDE6E6; }
body {
  min-height: 100vh;
  display: flex;
  flex-direction: column;
}
.container {
  display: flex;
  flex: 1;
}
main {
  flex: 1;
  padding: 0 20px;
}
nav {
  flex: 0 0 180px;
  padding: 0 10px;
  order: -1;
}
aside {
  flex: 0 0 130px;
  padding: 0 10px;
}
```

Now let's see how that looks:

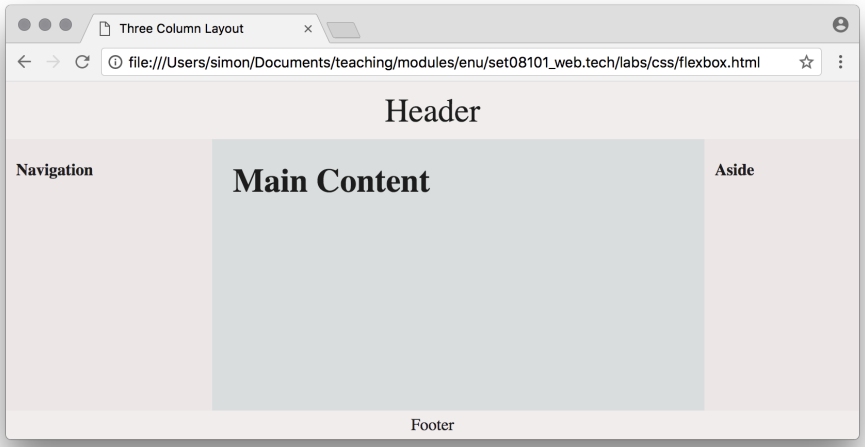


Figure 5.4

5.11 Grid Layout

A limitation of the Flexbox is that it works in one dimension. Items in a flexbox are laid out in a line if the page is wide enough and wrapped around as much as needed otherwise. However there is no way to easily force an item to be above or below any other item. That kind of behaviour is within the purview of two-dimensional layout tools, like the Grid layout. Instead of laying out items along a single row, what if we laid out our html elements in rows and columns? Then we could position things above and below each other on screen, independent, to a large degree, of the width of the bounding window.

As we mentioned before, this is similar in many ways to the HTML table. However, the HTML table has a semantic meaning, it is concerned only with the organisation of data so that data within the table is related in various ways and organised according to rows and columns. HTML tables have no visual or presentational meaning in terms of how the resulting table should be displayed, a grid layout is almost the opposite in some ways or rather is a presentational counterpart to the HTML table. On this interpretation, the grid layout has a presentational meaning but the members of the grid have no semantic relation beyond their placement relative to each other, e.g. grid elements in the same row or column arent necessarily related and we can't infer any relationship between elements of a grid layout from how it is organised.

5.12 Using the Grid Layout

We use Grid by setting the display: grid; property on a containing element, e.g. a ¡div¿ or even the ¡body¿ of the HTML document. This is similar in application to how we used the FlexBox. Essentially it means that we define a containing HTML element to have the grid display property. Then Grid will be used to layout the child elements within that containing element.

Grid has a number of related properties and values that can be used to control the layout of the grid, e.g. spacing between elements, flow of elements, arrangement of elements.

The Grid layout supports a range of CSS properties:

- grid
- grid-area
- grid-auto-columns
- grid-auto-flow
- grid-auto-rows
- grid-column
- grid-column-end
- grid-column-gap
- grid-column-start
- grid-gar
- grid-row
- grid-row-end
- grid-row-gap
- grid-row-start
- grid-template
- grid-template-areas
- grid-template-columns
- grid-template-rows

For each property you should investigate the documentation associated with it in the Mozilla developer Network CSS reference[2]. The Grid layout also gives us two basic organisational forms for our layouts, regular or irregular. These are differentiated based upon whether every row and column in the grid has the same number of elements, if they do, then the grid is regular. Otherwise it is irregular. Not all of the layouts that we might want to design will be regular, hence the irregular approach is quite useful when we design web pages. Let's look at examples of each in turn.

[2] `https://developer.mozilla.org/en-US/docs/Web/CSS/Reference`

5.13 Regular Grid Layouts

The behaviour of a grid layout is different when resizing a window compared to the same content using the FlexBox property. Compare the behaviour of the following when a window is resized to that of the FlexBox example earlier.

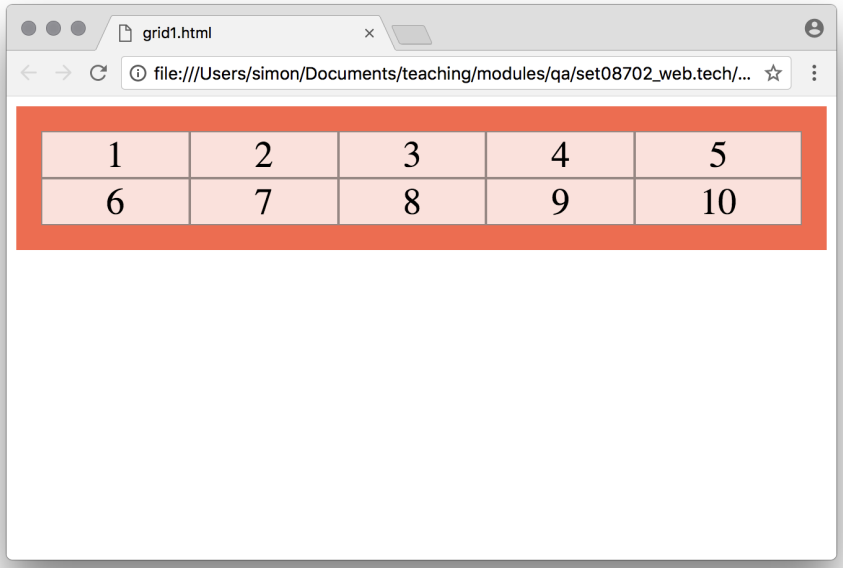


Figure 5.5

Now with a narrower window:

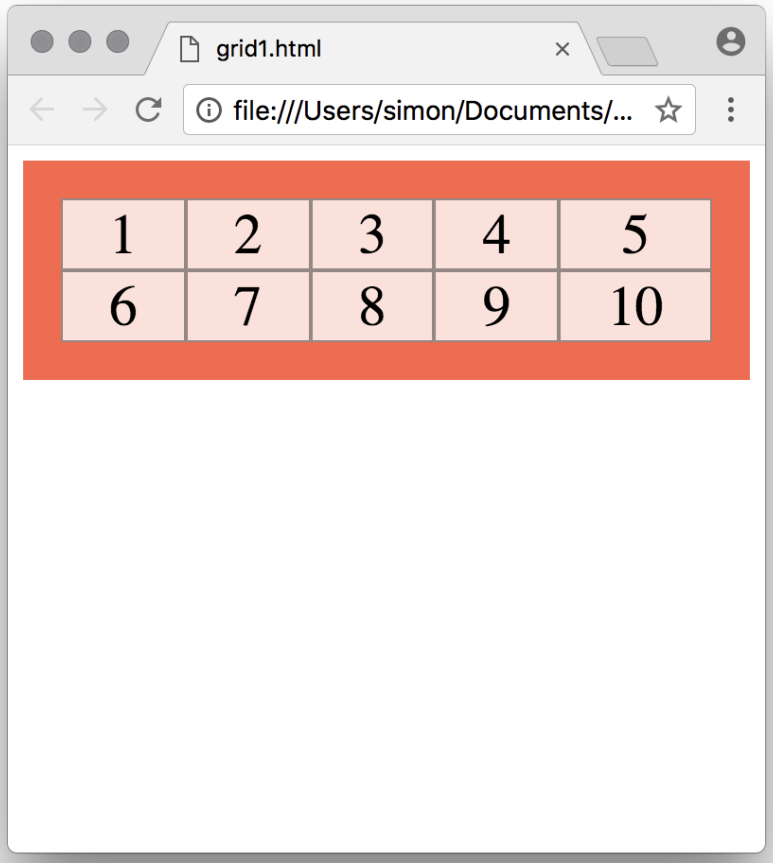


Figure 5.6

Notice how it doesn't reflow in the same way. Let's look at how we can implement this in HTML (grid1.html):

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="grid1.css">
  </head>
  <body>
    <div class="grid-container">
        <div class="grid-item">1</div>
        <div class="grid-item">2</div>
        <div class="grid-item">3</div>
        <div class="grid-item">4</div>
        <div class="grid-item">5</div>
        <div class="grid-item">6</div>
        <div class="grid-item">7</div>
        <div class="grid-item">8</div>
        <div class="grid-item">9</div>
        <div class="grid-item">10</div>
    </div>
  </body>
```

```
20 </html>
```

This is a fairly straightforward HTML page whose body contains a div which acts as our container for the items that are meant to be contained by the grid. Each item, in this example, is a piece of text, a number, to differentiate the items from each other. These are then wrapped in div tags and given the grid-item class property so that we can attach some CSS to them. Let's take a look at that CSS(grid1.css):

```
1  .grid-container {
2    display: grid;
3    grid-template-columns: auto auto auto auto auto;
4    background-color: tomato;
5    padding: 20px;
6  }
7  .grid-item {
8    background-color: rgba(255, 255, 255, 0.8);
9    border: 1px solid rgba(0, 0, 0, 0.4);*
10   padding: 10px;
11   font-size: 30px;
12   text-align: center;
13 }
```

Our CSS is fairly straightforward, some styling for the grid-container, and some styling for the individual grid-items. Note how the container has the display: grid property and we've set the grid-template-columns property to auto so that the browser can determine how big to make the columns based upon the size of the container and the size of the content of each item in the column. We specify the number of columns to create using the auto value in the grid-template-columns property, each declaration of auto specified an additional column, so we end up with five columns in this example.

5.14 Irregular Layouts

For an irregular layout, which is most useful when designing user interfaces, we can enable individual elements within the Grid to span multiple rows or columns. We achieve this by using the grid-column-start and grid-column-end properties. Let's look at an example that will produce something like the following:

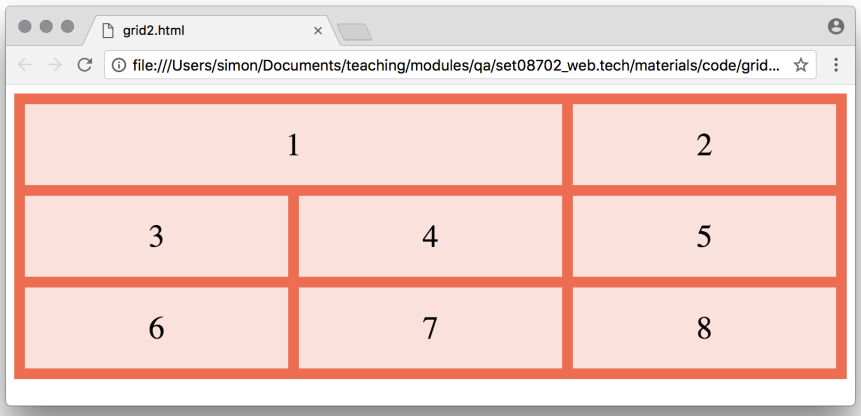


Figure 5.7

This can be very useful to give us a pleasing arrangement of the "logical units" that make up the semantic content of our page. First some HTML (grid2.html):

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="grid2.css">
  </head>
  <body>
    <div class="grid-container">
        <div class="item1">1</div>
        <div class="item2">2</div>
        <div class="item3">3</div>
        <div class="item4">4</div>
      <div class="item5">5</div>
        <div class="item6">6</div>
        <div class="item7">7</div>
        <div class="item8">8</div>
    </div>
  </body>
</html>
```

Notice that there is very little difference in this HTML compared with that from the regular Grid layout example earlier. The only real differences are that there are two fewer items in this example, and that each item has its own individual class ID. Now the CSS (grid2.css):

```
.grid-container {
  display: grid;
  grid-template-columns: auto auto auto;
  grid-gap: 10px;
  background-color: tomato;
  padding: 10px;
}

.grid-container > div {
  background-color: rgba(255, 255, 255, 0.8);
```

```
  text-align: center;
  padding: 20px 0;
  font-size: 30px;
}

.item1 {
  grid-column-start: 1;
  grid-column-end: 3;
}
```

Here we've done almost exactly the same as in the regular layout example but have specified three columns using grid-template-columns: auto auto auto; that is three "autos", one for each column. Note that whilst each item has its own individual class ID, we only had to provide individual CSS for item1 as it is the only one that we wanted to behave differently. Everything else is subsequently laid out using the default Grid layout for a three column grid.

5.15 Grid Layout for the User Interface

Our examples so far have just looked at how the Grid layout is used. But what about applying it to the design of a user interface layout. This is actually quite similar to our Flexbox Based layout except that it won't reflow in quite the same way as for FlexBox when the window is resized.

At the expense of reflow flexibility and responsiveness, Grid instead gives us more control over positioning of elements relative to others within the grid. Useful when we want UI elements to appear in specific locations relative to each other.

A reminder though that the more we try to override the default HTML layout engine in the browser the more effort we must extend to return to the previous responsiveness of raw HTML. Let's take a look at some HTML (grid.html):

```
<!DOCTYPE html>
<html lang="en">
  <head>
      <meta charset="UTF-8">
      <title>Modern CSS</title>
      <link rel="stylesheet" href="grid.css">
  </head>
  <body>
      <header>This is the header.</header>
      <main>
          <h1>This is the main content.</h1>
          <p>Abluo probo dolore vulpes. Et multo nullus magna inhibeo ex
              ex quidne conventio sudo pneum. Hendrerit duis vicis at
             qui iaceo iaceo ut consectetuer reprobo meus.Tincidunt
             nulla eros erat aliquip laoreet ingenium wisi valde eum.
             Uxor vel qui validus comis aliquam nutus facilisi feugiat
             pala.</p>
    </main>
      <nav>
          <h4>This is the navigation section.</h4>
```

```
16          <p>Pala consequat defui decet at accumsan adipiscing facilisis
                quis nibh.</p>
17      </nav>
18      <aside>
19          <h4>This is an aside section.</h4>
20          <p>Singularis ullamcorper nimis validus enim genitus demoveo
                voco et causa. Incassum incassum appellatio pagus augue
                vindico at in. Quis qui distineo. In abigo consectetuer.
                Esse populus virtus nisl tum ut nulla ideo vel qui.</p>
21      </aside>
22      <footer>This is the footer.</footer>
23  </body>
24 </html>
```

This should now be looking fairly familiar. An HTML document using some of the HTML5 semantic elements to define the different parts of our interface that we want to lay out. This layout is going to be very similar to our FlexBox layout UI example from earlier. A three column layout with fixed, full-width header and footer. The main difference is the control of the layout using Grid instead of FlexBox and hence different reflow behaviour on window resizing events. The associated CSS (grid.css):

```
1  body {
2    margin: 0;
3    padding: 0;
4    max-width: inherit;
5    background: #fff;
6    color: #4a4a4a;
7  }
8  header, footer {
9    font-size: large;
10   text-align: center;
11   padding: 0.3em 0;
12   background-color: #4a4a4a;
13   color: #f9f9f9;
14 }
15 nav {  background: #eee; }
16 main {  background: #f9f9f9; }
17 aside {  background: #eee; }
18 body {
19   display: grid;
20   min-height: 100vh;
21   grid-template-columns: 200px 1fr 150px;
22   grid-template-rows: min-content 1fr min-content;
23 }
24 header {
25   grid-row: 1;
26   grid-column: 1 / 4;
27 }
28 nav {
29   grid-row: 2;
30   grid-column: 1 / 2;
31   padding: 0 10px;
32 }
33 main {
34   grid-row: 2;
35   grid-column: 2 / 3;
```

```
36    padding: 0 20px;
37  }
38  aside {
39    grid-row: 2;
40    grid-column: 3 / 4;
41    padding: 0 10px;
42  }
43  footer {
44    grid-row: 3;
45    grid-column: 1 / 4;
46  }
```

Which should yield something like the following: A good basic structure for implementing a flexible three column page layout.

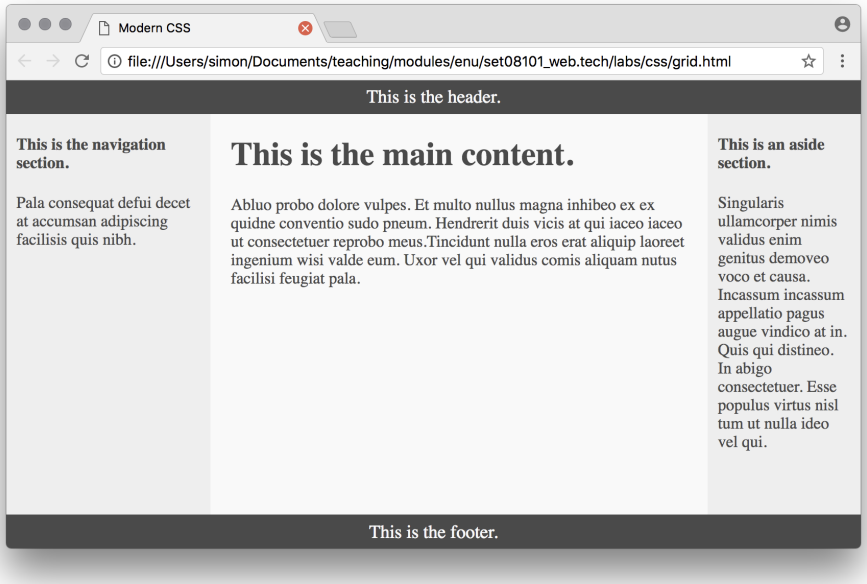


Figure 5.8

5.16 Summary

We should all now have some idea of the various approaches that we can take to place HTML elements within a page. However, under some circumstances, we may want more control over the specific layout of our page and some control over the relative layout of semantic groupings of elements on our page. We can use the Grid and FlexBox layouts to enable this control.

Beware that beyond this point, increasingly trying to attain pixel perfection in your layout is a road to hurt. This is especially so if you want things to look and act similarly in different browsers which use slightly different layout engines, and the worst case is if you also want similar behaviour in older browsers which can lead to lots of CSS hacks. To my mind CSS hacks, although clever, often feel like a code smell. When we notice them

we should perhaps relax and let the browser do more of the work instead of wrestling with it.

Chapter 6

Design for Hackers

Some true things[1]:

- The best tv shows have a song and dance episode
- All the best reggae starts with a drumroll...
- The majority of coders say that they cant do design

If you write code then you are a designer so we'll look at some "hacks" to help you grow your confidence when faced with visual design tasks.

As many computing students, and programmers in particular, don't consider themselves to be designers, we'll use this chaper to look at a bunch of ways, shortcuts, rules of thumb, and hacks that can help you to design and implement appealing websites. Along the way we'll consider elements of colour theory, typography and design standard, as well as look at the importance of design in terms of effective communication.

Our overall aim is to:

- Remedy any notion you have that design is too hard
- Assemble some simple heuristics to help build acceptably presented websites

6.1 Designing & Coding

Let's start by setting some boundaries. We're not aiming, in this module, to build the worlds most beautiful websites. That said, you can if you want, but it's not the default end goal. The main reason for this is that design is, to a large degree, its own independent domain with a largely independent skill set to those that we develop as programmers and technologists. Just as we don't expect designers to have top-level programming skills, we don't expect programmers to have similar design skills, although that's not to stop you striving to develop better design skills. Partly that will result from practise, partly from learning recognised skills and building knowledge, and partly from critical reflection, so learning design is really like learning anything else, the same processes to follow. The

[1] in my opinion

problem is that for many programmers and developers, they are rarely formally taught how to design, so they believe that they can't do it. The truth is that it is not that you can't do design, it's that you've not been taught to do it.

However, there is a caveat, the best designers produce great work because they practise, and they work really hard to ensure that every element is perfect and beautiful. They spend a lot of time producing designs in order to get better at producing designs, just like you put a lot of time into writing code so that can get better at producing code. So you will have to put some time and effort into the design aspect of your work if you want to see improvement.

One thing that we can learn from designers practise is the idea of critical evaluation. Designers critically evaluate other peoples work to see what they can learn and reuse in their own practise. When was the last time you looked closely at a site that you thought was visually pleasing and noted how the designers achieved the things you appreciated. Similarly, when did you last try to work out how to get a similar effect in your own sites? This could be extended to ask the same question about code. When did you last see some example code, perhaps on Stack Overflow or GitHub, and really tried to understand how it worked, tore it apart and reassembled it, to thoroughly know it?

If it's been a long time, or never, then why? Perhaps now is the time to start learning from those examples, so as you use the Web from now on, start looking at how the pages work from a visual and design perspective so that you can see what works. Try to develop an understanding for why some things work and why other approaches can be less effective. Use this process to build your own mental catalogue of approaches as a starting place for your own designs. Over time and with experience, just as with coding practise, you'll find that you spend less time looking for inspiration and more time just being inspired to create from scratch. This is because of the secret that designs rarely emerge from nothing, but frequently emerge from experience.

6.2 Design is important

In previous units Ive referred to CSS as "making things look pretty" and CSS can be just be that. But design can be much more important than just prettifying our sites. CSS is only an aspect of design that is mostly concerned with the visual presentation. So CSS is an aspect of design, but so is HTML, and, for that matter, so is JavaScript. When we decide how to represent a particular body of information as an HTML structure, then we are doing a form of design. Similarly, when we write code, like JavaScript, we design how that code works. We create functions and objects, we decide how the information flows between them, and we determine how to represent facets of our problem domain in terms of program state. Doing this well is a design task. So I'll argue that you're already a designer, just that your toolbox for visual design is a little emptier than your toolbox for other coding skills.

I suggested above that design is important, and that all of the coding processes we follow involve different forms of design. I'll now go further and state that we cant develop information systems without considering design. Every development task has some

element of design regardless of whether the results are made public or how many users there are. Good design helps us avoid mistakes and reduce errors, whether on the part of the user of the system, the quality and accuracy of the inputs and outputs, or the longer term manageability of the system.

Design helps to make systems both usable and accessible. In terms of real world situations, this can help avoid customer service calls or mistake fixing or error correction which in turn can help preserve time, user-base, revenue, and reputation.

6.3 The Hawaii Alert

This is a decent example of what can go wrong when design isn't considered. In 2018 a ballistic missile alert was accidentally issued via the Emergency Alert System and Commercial Mobile Alert System in Hawaii.

Figure 6.1

It turned out that this was a mistake and there was not a missile. It wasn't a drill either. It was a mistake. One that stemmed partly from poor design of the web interface that could be used to create such an alert.

We might consider that such incidents are rare, but once you start looking for them, it turns out that they happen more often than you'd imagine. Only 3 days after the Hawaii alert, a similar incident happened in Japan and alerted 300,000 subscribers to the NHK News and Disaster Prevention service that a North Korean missile had been fired. This was incorrect. In 2019 emergency sirens went off on Oahu and in 2020 Ontario in Canada mistakenly issued an emergency alert to all television stations and television providers, radio stations, and wireless networks in the province about an incident at a Nuclear power plant. All were false alarms.

This is a screenshot of the Web page used by Hawaii Emergency Management Agency to initiate emergency alerts. It is a pretty good masterclass in how to build a poor design.

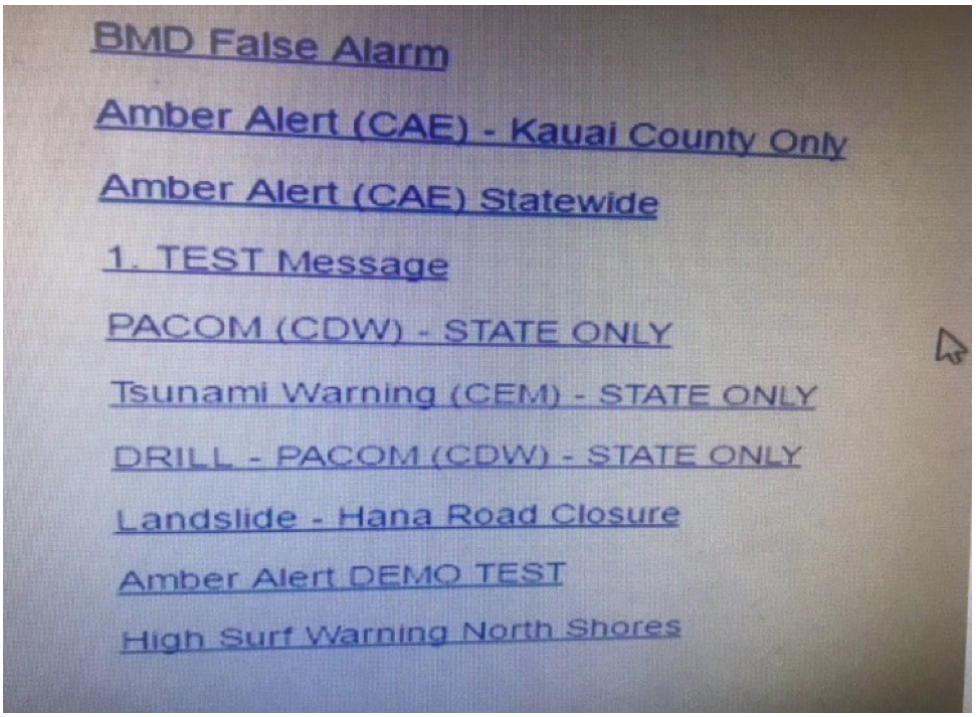


Figure 6.2

How many poor design decisions can you identify?

6.4 Data to Design

Given that we are all technologists it is pertinent to ask, how do we get from data to design? We're likely to be starting with data and a problem space concerning some aspect of that data and the design will follow from that starting point. Hence the notion of a flow from data to design as opposed to starting with a design and manipulating or finding data to fit it.

For the Web, we start by marking up our data with HTML tags. This gives us an idea of how things might be structured. We should also bear in mind that there might be multiple, acceptable structures for the page that are equally good. The outcome is that we end up with at least one set of unstyled HTML pages. Until we are at this point we don't have a site, as both CSS and JS depend upon an HTML page to host them, but the site we do have is quite raw and lacking both visual refinement, from CSS, and possibly helpful additional usability functionality that can stem from JS.

However, if we have HTML then we have something that works. At this point we can read the data and navigate the links. Sometimes this can also help us to start to see navigational paths through the data, and desirable presentational approaches that might have been unclear until we can start to interact with the data as part of a web page.

The default style of raw HTML is, however, quite plain. Perfectly usable, but lacking personality, and possibly without the kind of impact that additional styling can impart. For example, use of colour or adjusting typographical layout can make a huge impact in terms of the users experience.

In addition you will probably also want, or need, to personalise the plain HTML pages. If you're working for a client then they might have branding that needs to be applied, or a corporate colour scheme, or even a full corporate style guide (which we'll consider soon).

Given this scenario, how should we go about design?

Creating sketches, mockups, wireframes, and design documents are a good place to start. The process is simply to come up with ideas, decide which you prefer, then document your decisions. So ideation first, then sketches, mockups, and wireframes to flesh out and illustrate those ideas, then design documents to help document the ideas and provide a bridge to implementation in your finished site.

6.5 Mockups

Let's start our design process with mockups. Rather than go straight to implementing all our design ideas in HTML and CSS straightaway, it can be easier to rapidly prototype the core ideas so that you can test them out before committing too much time and effort.

There are lots of ways to create mockups and no right or wrong approach. It is all about ensuring that you can explore and evaluate your design ideas using quick and lightweight methods.

Many folk design sites on paper. They just draw out their design on paper, indicating where the important parts will be, trying out colour schemes and arrangements of content as necessary. Once this is done, then the design is implemented.

Because paper can be difficult to work with if you make a mistake, alternative low-tech approaches to mockup creating include using a whiteboard. This has the advantage that some areas can be wiped clean and redrawn as the design develops. A useful addition to both paper and whiteboard based approaches is to use post-it notes of various sizes to represent elements and blocks of content.

Other designers use general drawing, image, or graphics programs to draw their designs. Object-based graphics tools, like Omnigraffle, can be useful as well as they enable you to move layers and group things together to rearrange your interfaces.

There are also dedicated mockup tools, either for the desktop or online. Here are a few, some have free access, others not:

- `https://wireframe.cc/`

- `http://mockflow.com/`
- `https://www.invisionapp.com/`
- `https://mockingbot.com/`

You should try some out, and see what works for you. There is no clear best mockup or prototyping tool so the selection of a tool is a personal choice to a large degree. As soon as you start creating mockup though, you have to start making decisions, for example, about content and presentation. So over the next few sections we'll investigate some useful approaches for mocking up text and content, or generating colour schemes.

6.6 Using Placeholder Text or "Greeking"

So far weve assumed that we have data and just need to build a website or mockup to show it. However sometimes we dont have data or content during the design and development phase of a project. This could be for many reasons. It might not be written yet or might not even exist. From the web development perspective all we are concerned with is the fact that we don't have content and yet are still tasked with the job of producing a design or mockup for the site.

It can be very effort intensive to put together realistic text so that layouts look like you want them to.

In fact writing this kind of text is a professional job, that of the copy writer. Note that the "copy writer" should not be confused with copy rights. The first is about the production of content and the second is related to the laws and legal frameworks that ostensibly protect the material produced by a copy writer.

The solution is to not bother with realistic text until you really need it. Instead, we fake it. We create blocks of fake text in a process called Greeking. This is the process of using a dummy text to take the place of text in a design or layout, until the real text is written. Even though you don't yet have the real text, in many projects, there will be a clear idea of how long that text will be, either in terms of a specific word count, such as found in magazine or news style articles, or as a general outline for a given section, i.e. "there will be a short explanatory paragraph here", from which we could conclude perhaps a paragraph of 3 to 4 average length sentences of 15-20 words. This would give a range of 60 to 80 words that would need to be mocked up to temporarily place-hold within the design.

A reasonably standard placeholder text has been used for this job since around the 16th Century, so if you see text like this anywhere:

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur

sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Then you can conclude that the content has been greeked. Note that if you see text like this in a real, deployed website then you can usually conclude that someone has made a mistake. The text itself is section 1.10.32 of Cicero's "*de Finibus Bonorum et Malorum*" from 45 BC.

You might ask yourself, why not just use random or repeated words and text? The main reason is that greeked text doesn't look like normal writing so is less likely to be overlooked and left in the final site but also the distribution of words in terms of combinations and lengths is very similar to modern English. More formally, it is very close to a normal distribution of letters, words, sentences, and paragraphs, so hits the sweet spot of relative realism and difference. Newspaper editors have been using Greeking for a hundred years and because texts are approximately the correct length and organisation, they result in mocked up layouts that look about how they are expected to.

The key is that the placeholder text looks like real text, it has the same rhythms and cadences as real language, and hence real copy. Word lengths and frequencies are similar so we can assume that text of a certain length, the word count, will generally fit the area weve designed for. As you will usually know roughly how many words an article will be, but dont have the actual article yet, this is a good way to enable progress in a project that would otherwise grind to a halt.

6.7 Modern Greeking

Classical Latin can get a little boring. So designers have produced alternative modern versions of greeking. These might better appeal to your personal sensibilities for building personal sites. That said though, the traditional Latin/Greek approach is a good professional baseline for mockups. There are web sites that generate texts of specified lengths that you can incorporate into your mockups. Alternatively, you could just copy and paste sections of the following into your HTML as required when mocking things up.

Why not consider some of the following examples of modern greeking?

Classical Latin *Ut vero torqueo in utinam ludus laoreet ad ex. Et saluto modo molior. Pala consequat defui decet at accumsan adipiscing facilisis quis nibh. Abluo probo dolore vulpes. Et multo nullus magna inhibeo ex ex quidne conventio sudo pneum. Hendrerit duis vicis at qui iaceo iaceo ut consectetuer reprobo meus. Tincidunt nulla eros erat aliquip laoreet ingenium wisi valde eum. Uxor vel qui validus comis aliquam nutus facilisi feugiat pala. Singularis ullamcorper nimis validus enim genitus demoveo voco et causa. Incassum incassum appellatio pagus augue vindico at in. Quis qui distineo. In abigo consectetuer. Esse populus virtus nisl tum ut nulla ideo vel qui.*

Pseudo German *Frankfurter a wunderbar flippin. Ich mitten hinder oof weiner sightseerin heiden. Sauerkraut corkin stoppern haus. Pretzel underbite wunderbar hinder spritz oof rubberneckin sparkin footzerstompen weiner frankfurter rubberneckin. Glockenspiel an*

ich uber pukein glockenspiel cuckoo hans keepin. Blitz makin wunderbar. Nutske sparkin oompaloomp nutske oompaloomp achtung nicht floppern wearin. Blimp corkin oompaloomp nutske undervear buerger er verboten gewerkin blitz buerger pretzel. Haus sparkin underbite uber mitz mitten footzerstompen sparkin die ya. Kaputt auf haben makin corkin gestalt poken buerger das wunderbar.

TV Show Themes *Children of the sun, see your time has just begun, searching for your ways, through adventures every day. Every day and night, with the condor in flight, with all your friends in tow, you search for the Cities of Gold. Ah-ah-ah-ah-ah... wishing for The Cities of Gold. Ah-ah-ah-ah-ah... some day we will find The Cities of Gold. Do-do-do-do ah-ah-ah, do-do-do-do, Cities of Gold. Do-do-do-do, Cities of Gold. Ah-ah-ah-ah-ah... some day we will find The Cities of Gold. One for all and all for one, Muskehounds are always ready. One for all and all for one, helping everybody. One for all and all for one, it's a pretty story. Sharing everything with fun, that's the way to be. One for all and all for one, Muskehounds are always ready. One for all and all for one, helping everybody. One for all and all for one, can sound pretty corny. If you've got a problem chum, think how it could be. 80 days around the world, we'll find a pot of gold just sitting where the rainbow's ending. Time - we'll fight against the time, and we'll fly on the white wings of the wind. 80 days around the world, no we won't say a word before the ship is really back. Round, round, all around the world. Round, all around the world. Round, all around the world. Round, all around the world. This is my boss, Jonathan Hart, a self-made millionaire, he's quite a guy. This is Mrs H., she's gorgeous, she's one lady who knows how to take care of herself. By the way, my name is Max. I take care of both of them, which ain't easy, 'cause when they met it was MURDER!*

6.8 Typography

Traditionally, typography is the discipline of arranging type, that is letters, to make the resulting text legible, readable, and appealing. This also includes the style, arrangement, and appearance of letters, numbers, and symbols, basically all of the textual elements of a page. Typography for the web makes use of typefaces (fonts), point sizes, line lengths, letter-spacing, and line-spacing, essentially giving you a lot of control over how text is displayed to your users on your site. This is all governed by your use of CSS which can be used to specify not only a particular font, but how big the letters should be, how close together individual letters should be, and the space between lines of text. Whilst you can adjust any or all of these elements, as a rule, you'll likely find that you seldom deal with typography at this level. Most browsers include a family of fonts by default that cover most use cases and many designers only select an alternative font if they have a specific need for something different.

The main guidelines for fonts is to ensure that all of the fonts you select work together harmoniously. As a rule there is rarely any need to have more than one or two fonts on a given site, a default font for all textual elements and a secondary, complementary font that is used perhaps for headlines or special circumstances in contrast to the default. Changing fonts frequently within the same page however can be jarring for the users experience and inhibit their smooth consumption of your content.

It you do decide to use alternatives to the default browser font you might want to consider whether your design should be dependent upon a particular font and if not, ensure that it still works well with the defaults available. Remember that accessibility technologies might override your design choices so your site should remain usable and accessible regardless.

6.9 Colour-schemes

How do you pick colours for a site's colour-scheme?

Do you pick colours for things by just choosing your favourite colour? Then when you need another colour, you pick your next favourite? Then when you don't need to pick any more colours, there's your palette?

For most of us, this is a terrible way to create a colour-scheme for one simple reason. We don't, generally have sufficient knowledge, experience, colour sense, or colour discernment to select colours that work together. It turns out though that there is a scientific basis, called colour theory, for why some colours work together and other combinations don't. So, dont just pick colours as and when you need them. I have seen some very ugly colour schemes that have resulted from this kind of approach. So what should we do? The easiest thing to do is use a tool that will generate a scheme of complementary colours for you. We'll investigate that later in this unit. But before we look at strategies for choosing colours and building a good colour scheme let's first consider some poor colour-schemes. Generally, the following rules of thumb will help you to avoid many problematic colour combinations.

- Green, white, yellow on green background
- Light objects on light background
- Bright colour combinations
- Bright/Textured backgrounds with coloured text
- Vibrant colours against black background
- Too many colours
- Predominant use of blue as background

Now let's look at some examples of each of these.

6.9.1 Green, white, yellow on green background

Certain foreground colours tend to be absorbed, to some degree, by green when it is used as a background colour. Similarities in hue can also lead to a muddiness instead of clarity between foreground and background.

Figure 6.3

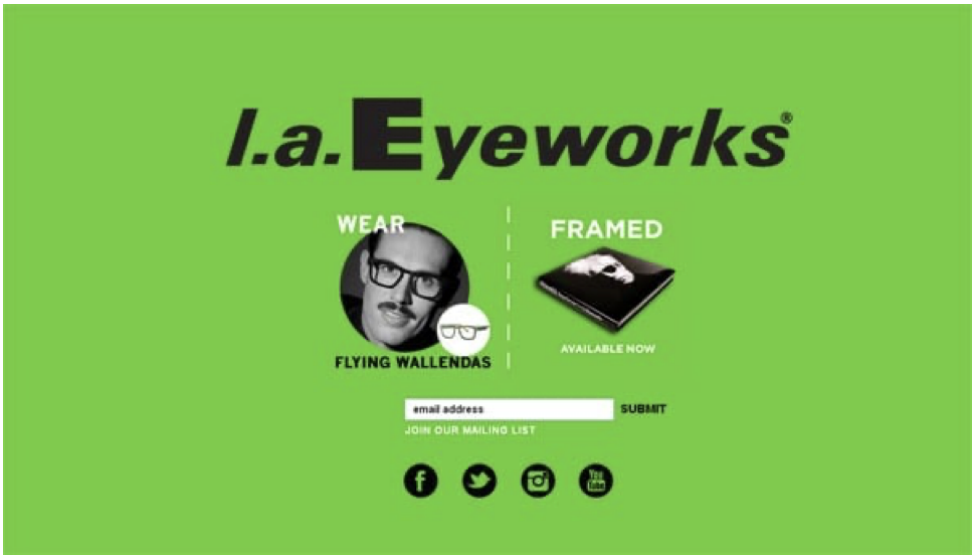


Figure 6.4

6.9.2 Light objects against a light background

To a similar degree this also applies to dark objects on a dark background. If there is insufficient contrast between foreground and background elements then your user can have difficulty in focusing and discerning points of interest. This can be compounded as the amount of material to read increases.

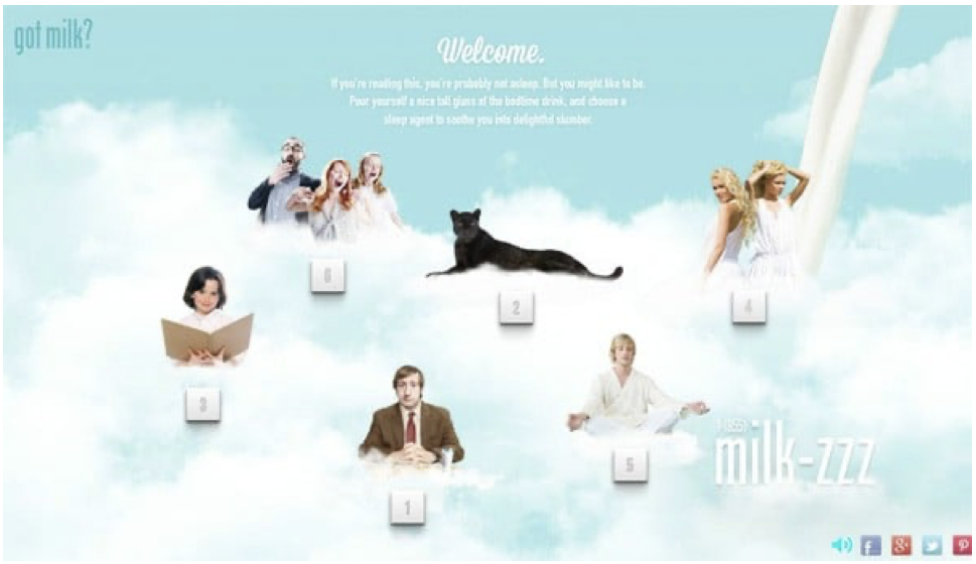


Figure 6.5

Figure 6.6

6.9.3 Bright colour combinations

These can be tiring to view and foreground elements, upon which we want our user to focus, can be lost amongst the rest of the page.

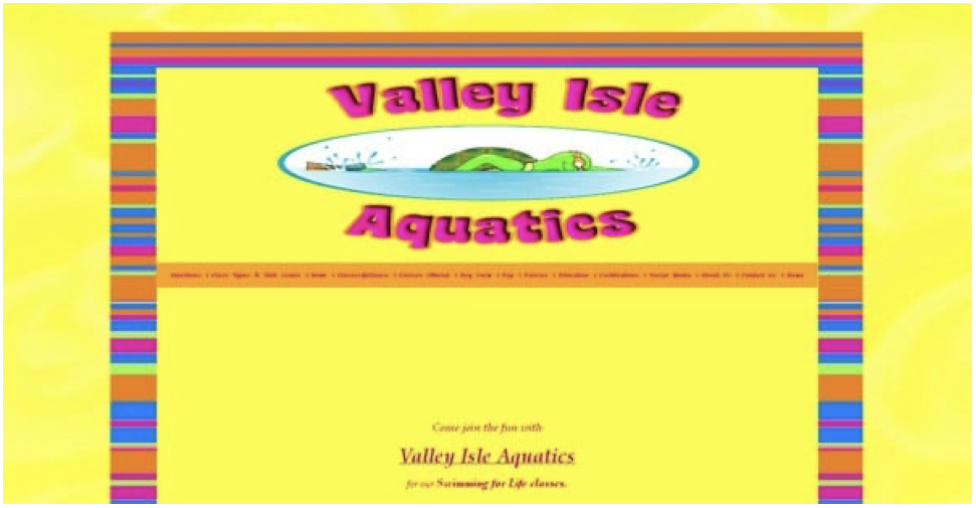


Figure 6.7

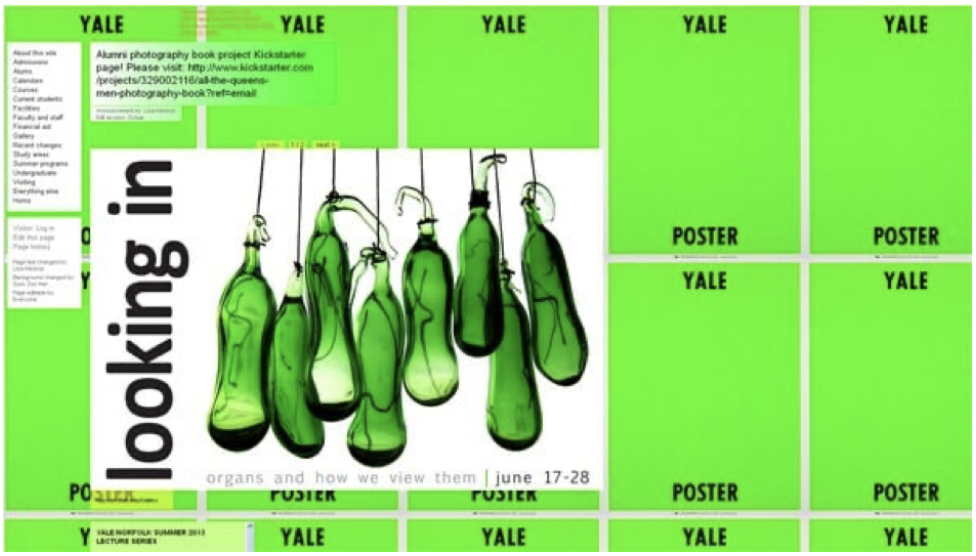


Figure 6.8

6.9.4 Bright/Textured backgrounds with coloured text

Effectively using a background that isn't a single, plain, neutral, colour is tricky to do well. The more bright and, or, textured the background, the more difficult it is to effectively add foreground elements that are easily legible. The eye is constantly drawn away from the foreground elements and the text can become lost.

Figure 6.9

Figure 6.10

6.9.5 Vibrant colours against black background

Black backgrounds always seem quite cooler, a reliable go to, but unfortunately they don't work so well in the context of web designs as they do when choosing clothing to satisfy your inner goth. However this kind of colour combination leads to low contrast between the foreground and background which can make text difficult to read.

Figure 6.11

Figure 6.12

6.9.6 Too many colours

If in doubt, throw everything against the wall and see what sticks. If choosing the right colours is difficult then why not just use them all? At least then you've got all the good combinations in there amongst the bad ones? It turns out that this approach doesn't work so well from a design perspective.

Figure 6.13

Figure 6.14

Note that this example has a number of issues, combining a complex background with the use of a lot of colours. I understand why it has been done, it makes sense to use the colours of the German flag as a starting point for the theme given the subject matter of the site, but this is hard to do effectively.

Figure 6.15

6.9.7 Predominant use of blue as background

As with black and green, most shades of blue can also make a poor choice of background. This is partly due to issues of insufficient contrast and partly due to the tendency of blue to dominate other colours. We don't want the background to dominate the foreground, hence we don't want a more dominant colour to invert the foreground/background relationship.

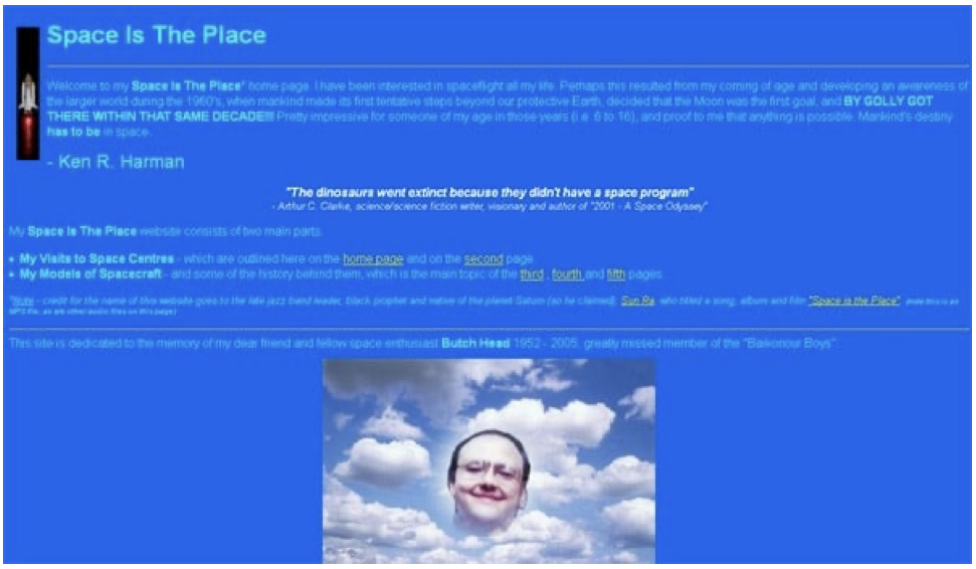


Figure 6.16

Figure 6.17

6.10 Choosing Colours

Start thinking about how many colours you need during your design process. This is not a straightforward process. It is not just a case of counting every element and then assigning a different colour for each one. Instead you must consider the relationships between the visual elements of your page. The most important relationship is foreground and background. You must have sufficient contrast between elements in the foreground and elements in the background. If there is too little contrast then your user will have trouble reading the text. However also bear in mind that too much contrast, such as black text on a white background, can be tiring if a lot of reading must be done from the screen. Whilst black on white is generally fine for printed materials, computer screens radiate light rather than reflecting it, and this can alter the visual perception as a result. So a good compromise is often the level of contrast that you might get with off-black text against an off-white background. However, as you've probably already noticed, most sites have a palette of colours with more than two elements for the foreground and background, so consider all of the other elements, perhaps headings, emphasised text, body text, hyperlinks, boxes, semantic blocks, &c. and decide whether you want them to share colours or have separate colours.

This process will give you an idea of how many colours you need in you palette but not concerned with what the colours are yet. As noted above, colours can be re-used in multiple places, so aim to minimise the number of colours rather than maximise. A five colour basic palette is very common even for complex sites. Many, if not most, colour schemes are in the range of between 5 and 9 elements with perhaps two elements, with fairly good contrast between them, being dominant in usage over the remainder. It can be worth trying out your colour scheme against your design document, if you've created one, as this can give a good way to rapidly see how all of the elements that make up your site will juxtapose against each other and also lets you easily apply the same colour scheme in different ways.

6.11 Palette Selection Tools

As many aspects of colour selection are fairly mechanical, a reliable solution, if you don't have an artist's eye, is to use a colour palette selection tool. These palette building tools are designed to use various aspects of colour theory and "complementary colours" to select the elements of your palette for you.

For example, you select a dominant or base colour for the scheme and the rest of the palette is built around that choice. The tool selects complementary, monochromatic, triad, analogous, compound, or shade based groups of colours for you from the colour wheel on the basis of colour theory. All you must do is decide how big your palette of colours needs to be, but you should have some idea of this already from your mockup activities.

Once you have your palette, which in practise might just be a list of colour hex numbers, you can then apply it to your design. You should aim to apply the colours from the tool consistently in implementing your design. Dont deviate and choose a different colour to replace an element of the palette as this can negatively affect all of the colours matching in the rest of the palette. Instead if you need to deviate from the colour palette you've built, then it means that there is probably another iteration of design required to fine tune the elements that make up the site.

Similarly, if you run out of colours then return to the palette tool and increase the size of the generated palette and reapply all of the adjusted colours to your design. This is why separating out HTML (markup/structure/representation) and CSS (presentation) is a good thing, altering the CSS alters everywhere it is used throughout your site.

This is a selection of common online palette generation tools:

- `https://coolors.co/`
- `http://www.colourlovers.com/palettes`
- `http://paletton.com/`
- `http://www.color-hex.com/color-palettes/`
- `https://color.adobe.com/create/color-wheel/`

There are others. Find one that you like and which you can use effectively. I've used coolers quite extensively in the past as a simple and easy way to get started and it looks like this:

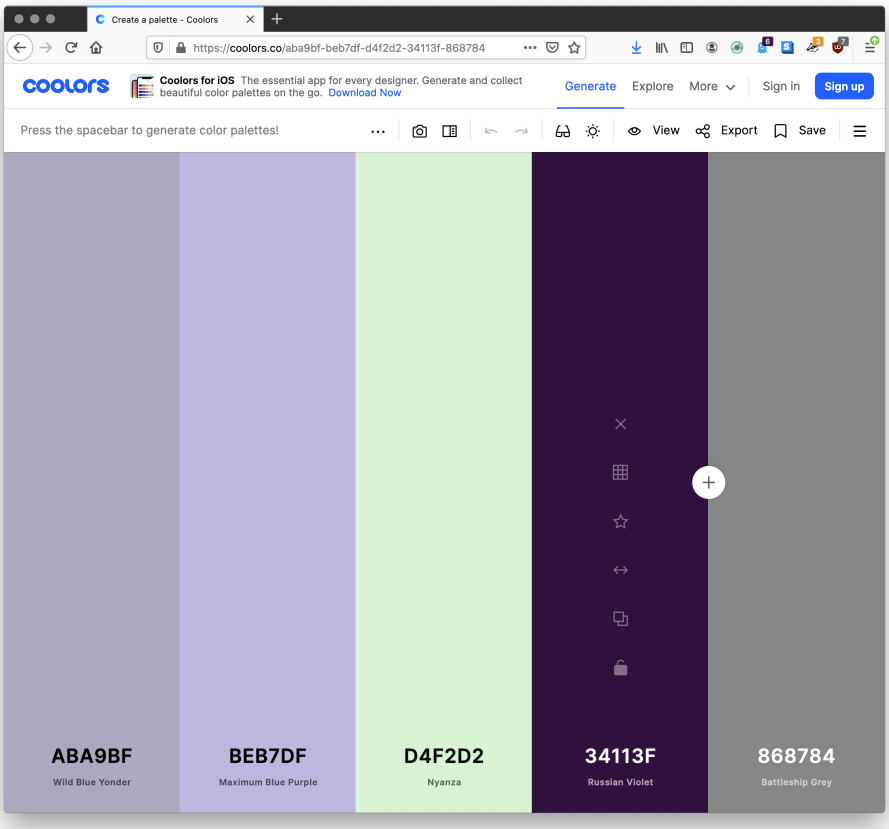


Figure 6.18

You can use the "+" icon when hovering over any colour to add additional colours, up to a maximum of 10 using the free version, and can return to any given colour scheme if you use the same URL, so bookmark your favourite schemes, or even better, note the address with its unique ID in your design documentation.

6.12 Design Guidelines

Once you've been through the design process, creating mockups, and making design decisions about such things as colour-schemes and visual presentation, you might ask yourself what the result should be. For some the set of mockups and designs is sufficient and you can move straight to implementation.

However if your site is going to have a longer life, or will be further developed by other people, or is meant to integrate into a wider organisation where there might be print, video, and other artefacts in addition to the website, for example, where there is a wider, perhaps corporate, brand identify beyond just the site, then it can be useful to document the designs and decisions somehow.

One approach to this documentation process is to develop design guidelines. These document design, interaction, and user experience decisions so that new additions and modifications are consistent. Design guidelines are a cohesive description of the look

and feel of your site that describes every element. This can be used to make sure you are consistent in applying your design and to make sure that others are consistent because they have a reference to which they can refer. Note that just a few out-of-design "additions" can turn a well designed site into something of a dogs dinner. By documenting your design you can extend the life of your project by helping to make it much more maintainable and enabling every design aspect to be referenced.

Design guidelines can be very extensive and complicated, they can become as complex as a language for detailing every aspect of your site, or be as simple as a single HTML page that demonstrates all the CSS and presentational aspects youve used. We'll look at both approaches now.

6.13 Design Documents

These are also known as "Pattern Portfolios" or "Style Guides". On the simplest level this can be a single deliverable, comprising a single HTML page which contains an example of every element, style, and component for your site. This way you can see how every aspect of your design looks in context with every other component all in one place. Developers can then use this as a reference when implementing individual pages within the site. As the design document is an HTML page and uses the live CSS for the site, then any changes to the design document are reflected throughout the implementation. In addition, because the HTML and CSS are both text, they can be committed to your source-code repository (perhaps in a folder named /design) and the version history can be tracked.

Note that design documents can be more complex than a single page, using as many pages as necessary to completely document the design. This should be driven by the need to balance keeping the document as small and easy to use as possible, whilst also adequately documenting the design.

Some people make their design documents public as part of their live site. Here are two examples of that

- https://paulrobertlloyd.com/styleguide/
- http://oli.jp/2011/style-guide/

6.13.1 Example Design Document

Here we have a visual example of a design document.

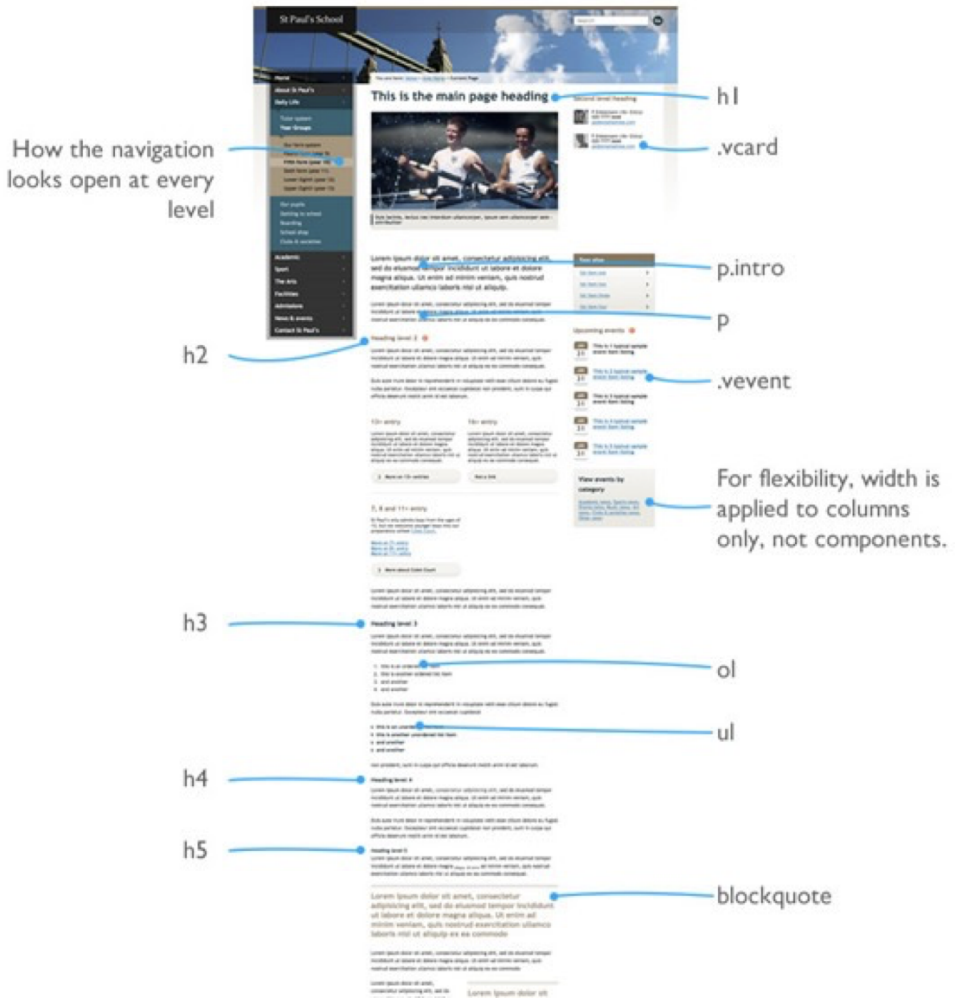


Figure 6.19

6.14 Design Guidelines In the Wild

Design guidelines beyond the design document are used by many organisations to design and document:

- Websites
- Mobile apps
- Paper comms (posters, flyers, leaflets) remember that paper is still very relevant for public communication

Remember that communications in larger organisations arent just for the Web but will often be multi-modal, hence the design guidelines might cover more than just the Web. The results can be large, diverse, and particular to the needs of the specific organisation. The best way to get to grips with these is to explore some examples.

6.14.1 TfL Design Standards

TfL (Transport for London) publishes a corporate design standard[2] to ensure that all communications maintain their design ideals

"This guide is designed to help us work as one brand, and to make it easier to produce high quality communications, experiences and propositions both internally and externally."

6.14.2 NHS Brand Guidelines

The NHS Brand Guidelines[3] have a slightly different focus (England Wide) and covering local GP practices as well as hospitals and 'corporate' communication. Aims to enable local doctors to develop materials, e.g. information leaflets, that are in keeping with wider NHS corporate branding standards

There is an additional element of trust here. There is lots of health communication that is quackery at best and outright dangerous at worst. Important to establish provenance.

6.14.3 BBC GEL

A GEL is a Global Experience Language. Global refers to the whole experience and the reach of the resulting artefact. The BBC use their own GEL[4] to ensure that all sites under the BBC brand are coherent

6.14.4 Edinburgh University GEL

Edinburgh university developed its own GEL[5] to support best practices in the promotion of their University brand.

6.15 Some Design Hacks for Coders

The following is an annotated list of rough design hacks and rules of thumb that I've gathered and generally adhered to over the years. They aren't hard and fast rules, but are instead a useful starting place. Apply them as works for the context in which you're working.

- Use palette selection tools to generate a decent set of colours then use those colours carefully and deliberately by applying them to your design. This presupposes that you have an idea for a design.
- Sketch out or else mock-up a design. This will likely be a living document and change greatly, but enables you to start working out how your site content will come together. Even if you don't have any content yet, you will probably have some idea of the kind of content that you will have in the future.

[2] `https://tfl.gov.uk/info-for/suppliers-and-contractors/design-standards`
[3] `https://www.england.nhs.uk/nhsidentity/`
[4] `http://www.bbc.co.uk/gel`
[5] `http://gel.ed.ac.uk/`

- Mock-up your layout using greeked text if you don't yet have any written content. In the meantime the greeked mock-up will enable you to still make some progress.

- If you do have content but you don't have any ideas for a design then remember that form can follow content. So why not try starting with an HTML only version of your site and content. This will give you a good skeleton to work with as your design matures. By doing this you might better understand the structure of your content which might make the design aspect easier.

- Don't try to fill the page. Your content will develop over time. Your pages might look fairly empty initially. That is fine.

- Make good use of whitespace. This is a corollary to not filling the page. Any content you do have should not be too cramped. Try to make good use of the CSS box model and make sure that the elements of your page have room to breathe, for example, using margins effectively around text can make for a much better reading experience. Think about when you create a written document, you leave white space all around the edge, even though it may seem wasteful, mainly because this makes it easier to read.

- Dont use too much whitespace. Another corollary to not filling the page and making good use of whitespace is to not go overboard. We still want good information density in our interface so that our user doesn't have to scroll too much. Our eyes will often skip around the page when reading and take in much more than we expect. The more that we have on screen, the more we can take advantage of this. This also means thinking about your content, if there is data that users might want to compare, then designing so that all of that information is close together and can be seen together on screen is not a bad idea.

- Use less furniture (rules, boxes, dots, outlines) around your page elements. Sometimes we are tempted to draw a box around things to set them off against other elements, or to delineate boundaries, but this is more visual clutter and can detract from the clarity of your message. It can be worth using less visible furniture sometimes to delineate your data, or else making appropriate use of whitespace to isolate individual elements from each other.

- Spacing and alignment are hugely important especially when comparing data. For example, when skimming over data it is easier when things are aligned vertically, particularly for numeric data but also for regular reading of text. As a result you should rarely need to center text other than perhaps in headlines or to isolate something special, but most prose should be left justified to enable your reader to skim over your paragraphs, allowing their eye to jump easily down the page. As normal text fonts have variable sizes between sigils it is worth selecting a monospaced font if you need to align digits in numeric data but bear in mind that such fonts are problematic for regular reading of non-numeric data. Similarly, if you are dealing with currency, aligning decimal points can help your reader. Narrow line-heights between rows of data can keep data close together. But note that your data should still be cleanly separated. This might involve some trial and error, but the goal is to get a good density of information viewable at once without cluttering things and reducing comprehension.

- You can also attempt to reduce the volume of data when it is presented visually. The visual presentation of your data can be different to the internal, stored representation. This process of reducing volume should be governed by what functionality you are trying to provide. For example, why have both first and last names in separate columns when they can be combined into a single name when displayed? Similarly, why have separate columns or rows for each of Town, County, and Postcode when they could be joined and displayed as a single "address" datapoint. Note though, that if the user has to interact with these things as separate data then you might opt not to combine them. As a result, this is driven by the needs of the site, the users, the data, and the design. It's worth removing redundant data and anything else that the user doesnt need to see from your page. As a rule, it is better for privacy preservation, and any associated security aspects if you consider that you can't lose or accidentally disclose user data if you don't possess it. So, generally, don't collect data that you know, or are pretty sure, that you don't need.

- Get feedback on your designs. Ask potential users and show them what you're doing and solicit their feedback. Note though that they can be wrong, they can have an idea of what they think they want and can be vocal about it, until you show them your novel new design that they hadn't ever considered. If you do engage with your users then watch them perform their tasks and use that as inspiration for your design, but dont slavishly reimplement real-world procedures if there is a better way. Remember that users who are doing things in the real world might have some great ideas about how to streamline the process.

- Perhaps most important, remember that youll never make the perfect UI for all people, so why not let your user export the data from your site using a common format, for example, using a Comma Separated Values (CSV) file, JSON, or XML. Your user can then use an appropriate tool to manipulate the data in the ways that they need to. For example, tabular data, that is, data arranged in row and columns, can be easily exported as a CSV file and then loaded into any spreadsheet applications or manipulated using a scripting language. This is much more flexible than trying to account for all the ways that a user might need to use data within your web interface.

- If in doubt, and all else fails, use an existing CSS design solution, such as bootstrap, as a shortcut to a pleasing prototype.

6.16 Summary

In this chapter we've considered the importance of design with respect to some real world problems and have considered the potential consequences of poor design. We've also explored some aspects of prototyping our sites, mocking up content, and selecting colour. We've then finished by considering some "Design Hacks for Coders", a collection of heuristics for guiding you towards creating better designs.

Chapter 7

Core JS

Javascript is the final distinct language that we'll study and makes up the third part of our triumvirate of core Web technologies alongside HTML and CSS.

Assuming that HTML enables us to define information, and CSS enables us to style that information visually, it is JavaScript that enables us to dynamically interact with our information, to manipulate our pages, or even to create and alter them on the fly (as well as to do other cool stuff like implement games, graphics, sound, etc.). Javascript is a full blown programming language that just happens to live, mostly, in the web browser. In this unit we'll introduce ourselves to the core of the language.

7.1 JavaScript

JavaScript (JS) is the third part of our triumvirate of web technologies alongside HTML and CSS. Just as HTML and CSS enable us to delineate content from presentation, JS enables us to further separate out functionality and behaviour from a Web pages content and presentation.

So what is JavaScript? It is a general purpose programming language, more specifically a high-level, dynamic, weakly typed, prototype-based, multi-paradigm, interpreted programming language.

For our purposes in WebTech, focussing on the client-side user interface, it is a browser hosted programming language. This means that JavaScript code runs inside the browser. JavaScript was originally a project at Netscape which lead to the first Javascript implementation. This was then formalised to produce the ECMAScript specification. Whilst the actual name "Javascript" is a trademark of Oracle, anyone can implement the specification to produce their own version of the language. As a result there is no single Javascript implementation, Instead there are a range of different engines in most web browsers. Each implements the ECMAScript specification to varying degrees so some features might be missing and some engines have additional features. This means that Web developed using JavaScript can be complex.

7.2 JS The Language

JS is multi-paradigm – This means that it can support a range of different programming styles, for example, event-driven, functional, imperative, and Object Oriented. This means that it is very flexible and adaptable to fit a wide range of problems and enables you to approach a solution driven by the needs of the problem domain rather than the needs of the language.

In addition to the core language, which we'll investigate very soon, JS provides an API for common tasks such as working with text, arrays of dates, dates, regular expressions, as well as Web specific APIs for doing DOM manipulation.

Note that there is no core language support for direct input/output (I/O) like we'd expect in most languages. This is because the host environment, the browser provides the "place" where your code runs, so the host environment provides the ability to do input and output. Input data usually comes from the browser, usually related to the web page that is loaded at any given time. Output is also handled by the browser, usually either through updates or manipulations to the currently loaded web page, calls to remote APIs over the network, or through browser logging mechanisms.

In most languages we'd expect I/O as well as other facilities like networking, storage, and graphics to be part of the standard library of facilities. In JS these are provided either by the host environment, i.e. the browser API or through additional libraries and APIs.

It's worth considering JS as a kind of "glue language". It is meant to be easy for everyone from web designers and part-time programmers as well as hobbyists to assemble the components of web pages to create their sites.

7.3 An Aside: Java(Script)?

If you're confused and are wondering about the naming of Javascript and Java, that's entirely understandable. These are two wholly separate languages so you shouldn't confuse one for the other. Whilst they might share some syntax or have similar libraries, and overlapping names, these are merely superficial similarities.

Brendan Eich was originally employed by Netscape to write a version of the Lisp variant Scheme to run in the browser. This was originally meant to fulfil the requirements that JS now fulfils. However the Netscape management made the decision that Javascript should look like Java which was the hot new thing in the mid to late 1990s. So a Java-like, or Java-inspired language, called JavaScript, was developed and released in Netscape's browser. The rest is, as they say, history.

7.4 JS Engines

We've already established that JS is interpreted, Virtual Machine (VM) based, and runs within a host environment. In this context interpreted means that JS source-code

is read, line by line, and the computer decides, on-the-fly, what to do as a result. This should be contrasted with languages like C in which the source-code is compiled into a different form before execution. VM based means that rather than running directly on the host machine, JS is run within a special execution environment which does the job of interpreting the source-code and working out what to do, then doing it. All languages run within a host environment but usually this just refers to the hardware platform and the operating system. However, in the case of JS running on the Web, the host environment is the web browser or whichever other, JS enables user agent is executing the JS.

Because browsers do more than merely execute JS, the part of the browser, the JS VM, that interprets and executes JS source-code is often called the JS engine. There are many JS engines, and as we pointed out earlier, you can implement your own from the JS specification, but some implementations are more important than others usually because they are widely used or have particularly good performance characteristics. These engines include:

V8 The Google Chrome JS engine. Note that V8 is also used in Node.js, another technology that enables JS code to be run outside of the browser host environment.

Spidermonkey The Firefox JS engine.

JavaScriptCore This is marketed as "Nitro" and is the JS engine used in Safari on MacOS platforms.

You can work with JS in the browser fairly easily as you would expect. But you can also install a standalone tool to run JS outside the browser (e.g. Node.js) this enables Javascript to run server side. For some people, using a similar implementation language for code that runs in the browser and code that runs on the server, for server-side dynamic sites, is an attractive prospect. Note though that whilst there are limited options for languages that run directly in the browser, most languages can be used to create server-side functionality.

7.5 What is JavaScript for?

JS is a, slightly weird, general purpose programming language. This means that we aren't limited to just web-oriented programming tasks. Basically any programming task that you could want to solve using any other language, can also be solved in JS.

However Web programming is the *raison d'être* for JS. This means, primarily, interacting with the DOM for a given web page and also interacting with the user agent. By interacting with the DOM we mean that we can use JS to alter or replace any HTML element for the page loaded in the current browser tab or even to completely discard the current content and replace it with a new page entirely generated from JS. There are lots of API functions in JS that support manipulating the DOM in a variety of ways. By interacting with the user agent, we mean that we can use JS to manipulate the host environment provided by the browser. Most browsers provide a set of APIs or Application Programming Interfaces that support audio, graphics, client-side data storage and a wealth of other functionality.

We said primarily above. A secondary use of JS is to provide a server-side environment so that our core tools can be (reasonably) cohesive whether they are operating on the server-side or the client-side. Finally, a tertiary use of JS is as a general purpose scripting language where any other scripting language might be used but this is a much more minority pursuit.

7.6 What Does JS look like?

On its own JS looks like this:

```
1  console.log("hello world");
```

You can try that out by typing it into the JS console in your browser. This just executes any JS that you type in immediately in the style of a Read-Evaluate-Print-Loop (REPL) and is a useful way to play around with JS as you are learning the language.

Realistically though you are more likely to see JS in the context of a web page. Here is one example:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4          <title>Example</title>
5      </head>
6      <body>
7          <button id="hellobutton">Hello</button>
8          <script>
9          document.getElementById('hellobutton').onclick = function() {
10             alert('Hello world!');
11             var myTextNode = document.createTextNode('Some new words.');
12             document.body.appendChild(myTextNode);
13         };
14         </script>
15     </body>
16 </html>
```

In this case the JS is hosted within our HTML document between ¡script¿ tags. Usually we would actually store our JS outside the HTML and reference the JS source file in the same way that we stores CSS in external files. But for our first exposure to JS it is easier to use a single mixed example. You can type the code above into an html file, save it as hellojs.html and then open it in your browser. Note that in the document we are creating an HTML button element and giving it an ID. We are then creating some JS that provides a function that runs when the button is clicked. We use the buttons ID to identify it from other HTML elements, although there are no others in this example. We use a DOM API function to access the button element by supplying the ID to the document.getElementById function. Once we have a reference to the button element we are then attaching a function to it which first creates a popup alert box saying "Hello world!" and then we're creating a new element, a text node, setting the content of the text node to "Some new words." and then appending the new text node to the HTML document displayed in the browser so that the content of the page is changed by our JS.

7.7 Using Javascript

Similar to CSS, and as noted above, we have three basic choices for how to integrate our HTML with our JS. These options are inline, in a script block, and in an external script file.

Inline looks like this:

```
<button id="hellobutton" onclick="alert('Hello World')">Click Me</button
    >
```

Basically we are directly inserting the JS into the element to which we want it to apply. Useful in some contexts but having all of the same drawbacks that we explored earlier when applying style directly inline onto HTML elements.

We can also use JS within a script block, e.g. a pair of ¡script¿¡/script¿ in our HTML page.

```
<script>
    document.getElementById('hellobutton').onclick = function() {
        alert('Hello world');
    };
</script>
```

We can also use an external script. In this case we need to tell the HTML document where to find any associated JS. We do that by placing a ¡script¿ tag into either the ¡head¿ or the ¡body¿ of the HTML file. There are some differences in how this is done between versions of HTML though.

HTML 4

```
  <script type="text/javascript" src= javascript.js"></script>
```

HTML 5

```
  <script src="javascript.js"></script>
```

Note that practically speaking you can use any number of external scripts and reference them from your HTML page. This means that you have significant flexibility in organising your JS code into separate files.

7.8 JS in the Browser Console

As we saw earlier, we can execute JS in the browser's console. This is useful for learning JS and experimenting without the hassle of setting up an HTML file to host it. However well quickly grow beyond this approach. Note though that the browser console does give us a small programming environment almost anywhere where there is a reasonably modern web browser. This is very useful to be aware of and we'll make the most of it for the next few examples.

Let's look at some simple examples of how we can write JS directly in the browser console. I encourage you to try them out yourself, and then to use the functionality in each as a starting place for exploring the JS language API and for your own learning experiments.

7.9 Hello Napier

This is basically the example we saw a few moments ago, but now with added screenshot goodness. We've included it as some form of "Hello World" program as is the traditional starting place for most new programming exercises.

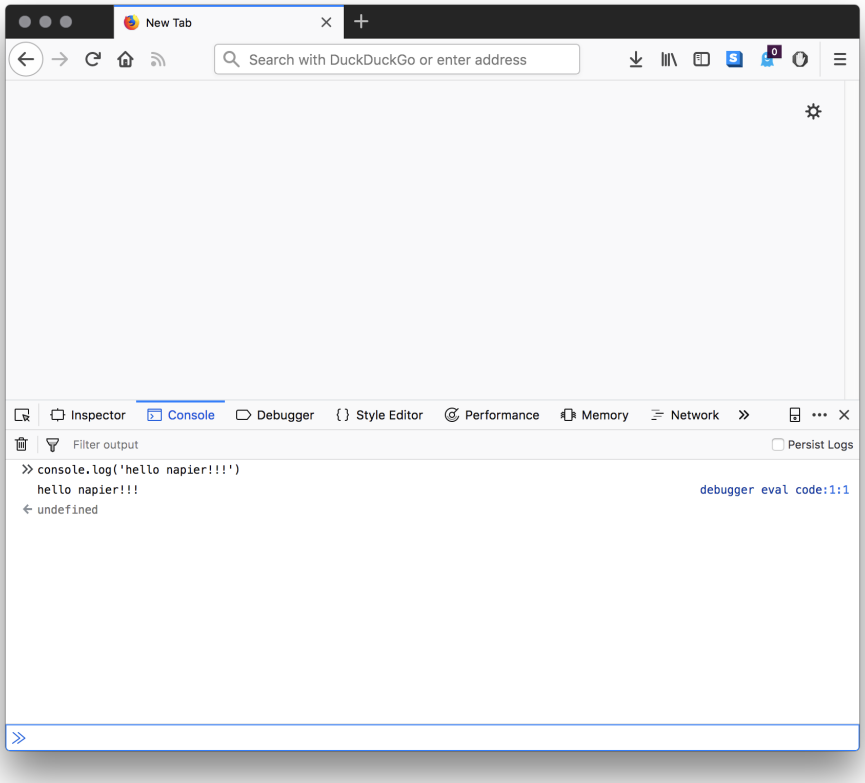


Figure 7.1

7.10 Interact with the Web Page/Screen

In this example we use the browser DOM manipulation API to access the background colour setting of the document body so that we can alter it to a different colour. This basic approach works for many of the style elements provided by CSS to enable us to programmatically manipulate a page's style.

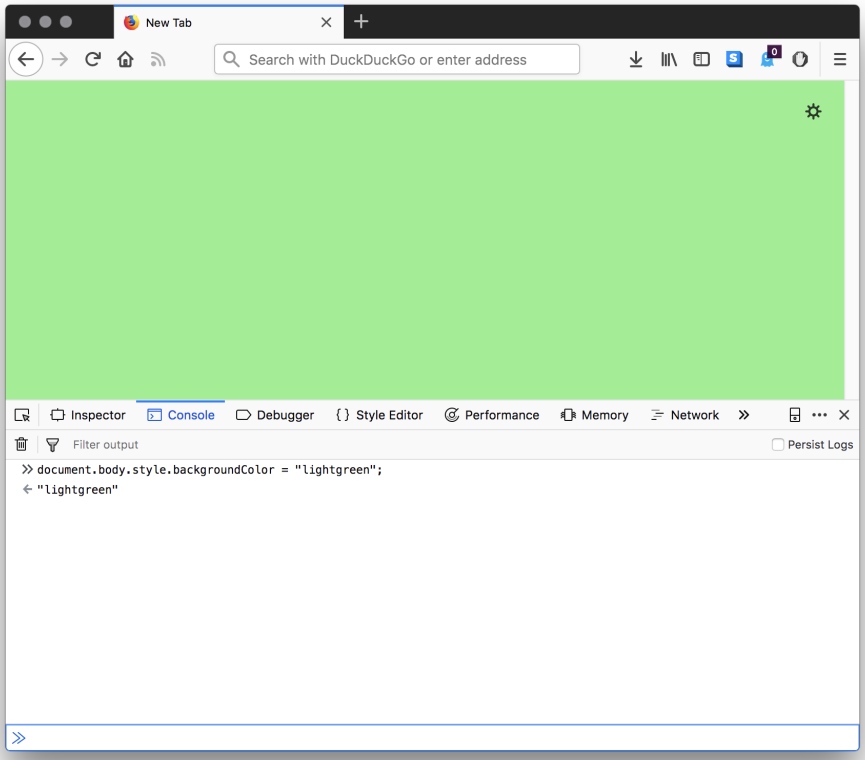


Figure 7.2

7.11 Use standard JavaScript functions

We can use JS language functions to provide information which can then be displayed on the current web page. In this example we create a new variable, called 'd' and then assign it the value returned by the built-in Date function. We then replace the HTML content of the current document displayed in the web browser to include the value stored in our variable.

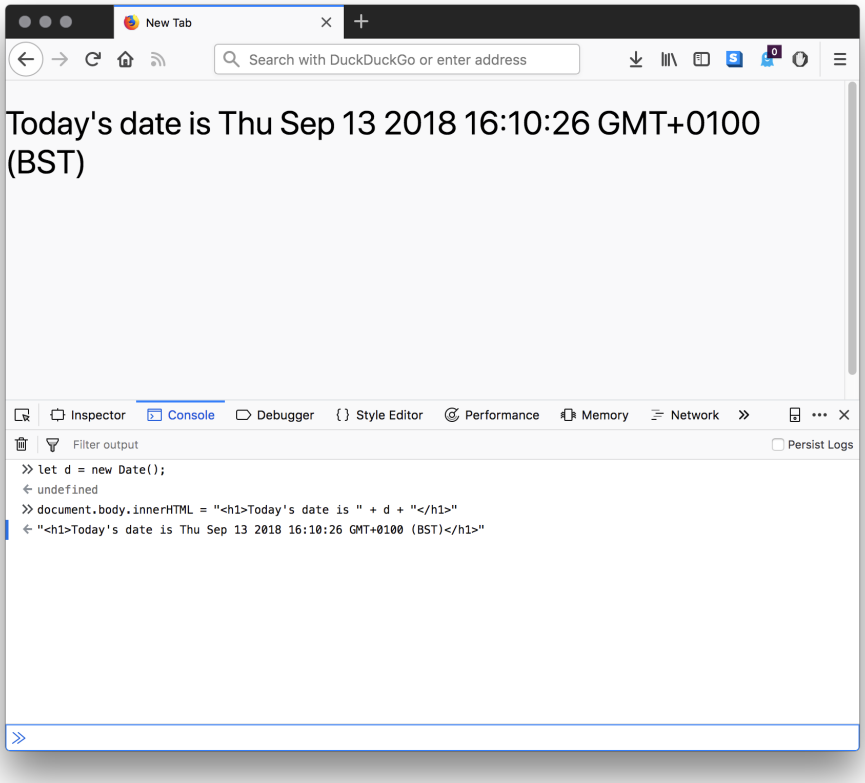


Figure 7.3

7.12 Construct A web Page

In this example we are going to construct a simple web page entirely from scratch using JS. First some variable, ‘p; to store a paragraph element, and ‘t’ to store a text node that contains the content for the paragraph element. We then append our text, t, as a child of the paragraph, p and then replace the body of the document, the existing content, with our new paragraph node and its encapsulated text node child.

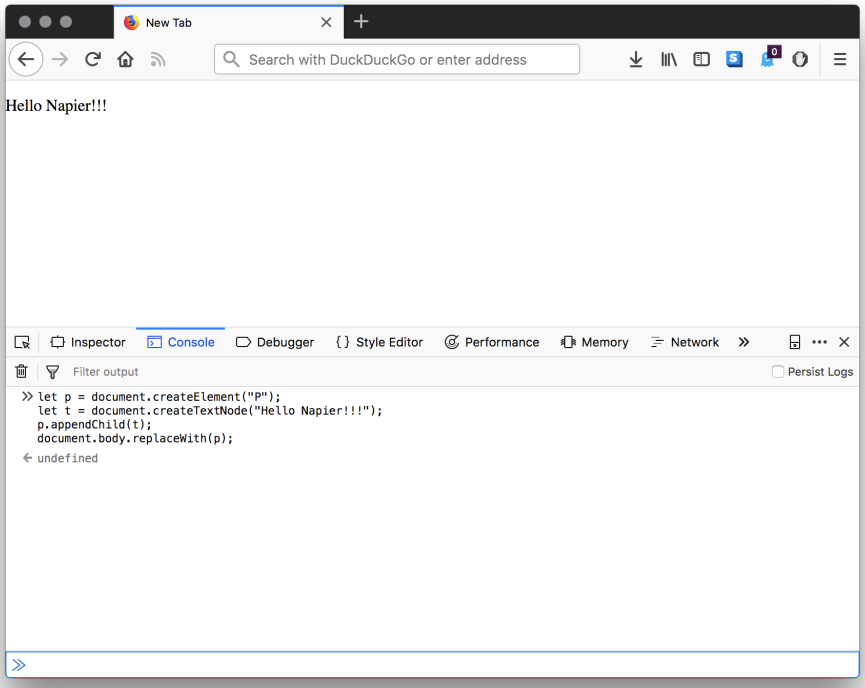


Figure 7.4

7.13 Graphics

This example is a very simple usage of the graphics API to draw a circle on a canvas element. The HTML canvas element is an HTML element that is designed to hold 2D and 3D graphics, as opposed to text which many HTML elements hold.

We create a few variables in this example, the first being the canvas element, c, and the second being a variable for the drawing context which we set to 2D and then subsequently interact with to create our actual graphics. On our canvas we want to draw a circle. If we were using a pencil in the real world, then a circle would be a single pencil line. In the canvas context, this line is called a path, and we tell the canvas drawing context that we want a path to be drawn using an arc. Bear in mind that a circle is an arc that begins and ends in the same place and follows a constant angle. We specify how the stroke of the path should be drawn, because we don't actually have a pencil here so we need to say what we want the path to look like. We can alter the style of the stroke, but for now we'll accept the default. Finally we replace the existing document body with our new canvas which causes the page to be redrawn with our new canvas as the sole content containing our new circle.

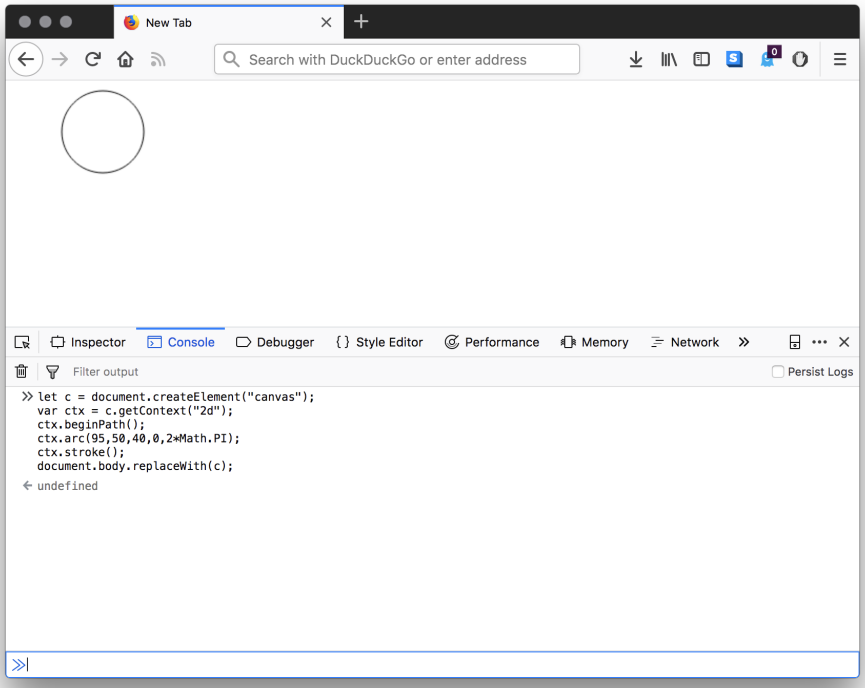


Figure 7.5

7.14 Sounds: Beeps

As well as graphics we can work with audio. In this example we create an audio context then build our sound from scratch. Sound is characterised by a waveform of a given frequency and that waveform is created by an oscillator. So we create our context variable and use it to retrieve a handle to the browser window's audio context. We then create a variable to hold our oscillator and set the waveform type, or shape, to "sine" for a sine wave and the frequency to 440Hz. The oscillator is then connected to the audio context and set to start which causes it to play and, as a result, make a noise.

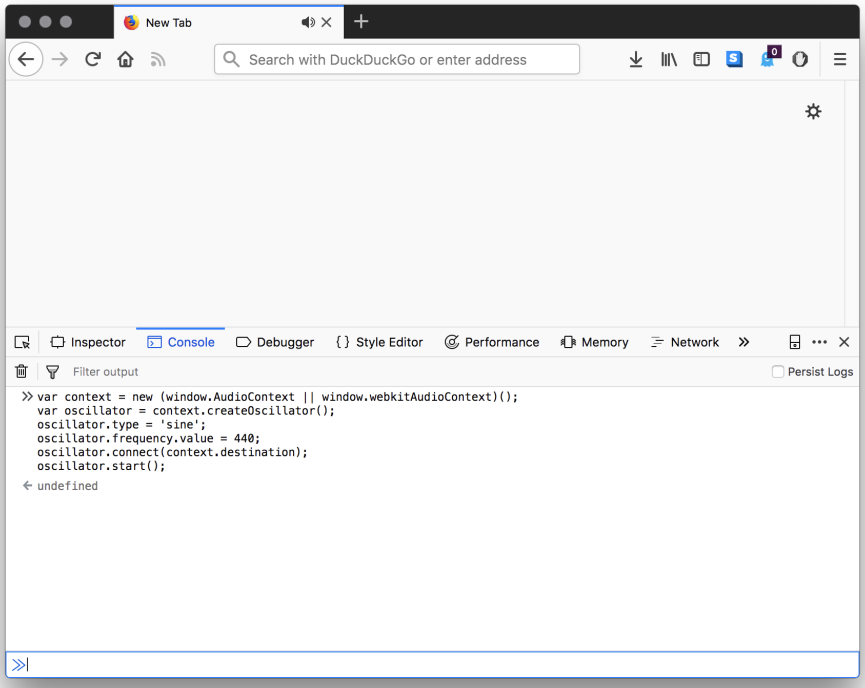


Figure 7.6

7.15 Sound: ChipTunes

We can go further than mere beeps though. So in this example we create a slightly more complex example that shows a way to play a variety of different sounds for different lengths of time. If we play different sounds in sequence, then we have a rudimentary tune. This is how we can build music from first principles.

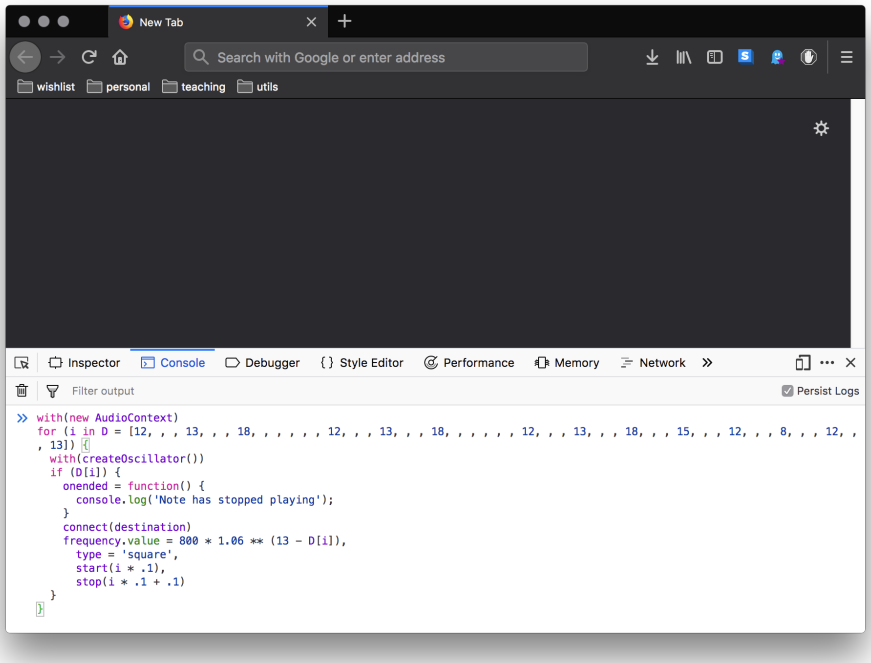


Figure 7.7

7.16 Sound: Theremin

In this final example we create an audio context and oscillator as before. However this time we've also added an event listener to the document which accesses the mouse pointer location. This is connected to the oscillator and we use the location of the mouse pointer to alter the oscillators frequency. In essence, we've created a version of an esoteric musical instrument called a Theremin. This shows one way that we can use user interaction to affect the audio that is played by the browser.

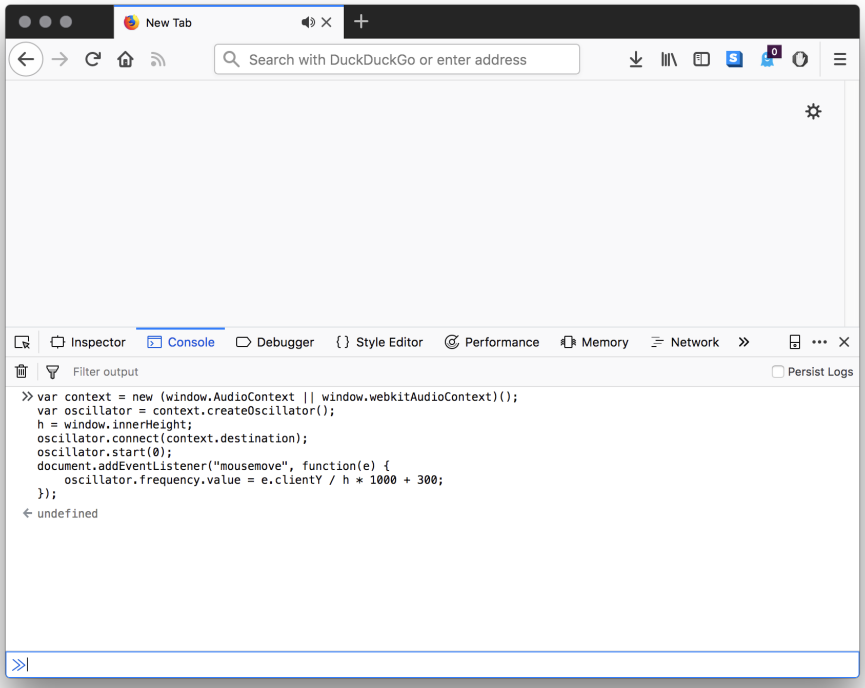


Figure 7.8

7.17 JS as a Language

Having tried out some JS and seen a little of what JS can do in the browser, we can, with the benefit of hindsight and some experience, now look at the language in a little more detail. Remember though that this overview of the language assumes that you can already program in some language. It is not aimed at teaching you to program but at helping you to add JS as a new language to your toolbox which should already contain at least one other language. If you already know some C-syntax style language like C, C++, C#, Java, Python, etc. then you will find that there is quite a bit of syntactic overlap. Obviously the detail can be very different, but the basic programming approach is similar and generally knowledge from one of these languages can be quite applicable to JS.

IMPORTANT: If you want to become a good programmer in any language then you need to practise. Reading code or documentation is much less useful than actually writing code and solving problems.

Let's look at JavaScript as a standalone language. By this we mean that we want to consider such things as:

- keywords, commenting, variables, primitive datatypes, native objects, and operators
- Examine control structures (if-else, conditional, switch) and looping
- Functions, Objects, and Exception handling
- JavaScript object serialisation using JSON

7.18 Case, Whitespace, and Statement Separators

JS is case sensitive. As with all programming we have to be precise so you can't alternate between upper and lower case as they don't always mean the same thing. It is best to develop a habit for using case then stick to it. I nearly always use lowercase (and also choose snake_case rather than camelCase when naming things).

Spaces, tabs, and newlines outside of strings are whitespaces but JS is not whitespace sensitive, unlike, say, Python. This is generally not a problem so you should choose a way to use whitespace then be consistent in laying out your code.

Semicolons are used to separate statements. However, like many web technologies, you can leave them out of your code and the JS engine will automatically insert them for you. However, in certain circumstances code can be ambiguous and the engine can make an incorrect choice of where to insert a semicolon. This can interfere with whitespace handling and cause valid code to be incorrectly parsed resulting in an error or incorrect functionality. As a result it is generally good practice to end statements with a semicolon.

7.19 JS Keywords

JS has a number of keywords that can be organised into four groups according to whether they are core JS keywords, built-in keywords, DOM and Window object related keywords, and event related keywords. You should avoid any of the following when naming your own variables, function names, or labels but obviously use them when you want to use these keywords as intended by the JS language.

JavaScript keywords abstract, arguments, await*, boolean, break, byte, case, catch, char, class*, const, continue, debugger, default, delete, do, double, else, enum*, eval, export*, extends*, false, final, finally, float, for, function, goto, if, implements, import*, in, instanceof, int, interface, let*, long, native, new, null, package, private, protected, public, return, short, static, super*, switch, synchronized, this, throw, throws, transient, true, try, typeof, var, void, volatile, while, with, yield

Built-in Objects, Properties, & Methods Array, Date, eval, function, hasOwnProperty, Infinity, isFinite, isNaN, isPrototypeOf, length, Math, NaN, name, Number, Object, prototype, String, toString, undefined, valueOf

HTML & Window objects & properties alert, all, anchor, anchors, area, assign, blur, button, checkbox, clearInterval, clearTimeout, clientInformation, close, closed, confirm, constructor, crypto, decodeURI, decodeURIComponent, defaultStatus, document, element, elements, embed, embeds, encodeURI, encodeURIComponent, escape, event, fileUpload, focus, form, forms, frame, innerHeight, innerWidth, layer, layers, link, location, mimeTypes, navigate, navigator, frames, frameRate, hidden, history, image, images, offscreenBuffering, open, opener, option, outerHeight, outerWidth, packages, pageXOffset, pageYOffset, parent, parseFloat, parseInt, password, pkcs11, plugin, prompt, propertyIsEnum, radio, reset, screenX, screenY, scroll, secure, select, self, setInterval, setTimeout, status, submit, taint, text, textarea, top, unescape, untaint, window

HTML Event Handlers onblur, onclick, onerror, onfocus, onkeydown, onkeypress, onkeyup, onmouseover, onload, onmouseup, onmousedown, onsubmit

7.20 Comments

There are two basic ways to insert your own comments into your JS. Like so:

```
1 // A short, one line comment
```

or like so:

```
1 /* a longer, multi-line
2 comment about something
3 that must be important */
```

Note that:

```
1 /* Comments /* cannot be nested */ as this is a syntax error */
```

7.21 Variables

Variables can be declared in three ways:

- Block level variables using the let keyword
- Function level variables using the var or const keywords

Variables in JS dont have a type attached. This means that any value can be stored in any variable. This is different to other languages, like C for instance, which require a type to be specified for every variable.

A variables value is undefined until it is initialised, e.g.

```
1   var a = 203
```

A variable declared anywhere inside a function will resolve to that variable when its name is used within the function but variables declared outside a function are global. If you try to use a variable but it can't be found by the JS engine then a ReferenceError exception will happen.

7.22 Primitive Datatypes

Although we don't have to declare a variables datatype, that doesn't mean that they don't exist in JS. There are a number of primitive datatypes available, these include Undefined, Null, Number, String, Boolean, and Symbol.

Let's consider each in turn:

Undefined This is assigned to all uninitialised variables and returned where checking for object properties that dont exist.

Null This is used when a variable has been declared but the value is empty.

Numbers These are IEEE754 standard doubles. They are floating point numbers which can be rounded to whole numbers using the toFixed() function. The accuracy of primitive JS numbers is 16 significant digits.

Strings These are handled as a sequence of characters enclosed in double quotes. Individual letters in a string can be accessed using the charAt() function. Strings can be compared using '=='.

Boolean Standard boolean binary values of true and false are available to be used for comparisons.

Symbol This is new to JS as of the ECMAScript 6 specification and are used to create a unique and immutable identifier.

7.23 Native Objects

JS includes a range of useful native objects including Arrays, Dates, Errors, Math, Regular Expressions, and Functions.

Let's briefly consider each in turn:

Arrays These are objects that represent lists of values. Arrays are indexed by keys which are numeric and use zero based indexing. Arrays can be multi-dimensional so they can be used for matrices of various dimensions.

Date This is an object that stores a signed millisecond count from zero. Zero represents the following date: 1970-01-01 00:00:00 UT and everything else is an offset from that zero point. The date object supports various methods to access fields of the date object.

Error An object that can be used to create custom error messages.

Math An object that stores various mathematics related constants such as values for E, the natural Log, Pi, &c. It also support various functions like max(), min(), and random() amongst others.

Regular Expressions An object that is used to store text patterns and extremely useful when needing to find instances of some search pattern within another body of text.

Function An object that is constructed using the Function constructor and used to collect statements into reusable groups. An awful lot of JS programming comprises writing functions.

7.24 Operators

There is a full complement of the standard operators that you should recognise from other programming languages. These include the following:

Arithmetic +, -, *, /, %

Unary + (string to number), - (reverse sign), ++,

Assignment =, +=, -+, *=, /=, %=

Destructuring assignment [a, b, c] = [3, 4, 5];

Logical !, ||, &&

String =, +, +=

NB. There are also bitwise operators and assignments as well as a ternary conditional.

7.25 Control Structures

We need control structure to enable us to affect the flow of our programs, letting us check conditions and branch to different paths of execution. The standard control structures include the if/else structure, the conditional operator, and the switch statement. Again, these should all be familiar from languages that you've already experienced and all work in a fairly similar way.

7.25.1 if/else

```
if (expr) {
  //statements;
} else if (expr2) {
    //statements;
} else {
    //statements;
}
```

7.25.2 Conditional Operator

```
result = condition ? expression : alternative
```

7.25.3 Switch Statement

```
switch (expr) {
    case SOMEVALUE:
      // statements;
    break;
    case ANOTHERVALUE:
      // statements;
      break;
```

```
    default:
      // statements;
    break;
 }
```

7.26 Looping

When we need to do something more than once, a standard strategy is to loop over the statements however many times are required. JS supports the standard approaches to looping that we should recognise from other C style languages and syntaxes. These include:

7.26.1 For Loop

```
for (initial; condition; loop statement) {
    /*
    statements will be executed every time the for{} loop cycles, while
       the
    condition is satisfied
    */
 }
```

7.26.2 For In Loop

```
for (var property_name in some_object) {
    // statements using some_object[property_name];
}
```

7.26.3 While Loop

```
while (condition) {
    statement1;
    statement2;
    statement3;
    ...
}
```

7.26.4 Do/While Loop

```
do {
    statement1;
    statement2;
    statement3;
    ...
} while (condition);
```

7.27 Functions

Functions are at the core of JS programming. There are various ways to declare (create) functions. For example:

Using the function keyword to define a function:

```
function add(x,y) { return x + y; };
```

Using the function keyword to define a function and assigning it to a variable:

```
var add = function(x,y) { return x + y; };
```

Creating a new function and assigning it to a variable using the Function() function

```
var add = new Function( x , y , return x + y );
```

7.28 Objects

JS supports object orientation using a form of Object Oriented Programming (OOP) that uses prototype-based inheritance. We can create objects and assign data and method to them in at least four different ways but this is mostly syntactic sugar. The following is probably the easiest way to create an object in JS, by just using an object literal. An object literal is an instance of the JS global Object() object type. Let's see it in action:

```
var cow = {
    colour:'brown',
    commonQuestion:'What now?',
    moo:function(){ console.log('moo'); },
    feet:4,
    accordingToLarson:'will take over the world'
};
```

This gives you an object called cow that you can interact with, e.g. cow.moo(). Another way to do this is to create a new object then set its parameters, like so:

```
var cow = new Object();
cow.colour = 'brown';
cow.commonQuestion = 'What now?';
cow.moo = function(){ console.log('moo'); }
cow.feet = 4;
cow.accordingToLarson = 'will take over the world';
```

We can also use a constructor function to simplify things, especially if we want to instantiate multiple different objects, e.g.

```
function Cow(colour, commonQuestion, feet, accordingToLarson) {
  this.colour = colour;
  this.commonQuestion = commonQuestion;
  this.feet = feet
  this.accordingToLarson = accordingToLarson;
```

```
6   this.moo = function(){ console.log('moo'); };
7 }
8
9 var cow = new Cow("brown", "what now?", 4, "will take over the world");
```

Note that with this approach it doesn't let you pass the functions into the constructor, they need to be predefined in the constructor function, e.g. the moo() function. In more recent versions of JS, based upon the ECMASCRIPT 6 specification you can also use the new class syntax that is similar to other languages.

```
1  class Cow {
2    constructor(colour, commonQuestion, feet, accordingToLarson) {
3      this.colour = colour;
4      this.commonQuestion = commonQuestion;
5      this.feet = feet
6      this.accordingToLarson = accordingToLarson;
7      this.moo = function(){ console.log('moo'); };
8    }
9  }
10
11 var cow = new Cow("brown", "what now?", 4, "will take over the world");
```

The only real difference between this and the previous example is the use of the class and constructor syntax. Yet another way to create objects is with the Object.create() method, e.g.

```
1 var cow = {
2     colour:'brown',
3     commonQuestion:'What now?',
4     moo:function(){ console.log('moo'); },
5     feet:4,
6     accordingToLarson:'will take over the world'
7 };
8
9 var cow2 = Object.create(cow);
```

Which creates a new object, cow2, whose parameters you can then update to differentiate it from the other cow object literal.

7.29 Exception Handling

Your programs are unlikely to be perfect, and even if they are, unexpected circumstances can still occur. To help deal with these circumstances we can use exception handling. We can handle run time errors by:

1. Trying to execute the statements
2. Catching any thrown exceptions
3. Finally cleaning things up.

e.g.

```
try {
    // Statements in which exceptions might be thrown
} catch(errorValue) {
    // Statements that execute in the event of an exception
} finally {
    // Statements that execute afterward either way
}
```

7.30 Best Practices

Just a few tips and best practices for starting off your JS programming style. Some of these are debatable but thinking about them will make you into a better web developer.

- Call things by their name easy, short and readable variable and function names
- Avoid globals
- Stick to a strict coding style
- Comment as much as needed but not more
- Avoid mixing with other technologies
- Use shortcut notation when it makes sense
- Modularise one function per task
- Enhance progressively
- Allow for configuration and translation
- Avoid heavy nesting
- Optimize loops
- Keep DOM access to a minimum
- Dont yield to browser whims
- Dont trust any data
- Add functionality with JavaScript, dont create too much content
- Build on the shoulders of giants
- Development code is not live code

The W3C keeps an annotated list[1] of best practices that is worth keeping an eye on as you develop your JS skills

[1] https://www.w3.org/wiki/JavaScript_best_practices

7.31 Summary

In thischapter we've started to explore the JavaScript language. We should now understand the role that JavaScript plays amongst the other core web technologies and have some idea of how JavaScript interacts with HTML, CSS, and the Browser. We've also surveyed the core JavaScript syntax so we are now ready to start exploiting it within our Web pages.

Chapter 8

Client-Side JS: Browser APIs

JavaScript is how we add dynamism to our web pages mostly through a set of global JS objects which provide access to the client environment. So now we have a working knowledge of Javascript, we'll start to look at how it can be used to manipulate the HTML that makes up our webpages, as well as to interact with the browser. If we think of the browser as the host environment that defines the space in which our Javascript functions run, then it is useful to explore what features of the environment are made available to us through the Browser's Application Programming Interfaces (Browsers APIs).

8.1 JS & HTML

In the last unit we considered how JS practically interacts with other web technologies. In particular:

- JS isnt self-hosted within the browser. That is we can't just retrieve a JS file directly from a web address and run it. The JS has to be hosted within a page. A minimal host document is required to cause the browser to then retrieve our JS. Once retrieved the JS is then loaded and executed by the browser's JS engine.
- JS can be:
  - added inline with HTML elements,
  - spread throughout our document, or
  - added as a block collected into a single place within a document, or as an external/linked file(s)

Note that our examples from the last unit were all run in the browser web console. Whilst strictly speaking this means we can run JS in the browser, it actually gives us a fairly rich environment for executing JS, learning the language, and experimenting, but it isn't generally how we would distribute our JS code to end-users

Well investigate this a little more and use it as a motivating framework for exploring interactions between HTML and JS. In the last unit we briefly surveyed three methods by which JS can be integrated with HTML. Let's recap those now.

8.2 Recap: A Simple Inline JS Example

In this example we have our JS entirely encapsulated within the onclick function of an HTML button element.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
      <button onclick="var textNode = document.createTextNode( Napier
          !');
      document.body.appendChild(textNode);">Hello</button>
  </body>
</html>
```

This means that our JS is localised entirely to the specific button element that it is attached to and can't be reused elsewhere, for example, if we have similar code attached to another button. This means that you need to repeat code across your pages, as well as mix JS amongst your HTML. Generally this is a bit of a "code smell" and, whilst useful for quickly trying out some code, is not an ideal solution.

8.3 Recap: A Simple JS Script Block Example

This version moves our JS into its own script block. Note that it doesn't do anything more than our last, inline, version. In fact it needs a little bit more code to achieve the same effect. This is mainly because the inline version is immediately attached to its HTML element and the context in which it runs. Now we've separated our JS from the HTML element it affects, we have to add additional code to get a 'handle' on the original HTML element. In this case we've achieved our aims by adding an ID to the button and then using getElementById to create the link between our JS and the button element.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <button id="hello_btn">Hello</button>
    <script>
        document.getElementById('hello_btn').onclick = function() {
           var text_node = document.createTextNode('Napier!');
            document.body.appendChild(text_node);
        };
    </script>
  </body>
</html>
```

Overall, the JS is now reasonably separated from the HTML elements within the page which means that functions can be more easily organised and reused in multiple places. Whilst the JS is still hosted within the parent HTML document it is encapsulated within

a pair of script tags. This actually makes it much easier to move to the next integration method...

8.4 Recap: A Simple JS Eternal Script Example

In this example our JS is now entirely stored within a separate file. First our index.html file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <button id="hello_btn">Hello</button>
    <script src="tmp.js"></script>
  </body>
</html>
```

and then our index.js file:

```
document.getElementById('hello_btn').onclick = function() {
    var text_node = document.createTextNode('Napier!');
        document.body.appendChild(text_node);
};
```

Notice that we still have a script tag in the HTML file, but it now just points to a web address where the JS is located and can be retrieved from. Again, the only link between the HTML element, our button, and the JS that is executed when the button is pressed is the button's ID.

The pattern found in this example is probably the most standard framework for relating JS and HTML, keeping a useful separation of concerns that can lead to better manageability and increased reusability of code.

8.5 A Slightly More Complex Example

The last example was fairly static though, it didn't do much other than wait for user input. The following example goes a little further. As well as giving us a button that, when clicked, adds text to the page, we're also actually building the page itself. Look at the body of our HTML file. It doesn't actually contain any HTML elements, so where is our interface coming from? It turns out that we're building it entirely through JS code! This is a really powerful opportunity that JS provides us for dynamically changing the pages that our users see.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="tmp.js"></script>
  </head>
```

```
6   <body>
7   </body>
8 </html>
```

and our index.js file:

```
1  function init() {
2      document.title = "Hello Napier Example"
3
4      var button = document.createElement("button");
5      button.innerHTML = "Hello";
6      button.id = "hello_btn";
7      var body = document.getElementsByTagName("body")[0];
8      body.appendChild(button);
9
10     document.getElementById('hello_btn').onclick = function() {
11         var text_node = document.createTextNode('Napier!');
12             document.body.appendChild(text_node);
13     };
14 };
15 window.onload=init;
```

8.6 The JS-HTML Relationship

On a practical level the JS-HTML relationship, where an HTML document references some JS that then runs, is a simplified way of looking at things. It is actually not merely the case that the browser retrieves an HTML document, which in turn references some JS, which then just runs.

The reality is a little more complex. You can get away with the above understanding but youll miss a lot of powerful functionality if you dont consider the wider context of the DOM (the Document Object Model) and the browser environment. A lot of rich possibilities are created by the set of APIs that the DOM and browser environment provide.

To understand this relationship, and to start seeing some of the possibilities that are provided, were going to primarily consider the following in this unit:

- The Window Object
- The Document Object

8.7 The Window Object Hierarchy

The Window object is the root of a tree of browser and web page related objects along with their associated APIs. One of the children of the Window object is the document object, which gives us access to the DOM. All of the other children give us access to the wider infrastructure that we can manipulate, all related to the browser window. Not only are we able to interact with the parsed HTML document (our current page) but also with a wide range of additional related aspects of the browser.

More specifically these are additional objects that JS can access and hence we can write JS code to manipulate them within the boundaries provided by their respective APIs.

Note that there are browser differences which mean that some objects are available on some platforms/versions and not on others. For the most part this is not an issue, but you should be aware of the possibility. The diagram gives you an idea of the range of objects that are accessible to our JS code.

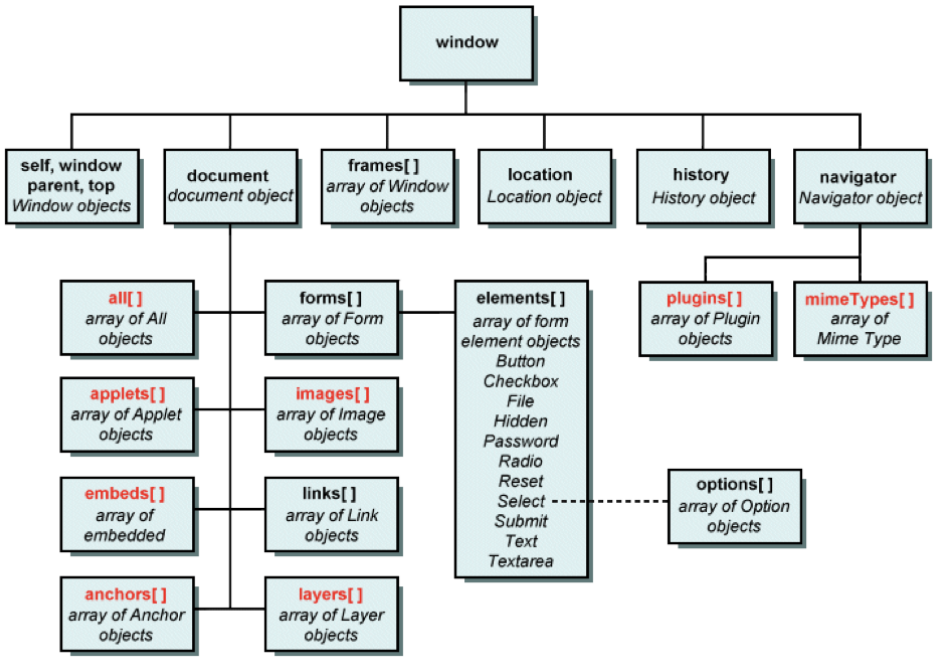


Figure 8.1

8.8 Global Objects

First an aside about global objects and scope. Generally our JS code is either associated with a Window and hence has a Window object associated with it, or else it runs in a separate background task within the browser. We'll not specifically cover background tasks here, they are a way to run your code "in the background" so that it doesn't interfere with the responsiveness of the current tab. This can be very useful for more involved JS applications so if you want to know more then it is worth researching the term "webworkers" if you do want to run bits of code independently in the background.

The window object is a global object. This means that it is always in scope or has global scope. Any JS variables that you create outside a function also have global scope. Additionally any JS functions that you create outside an object have global scope. What does this mean? That we can always access the Window object, from any part of our code, and that any global variables, or functions that we create are also accessible via the Window object. If we define a global JS variable or function then we can always access it via the window object as a property of the global object.

Let's look at an example of that:

```
1 var daka =   more  daka  ;
2 daka === window.daka;
3 // This will evaluate to true if the daka variable is available via the
     window object
4
5 function hi() {console.log(   h i   );}
6 window.hi === hi;
7 // Will evaluate to true for the same reasons as above
```

All we've done in the example is to create a variable and a function with global scope, then demonstrated that they are accessible through the Window object.

8.9 The Window Object

The Window object represents an open browser window, or a tab within a window, for browsers which support tabs. The Window object contains a document object, representing the DOM, the parsed and instantiated set of objects that the browser creates to represent the HTML element described in the source HTML document.

So our HTML page, as identified by the Web address we've navigated to, is a part of the Window object via the document/DOM child object, but it also gives access to much more.

For example, a single window could use HTML frames to display multiple HTML pages in the same window, so there is also support for listing, retrieving, and interacting with frames and hence the pages they contain, using the window.frames property.

Similarly we can find out information about the browser that our page is loaded into via the window.navigator property. This can be used to retrieve, for example, the name of the browser application and its version.

We can also interact with the history of the current window. By history we mean the list of pages that have been navigated since the current window/tab was opened. Each time you successfully navigate a hypertext link the new page is displayed and the last page is added to the history list. We can find out how long the list of previous pages is and we can use JS to navigate backwards and forward sequentially or to jump backwards and forward between pages. We can also interact with the location (URL/address) of the current page using the window.location property.

8.10 The Console Object because "There will be bugs"

The console object provides a really useful set of functions for debugging activities. When you write code, such as JS, you will create bugs. As your code becomes more elaborate, then you are more likely to accidentally introduce bugs. Whilst you will get better at noticing bugs related to syntax, for example, mis-spelling a keyword you will eventually start to create much more subtle bugs, the kind that don't cause a catastrophic

crash but are more subtle and corrupt your data or give rise to incorrect results. These bugs are much more difficult to solve. In addition to using the debugging tools that are amongst most modern desktop browser's suite of developer tools, you can also use the Console object from within your JS code to create log messages of information from within your program. These log messages are displayed in the browser console, and can also be searched and filtered using the browser console tools.

The console logging functions include console.log, console.info, console.warn, and console.error and can be used to segregate output that is merely informational from output that is important and giving details about errors. You as a developer must decide how important the output is and decide which style of console logging to use for each message that you want to log.

Logging works by passing a message, a string, to the function, e.g.

```
1 console.log("Hello Napier");
```

You can also pass CSS styles to logging functions, e.g.

```
1 console.log("%c hello", "background: #222; color: #bada55");
```

This can make the console much more easy to navigate so that you can find debug information when needed.

The console also provides access to several other useful debug features that are useful when developing and evaluating our code. These include timing functions, counting and grouping functions, as well as standard debug features like assert() and trace(). We'll look at examples of each over the next few sub-sections.

8.10.1 Console Timer Example

Sometimes we want to know when something happened, or how long something took to complete, so we use the console.time function to find out. In the following code we create a timer instance, "t1" then send log messages to that timer instance before finally ending the timer when we're done. Each call to timelog and timeEnd causes the elapsed time to be displayed in the console.

```
1 console.time('t1');
2 console.timeLog('t1', 'starting engines...');
3 console.timeLog('t1', 'in ur code, doing your things...');
4 console.timeLog('t1', "work complete, shutting down...");
5 console.timeEnd('t1');
```

8.10.2 Console Count() Example

Rather than how long something took, sometimes we just want to know how many times something occurred. This ability is what the count function gives us. In the following example we are using the count function to count how many times the greet function is called.

```
let user = "";

function greet() {
  console.count();
  return "hi " + user;
}

user = "bob";
console.log(greet());
user = "alice";
console.log(greet());
console.log(greet());
```

We can also give our counter a label, e.g."alice" so that it can be distinguished from the "default" counter like so:

```
console.count("alice");
```

8.10.3 Console Group() Example

Because our JS programs might become quite large and complex, it can be useful to group and organise our log messages. The console.group function enables us to do this. The console.group function hierarchically groups debug and log output together to make it easier to navigate in the console. Let's look at an example:

```
console.log("This is the outer level");
console.group();
console.log("Level 2");
console.group();
console.log("Level 3");
console.warn("More of level 3");
console.groupEnd();
console.log("Back to level 2");
console.groupEnd();
console.log("Back to the outer level");
```

Running this code in the browser console will give output similar to this:

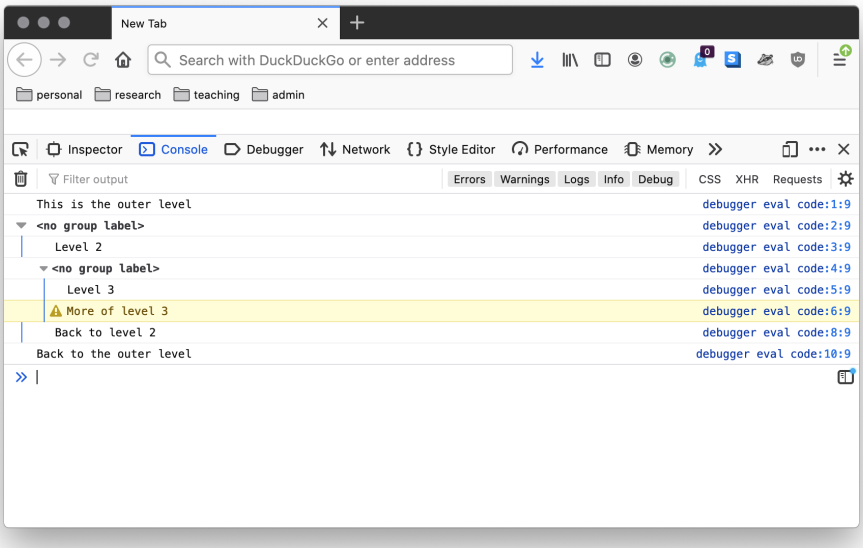


Figure 8.2

8.10.4 Console Assert Example

Assertions are used in many programming languages as a way to verify that our expectations about the way that a piece of code works actually match the reality of how it works. The basic idea is that we make an assertion, i.e. the value of variable x at this point in the program is y, and then the actual values of x and y are compared to each other and the assertion is true if they match and false otherwise. Let's look at a small example:

```
1 const errorMsg = 'the # is not even';
2 for (let number = 2; number <= 5; number += 1) {
3     console.log('the # is ' + number);
4     console.assert(number % 2 === 0, {number: number, errorMsg: errorMsg
          });
5 }
```

Running this code in the browser console will give output similar to this:

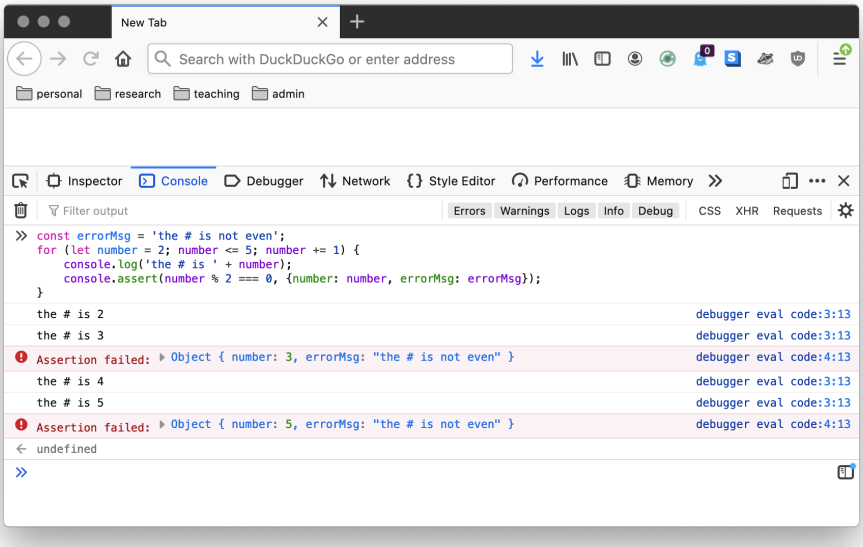


Figure 8.3

8.10.5 Console Trace Example

Sometimes we want to trace the stack of function calls that have led to the current situation in our running JS code. The console.trace function lets us do this.

```
1 function foo() {
2   function bar() {
3     console.trace();
4   }
5   bar();
6 }
7
8 foo();
```

Running this code in the browser console will give output similar to this:

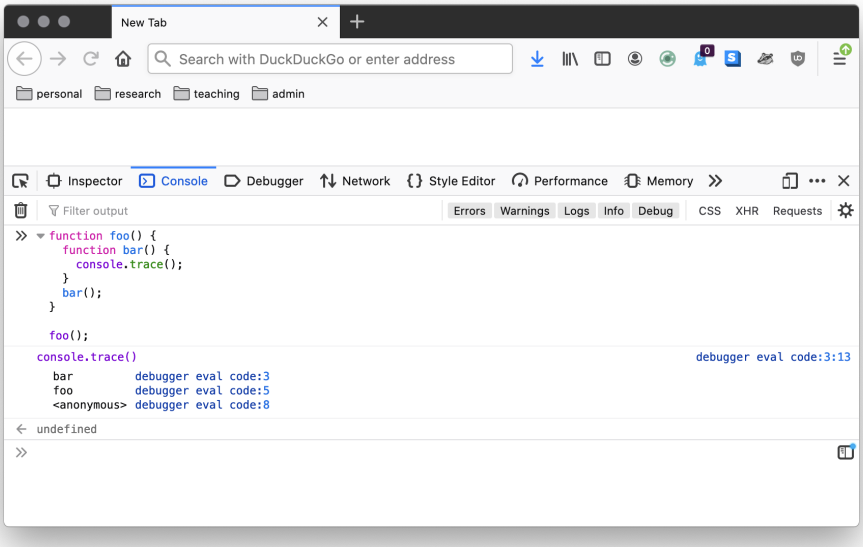


Figure 8.4

8.11 History Object

Returning to the Window object again, let's examine the History object. This stores the list of URLs that the user has visited within the current browser window or tab. It doesn't give access to other tabs however, only the current one. Access to specific URLs that might previously have been visited is prohibited, as is access to other windows or tabs that the browser might have opened. This is for both security and privacy reasons and aims to keep individual windows and tabs relatively isolated from each other.

The History Object has the following properties:

- length - the number of URLs in the history list

The following methods are also available for use:

- back() - Load the previous URL
- forward() - Load the next URL
- go() - Load a specific URL from the list by jumping n places (positive or negative) to move that amount through the history list

Note that the history object is not standardised but is supported in the above format by major browsers, so you might find some more esoteric browsers whose behaviours differ.

8.12 Navigator Object

The Navigator object stores information about the browser. The properties that are available and which can be retrieved include the following list (which is not exhaustive):

- appName - Returns the browser name
- appVersion - Returns the version of the browser
- cookieEnabled - Are cookies enabled?
- platform - Which OS is this browser installed on?
- userAgent - Access the header sent by the browser to the server to self identify

Note that again, the navigator object is not standardised but is supported in the above format by most major browsers.

8.13 Screen Object

Similar to the Navigator object, the screen object is a source of information about the screen that the users browser is displayed on. This can be useful, for example, when using different layouts and trying to decide when to switch from one to another.

The supported properties include:

- availHeight - Height of screen minus certain furniture, e.g. windows taskbar
- availWidth - Width of screen minus certain furniture, e.g. windows taskbar
- height - Total height of the screen
- pixelDepth - Colour resolution in bits per pixel
- width - Total width of the screen

Note that, as for the Navigator and History objects, the Screen object is not standardised but is supported in the above format by major browsers (do you notice a theme developing here. . . ?).

8.14 Document Object Model

Now, having surveyed all of the other child objects of the Window object, let's turn our attention to the Document object and the Document Object Model (DOM). When an HTML file is retrieved and loaded the HTML tags are parsed into a hierarchical data structure, a tree, that more-or-less matches the structure of our HTML. The key difference is that this data structure, the DOM, is easier to manipulate, i.e. from JS code (or CSS). The browser uses the DOM during the rendering process and is a critical underpinning metaphor or model for the way that most modern browsers operate. Once an HTML file is parsed, the browser ignores it and prefers to use only its internal DOM representation of the page. Here is a basic illustration of a simple HTML document parsed into a DOM representation. Obviously things can become much more complex than this.

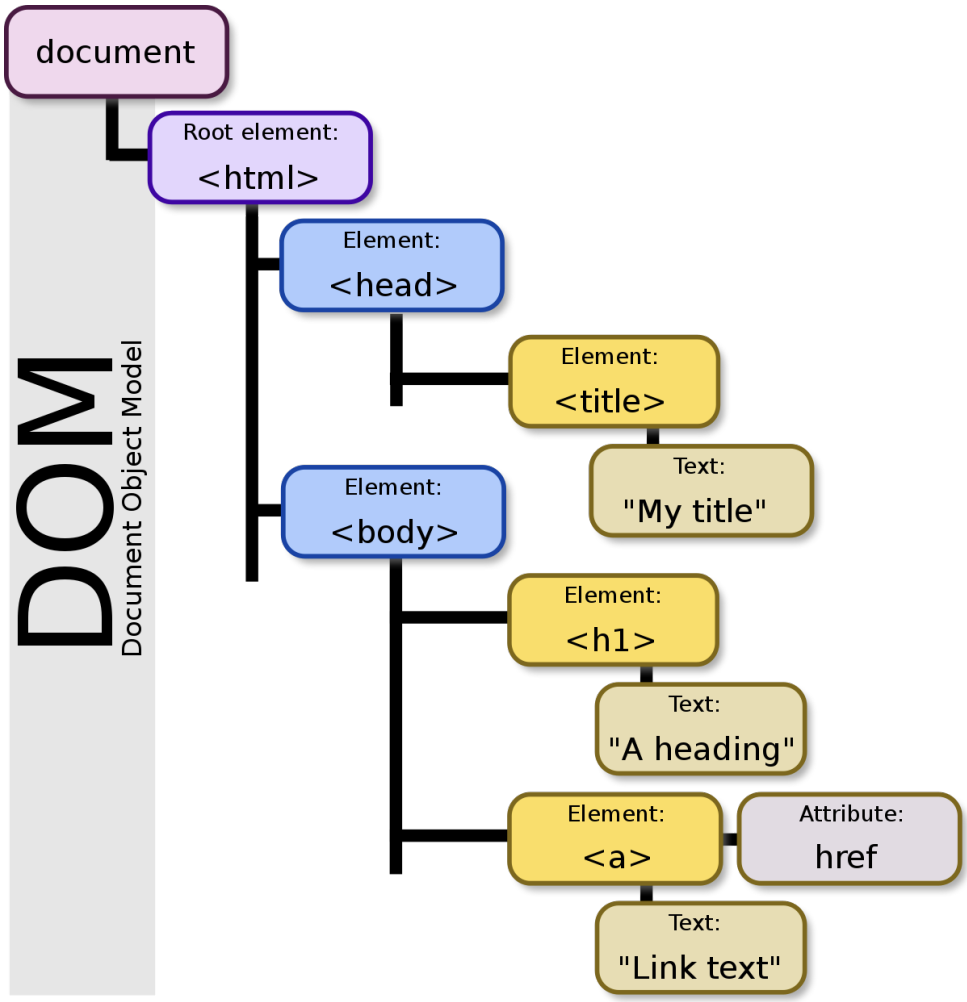


Figure 8.5

8.15 DOM as API

The DOM is a programming interface (API) for HTML. The DOM represents the page so that their structure, style, and content can be changed without needing to change or manipulate the source file(s) that define the page. This means that if we refresh a page, the result is a new instance of the page parsed into a new DOM, regardless of any changes that we'd made before the refresh. Note that this only holds if extra measures haven't been taken to persist any changes, for example, by using a cookie or browser local storage.

The DOM operates in an Object-oriented way. Logical elements of the page are exposed to the user, who in this case is the programmer, as a containment hierarchy of objects which can, in turn, be modified.

Be aware that many browsers extend the basic DOM to provide browser specific features but there is a common core that is shared across most modern browsers based upon the standards defined by the W3C and WHATWG.

8.16 Accessing the DOM

Our browsers JS environment gives us access to a global document object which we can access through the document object or else via the Window object, e.g.

```
1  window.document
```

Strictly we should access the document object and its contents as follows:

```
1  window.document. ...
```

but both the window and document are exposed to JS as global objects so you might either approach both in real world code. As with most things, choose a style and be consistent in your own code, at least within any given project.

It is a useful exercise to explore the document object in the browser console using the autocomplete function. Type in document and your browser should display a popup list of available attributes and functions that you can explore. This is a useful way to find out what is available on any given page.

Once you access the DOM via the document object you can subsequently access, and explore, the documents objects using the dot (.) operator.

Try this out with some of your existing pages that you created earlier. Load one into a tab then open the browser console and explore which elements of your pages are exposed and available for manipulation. You should notice that, for example, the head and the body and the elements that they contain, and so on, are all accessible from JS. It is also worth noticing that what the DOM makes available to you is heavily dependent upon the contents of the specific page that you are on. A different page will likely have a very different DOM representation in terms of the detailed contents of the page, even if the broad structure, head and body, are the same.

Using the DOM, we can not only access but also modify, add, or remove elements so that we can change the contents of the current page, or even completely replace all of the HTML content with new content from code.

8.17 Accessing Elements

One we have access to the DOM via the document or window.document objects we can access the elements within the page using the getElementById function, as we saw in previous examples:

```
1 <!DOCTYPE html>
2   <html>
3     <head>
4       <title>SET08101 - Interacting with the DOM</title>
5     </head>
6   <body >
7     <p>
```

```
8       <a href="#" onClick="addMessage();">Click Me</a>
9     </p>
10    <h1>OUTPUT:</h1>
11    <p id="outputDemo"></p>
12    <script>      function addMessage() {
13      document.getElementById("outputDemo").innerHTML = "HELLO NAPIER
14      }
15    </script>
16  </body>
17 </html>
```

In addition we can use other methods to retrieve elements of the DOM, especially if there isn't a specific ID to use. For example we have:

- getElementsByClassName()
- getElementsByName()
- getElementsByTagName()

Which enable us to retrieve collections of elements that match either the specific class name attribute, name attribute, or tag name. Additionally there are also the querySelector() and querySelectorAll() methods which can be used, respectively, to access either the first element in the DOM that matches a given CSS selector or else all elements that match the supplied CSS selector.

8.18 Adding & Manipulating HTML DOM Elements

Just retrieving the contents of the page using the various get methods doesn't really help us much more than we could achieve if we just read the entire web page. So there are also mechanisms to let us manipulate the DOM by adding in new HTML elements, styling them with CSS, or attaching JS code to them. In the following example we create a new <h1> element then create some text in a text node. We attach the text node to the <h1> element by appending the text node to it as a new child which causes the child text to effectively be enclosed in the parent <h1> tags. The new heading tag and its child text are then appended to the document's body which causes the new heading to be displayed.

```
1  <html>
2    <head>
3      <script>
4         window.onload = function() {
5
6           const heading = document.createElement("h1");
7           const heading_text = document.createTextNode("DAKA DAKA");
8           heading.appendChild(heading_text);
9           document.body.appendChild(heading);
10        }
11     </script>
12   </head>
13   <body>
14   </body>
15 </html>
```

In addition to appending a child, all DOM elements let you manipulate them in a consistent way using the DOM element object API, for example using the various insert methods to place elements specifically in relation to existing elements instead of merely appending them. A summary of available methods for manipulating HTML DOM elements can be found in the W3Schools documentation[1].

8.19 Resources

There are many source of information about the Browser-based JS APIs:

- The MDN JS Reference[2] not only includes reference information but also a set of tutorials for exploring JS
- W3Schools JS and HTML DOM reference[3]
- W3schools JS Tutorial[4]

8.20 Summary

In this chapter we've examined the JS APIs that most modern browsers support. Briefly we've considered:

- Looked at the role that JavaScript plays amongst other web technologies
- Gained an understanding of the core JavaScript syntax
- Discovered how JavaScript interacts with HTML, CSS, and the Browser

[1] `https://www.w3schools.com/jsref/dom_obj_all.asp`
[2] `https://developer.mozilla.org/bm/docs/Web/JavaScript`
[3] `https://www.w3schools.com/jsref/`
[4] `https://www.w3schools.com/js/DEFAULT.asp`

Chapter 9

Client-Side JS: Data Storage APIs

The web is for sharing information and sometimes we want to change the information on webpages and have our changes stick around for longer than our browser tabs are open. In this chapter we'll continue to build on the last two topics, but this time focusing upon a specific set of APIs that the browser makes available for storing data. This is what we can use to save data in the browser so that it is still there if we reload a page, very useful if we want to build a web app with persistent functionality or if we want to create a nice user experience.

Well examine the broad range of options for web-oriented storage on the client (with a little consideration of the server). This involves not only an awareness of specific technologies like the APIs for cookies or web storage in the browser, but also how client-side JS can make use of databases to persist information and the limitations that working on the client side impose.

We'll also take a look at the JSON language as the default JS way to describe and structure our data for serialisation.

Briefly, we'll cover the following major client-side data-storage and data-representation related topics:

- HTTP Cookies
- DOM Storage
- Indexed DB
- JSON

9.1 Client Side Storage & Data Persistence

Data on the web can be ephemeral. We will often visit pages, interact with the contents of the page, and leave without caring that our data is lost. However occasionally we will want to persist our data. This is increasingly true when we consider the move towards web-based applications, as opposed merely to web-sites. Similarly, whenever we offer our user options to personalise their experience, it makes sense to save those options somehow so that when they return, things are still saved the way they left them. Even if they closed

all their tabs and restarted their browser in-between, once they revisit our site, their data is as they left it.

Note however that client-side storage has one major drawback, if a user clears their browsers cache, or other data storage associated with their browser, then all of the information that we are considering storing on the client side will be lost. This is the main downside to client side storage, but, to be honest, it is a downside that is surrounded by other advantages.

Generally client side data is already on the client so it makes sense to persist it on the client if only the client needs to reuse that data. This is good enough for many circumstances. However, a more robust solution that allows the user to move between browsers or machines will require data to be stored on a server. Similarly, if you have any functionality that involves aggregating data, such as social media style functionality, then this usually also requires data to be stored on a server. Similarly, moving data to and from the server involves additional steps which can introduce opportunities for your system to either fail, or run in a less than optimal way. However, storing data on a server is outside the remit of this module, and also raises additional considerations related to security and privacy of user's stored data.

If we architect our sites well then our user's data might never have to leave the client and for many sites, this is perfectly fine.

9.2 Client Only Data

If the server never sees any data, if it only provides the interface and web-app or site for working with the data entirely on the client side then this simplifies a whole series of issues:

1. Client side data has good privacy preservation. If a user's data never leaves their browser then it is hard for it to be misused, abused, or lost. This greatly simplifies the legal position for web developers.

2. There is a different security model between client-side only data and data that moves from client to server. In the former, an attacker must target individual clients to access data. In the latter there are more points to target, not only individual clients, but also the server and also the data in transit between client and server.

3. There is no need for potentially expensive data storage or management facilities and no need to determine when to keep data and when to discard it.

4. If data isn't moved to the server then there is no need for user management facilities to support separating data and making it only accessible to the right users.

However, there are limitations to what you can do if your user's data is only on the client:

1. There is no way to easily aggregate data and provide facilities based upon that aggregation. This means that crowd based data analysis or features based upon multiple users interacting aren't possible.

2. It's not easy to facilitate user-to-user communication unless mediated by your server, in which case data passes through you possession. This means that putting different users in touch with each other is currently difficult to do. Note that the WebRTC protocol and API have been moving rapidly towards direct browser-to-browser communication but we are not there yet. Expect to see this in the short to medium term and take note that it will have an impact on the design and implementation of web sites.

9.3 Ephemeral Client Data

There are multiple ways to store data on the client side, that is, within our user's browser. This is regardless of whether any of that data has previously been on a server or whether it has been entirely captured or generated on the client. Our users data never has to leave their browser unless there is a good reason for that to happen. One reason is that data in the client can be considered to be ephemeral, lasting perhaps for only a short time, despite the opportunities that developers have to persist data.

The shortest that client side data can last is as long as the browser tab is open. Once the tab, or window, is closed any data within it will be lost. Note that some of the data might persist automatically within the browser's cache, but if the page is reopened, even using cached information, a fresh DOM will be created and anything, like user input, from the previous tab, will be lost. Our use of tools like cookies, DOM storage, and IndexedDB are merely means to extend how long this data can last, firstly beyond closing and reopening a tab or refreshing the page, but conceivably much longer.

Ultimately though, client side data must always be considered, ultimately, as ephemeral. A user might delete their local data, wipe cookies, clear caches, re-install their operating system, or use a site from a different machine. All of these can affect the availability or otherwise of any data that the user has previously created or generated.

So client side storage should be treated differently to server side storage. To some degree it is permanent storage and will live for as long as needed but there is no guarantee. If we need to persist data beyond the examples listed above then we need to consider saving data outside the browser, to the users local file system, or else storing the data on a remote server. Either solution brings its own additional challenges. Note though that it is seldom all or nothing, client side persistence of information can enable a site to continue working even though there is no network connection, so having a default design approach of "offline first" is not a bad idea. This way client side data can be treated as ephemeral, with the expectation that it will be downloaded from or synchronised with a server again if necessary at some point in the future. As a result we should consider this, when appropriate, in our designs of the client side architecture of our sites.

9.4 Client Side Options

On the client we have three basic options for data storage and persistence. These are, in order of increasing complexity, power, and storage size:

1. Cookies
2. DOM Storage (also known as local storage or web storage)
3. IndexedDB

All three are provided by Browser based APIs which are accessible through JS. This means that you can usually investigate, explore and exploit them direct from the browser console, but usually we would do so from JS within a site that weve navigated to.

Cookies are the simplest, a single named string, associated with a given web address that contains data about the current page. This can be useful for very simple information, such as security tokens, usernames, or small amounts of data like user settings that you want to persist if the page is reloaded. DOM storage is designed to overcome some of the issues with cookies, but at the risk of being slightly more complicated to implement and manage. Whilst cookies are designed primarily for communicating data between client and server (hence the security token suggestion above), DOM storage is designed for client side scripting and data persistence for web apps and isn't communicated to any associated server in the same way that cookie strings are. IndexedDB is the most complex client side data storage mechanism available to us and is an API for managing a NoSQL style database of JSON objects. The amount of data that each can store increased, with cookies supporting the least, and IndexedDB, the most. As the amount of data storage has increased so too has the complexity of each solution, as well as the amount of code needed to effectively manage each approach.

Let's now look at each in turn, starting with cookies.

9.5 Cookies

A cookie is a small piece of data that is usually sent from a website server to the client machine/browser where it is then stored. Cookies can also be created, on the client, through JS so can be misused for data persistence solely on the client.

A cookie is actually a single string and is transmitted in the HTTP headers during client-server communication, for example, as part of the process of retrieving a web page during an HTTP GET transaction.

Cookies are designed to be a reliable mechanism for maintaining information about state, i.e. keeping track of things across pages during navigation. This enables the stateless nature of the underlying HTTP protocol to be avoided and is a useful and important technique in those circumstances. Cookies can also be used to record activity, such as where you click, your user credentials, your log-in status, your history of interaction on a site, as well as arbitrary information that your pages might need to use, such as names, addresses, etc.

Cookies have a poor reputation, mainly because they have been greatly misused over the years in privacy and security impacting ways. This has led to laws restricting their use in tracking people and requiring prior notification of their use, through effective informed consent, before non-essential cookies can be used. Note the important point about "non-essential" use however, cookies are perfectly fine to use if they are essential to the functionality of your site and don't track users or otherwise invade their privacy, i.e. you cannot just deem everything to be essential and then not inform your users of their use. The non-essential clause is not a "get out" for ignoring the user notification laws but is a recognition that there are important and valid uses for cookies and that under some circumstances they can and should be used to provide essential functionality.

9.6 Creating, Reading, Updating, & Deleting Cookies

Cookies can be created and manipulated in many ways. They are often created on a Web server and then transmitted through the HTTP headers to the client during an HTTP request-response cycle.

They can also be created through JavaScript using the document.cookie property of the browser API and are thus a useful way to store small volumes of data. Data in a cookie is a name-value pair, e.g.

```
1  name = Carol Danvers
```

In JS a cookie can be created very easily using something like this:

```
1  document.cookie = "username=Carol Danvers";
```

Cookies can easily be retrieved into a JS variable for the current page, e.g.

```
1  var new_cookie = document.cookie;
```

We can also update an existing cookie the same way it is created (causing existing cookie to be overwritten):

```
1  document.cookie =  username=Captain Marvel ;
```

Note that we dont explicitly need to delete cookies, instead we can just set an expiry parameter like so:

```
1  document.cookie =  username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
       path=/;   ;
```

9.7 DOM/Local/Web Storage

DOM storage (also known as local storage or web storage) is a set of APIs to provide persistent storage for use in web apps. Despite the different names, they refer to the same set of APIs and are dependent upon browser support in your users client software. We'll refer henceforth to DOM storage for simplicity. Whereas cookies were designed for moving ephemeral user specific data between server and client and vice versa their use for data storage is really a misuse. Dom Storage is designed to support the needs for most client side web apps which need to store data locally.

DOM storage is similar to cookies but with greater capacity, a cookie is limited to 4KB whereas DomStorage can store 5, 10, or 25MB of data, depending upon platform and browser. DOM storage uses two global JS objects, sessionStorage and localStorage, which provide two related but different levels of data storage functionality.

The sessionStorage object is for data that is limited to the lifetime of the window and is on a per origin, window, or tab basis. It enables separate instances of the same web application to run in different windows without interfering with each other.

The localStorage object is persistent after the browser closes and is per origin where origin is a combination of protocol, hostname, and port number. A localstorage object instance is available to all scripts loaded from pages with the same origin and therefore is useful if you need multiple pages, for example, multiple instances of the same web-app in different tabs, to share data amongst themselves.

We can store a value for the rest of a session very simply:

```
1  sessionStorage.setItem( key , value );
```

Then we can retrieve a value later by supplying the key (so we need to keep track of keys or have a way to supply them within our web-apps):

```
1  alert(sessionStorage.getItem( key ));
```

Note that localstorage works in an almost identical fashion, just with different object names to differentiate local from session.

Store a value beyond current session:

```
1  localStorage.setItem( key , value );
```

Retrieve value:

```
1  alert(localStorage.getItem( key ));
```

9.8 Indexed DB

Indexed DB is a W3C recommended standard for a low-level browser API for client-side storage of a local, transactional database. This is essentially a full-blown database hosted in our browser rather than merely a key-value datastore. This enables sites to permanently collect and save large amounts of structured data. Note that our notion of permanence here is still subject to the discussion of ephemeral data from earlier.

Indexed DB is a form of NoSQL storage and basically creates collections of indexed JSON objects. There is preliminary support in many browsers, including, Firefox, Chrome, Safari but it is worth allowing the technology to mature somewhat before you rely upon it. As a result it is included here for reference and completion, and to prepare you for the future.

Indexed DB promises a lot of power for data storage and manipulation in our web apps. It is aimed at browser implemented functions (such as bookmarks) and web applications (like email clients). For our purposes, and most websites, Indexed DB is overkill where DOM Storage or alternatives that are hosted entirely within JS, like PouchDB, JsStore, &c., are more appropriate

9.9 JavaScript Object Notation (JSON)

Now we need to take a slight diversion to fill in a gap in our knowledge. Weve mentioned JSON very briefly in passing so far, but as we start to build more feature rich sites, we'll have more data to structure, to manipulate, and to store. Our data won't live all of its time within our code or within working memory, so serialisation formats can be useful. "Serialisation" is just a fancy term that is used to describe turning the data used within a program into a form that is better suited for other purposes, like storage, or human inspection, outside of the running program.

Although it was originally developed for serialising JS Objects, JSON is now a de facto generalised standard for describing data, particularly in the Web, but also, increasingly, within many other computing scenarios, e.g. saving data from a desktop application, transmitting data across a network, or saving data from instruments or sensors.

Data is often sent to web-apps, and retrieved from web-apps, in JSON format. There are other formats but JSON has become the default, probably due to the close relationship between JSON and JS. Data is often also stored as JSON either in the filesystem or in data-stores. Data is often manipulated within JS as JSON. As a result we should probably take a closer look at it.

JSON is an acronym for JavaScript Object Notation and it is yet another language, but the last one we'll consider for this module. Really though, it is so closely related to its parent JS that if we have gotten to grips with JS, and specifically with JS objects, arrays, and variables then JSON itself should not present any major surprises. Really, if you already know JS then JSON can be considered as more of a different perspective on some things that you should already know than as a new language. JSON is basically

a transliteration from objects within JS into a plain text format that can be saved and reused in other contexts including outside of JS. Note that JSON within JS and JSON outwith JS look almost identical in practice. So if you have become familiar with JS objects then JSON isnt going to be a huge challenge.

It's worth also relating JSON to HTML. Neither is a general purpose programming language, they both lack features necessary to be programming languages, but they are both used for specific tasks related to describing data. In the case of HTML, for describing the structure of human knowledge in the form of hierarchically organised text documents, and in the case of JSON, for describing hierarchically organised collections and structure of knowledge.

9.10 JSON Examples

Before looking in more detail at JSON as a language, it is worth seeing an example of some actual JSON first and comparing it to the equivalent JS object. The following should look reasonably familiar from our previous discussion of JS objects:

```
1 var cow = {
2   name:  daisy",
3   likes:  clover  ,
4   moo:function() {
5     console.log(  moo  );
6   }
7 };
```

This is our construction of a JS object to represent our cow "Daisy". Daisy has a name attribute and an attribute that describes something she likes. Daisy also has a function, moo, because all cows moo.

In the following example we see the JSON serialisation of the Daisy cow object:

```
1 {
2     name  : daisy",
3     likes  :  clover
4 }
```

That's it. That's our Daisy object described in JSON. Notice that only Daisy's attributes are recorded in our JSON document. The function has been discarded. Only data, or attributes are stored in JSON files so we might have to account for this in our programming. This is a pretty-printed example of JSON which uses whitespace and indentation to show the structure of the data for human readers. Frequently though, you might see JSON represented like this:

```
1 {   name  : daisy",   likes  :  clover   }
```

Everything is presented inline and without additional formatting to make it easier to read. Sometimes you might need to copy such examples into a tool like JSONLint (which we'll see later) to get the JSON to be automatically checked and structured for better human readability.

9.11 JS $\longleftrightarrow$ JSON

We can move backwards and forward between JS objects and JSON using some standard JS functions. We can use JSON.stringify to turn a JS object into a JSON string:

```
1  JSON.stringify(js_object);
```

More formally, this will serialise the JS object "js_object" into a string. For the moment let's just consider a string to be a linear sequence of characters that are enclosed within double quotes but soon we'll see more formally, in a railroad diagram, the full extent of the range of characters that can be used to build a valid JSON string.

We can also take an arbitrary, but correctly formed JSON string and parse that into a JS object hierarchy using the JSON.parse function:

```
1  JSON.parse(json_string);
```

This will parse, or "read and interpret", the string identified by "json_string" and construct a JS object hierarchy for use in our program.

For our purposes, if we want to manipulate data (JSON) in code then our data needs to be parsed from its string representation and turned into an object within our program. Conversely, if we want to store data from our program or send it elsewhere, such as to an API that consumes JSON data, then we need to stringify our JS objects into a JSON document.

Note that in practice it's not always quite that simple, but it is sufficient for our purposes on this module and for most client side web development.

9.12 JSON as a JS-independent Language

Rather than being an ad-hoc serialisation of JS Objects, JSON has also been formalised and standardised in its own right. JSON can be used standalone to describe the structure of data and, as a result, is now used in many non-Web and non-JS contexts as a general purpose data description language. So a comparison to other such languages and formats, like XML, YAML, and RDF, to name just three, is appropriate.

It is fairly straightforward to describe the main points of JSON:

- JSON is a plain text format. You can create and edit JSON using just your text editor. This is easier once you are used to what the language allows. In the meantime, using a tool like JSONLint.com (which we'll see soon) can be useful in helping you to get it right. That said JSON is not a general purpose programming language, it doesn't let you do iteration or selection or evaluation of data, it just enables you to describe it. If you already know JS and are familiar with JS objects and variables then JSON should make plenty of sense.

- A JSON document is iteratively constructed. You start with either an object or an array [] and then add values to them (including possibly other sub-objects or sub-arrays) ad infinitum until you have described everything you need to describe.
- A JSON array can store a comma separated list of values.
- A JSON Object can store a comma separated list of key:value pairs.
- JSON Values are objects, arrays, strings, numbers, booleans, or null

Even if you build a JSON document independently from your JS, you can still read that into your JS programs and instantiate a collection of data or objects for use within your program.

An easier way to see how all of this fits together is to visualise the language using railroad diagrams which is what we'll see next.

9.13 JSON RailRoad Diagrams

Railroad diagrams are a useful way to understand formal programming languages. Just like a train on rail tracks, which has no choice but to follow the tracks, these diagrams are designed to be read by following the line omissions from one side (in this case starting on the left) to the other. The tracks branch and rejoin in ways that allow only certain words or constructs from the language to be included at any given time, and support sequence, repetition, and omission of words in line with those allowed by the language.

A Valid JSON document contains either an object or an array, construction of which is shown in the following two diagrams. The first diagram shows us how to build an object, and the second, how to build an array. An array is a comma separated list of zero or more values, all enclosed in a set of square brackets. Meanwhile an object is a comma separated list of zero or more string:value pairs, this time enclosed in curly braces.

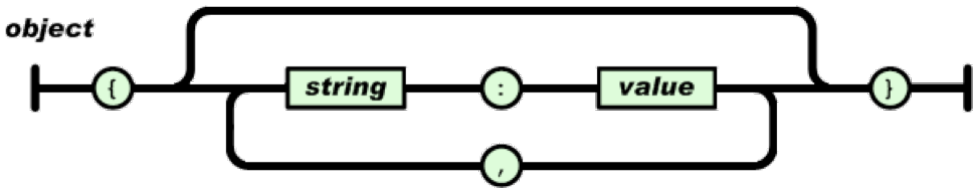


Figure 9.1

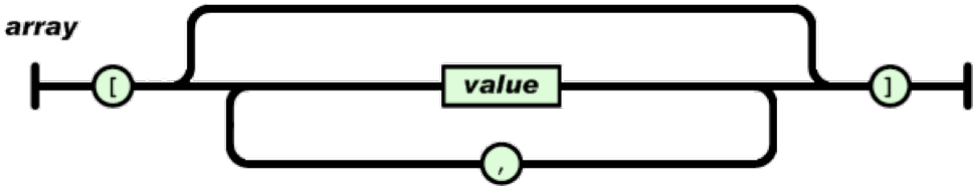


Figure 9.2

The values supported by JSON, together with the railroad mechanisms for constructing strings and numbers are given in the following diagrams.

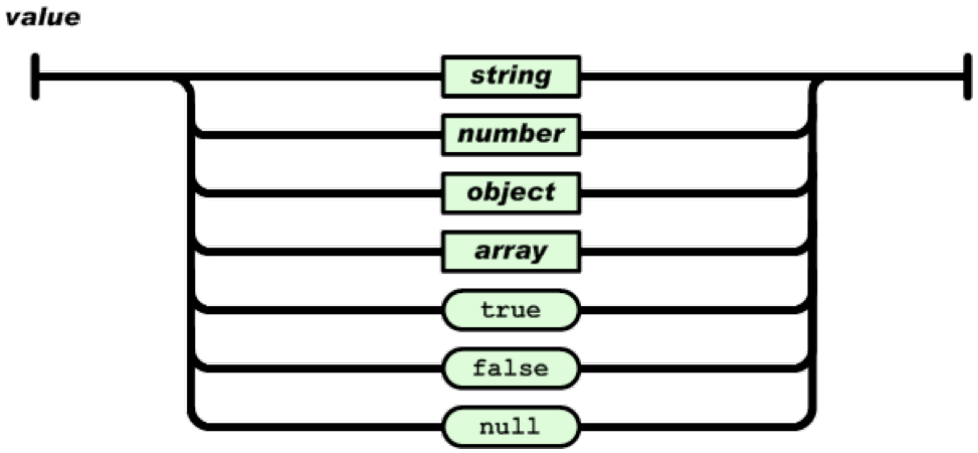


Figure 9.3

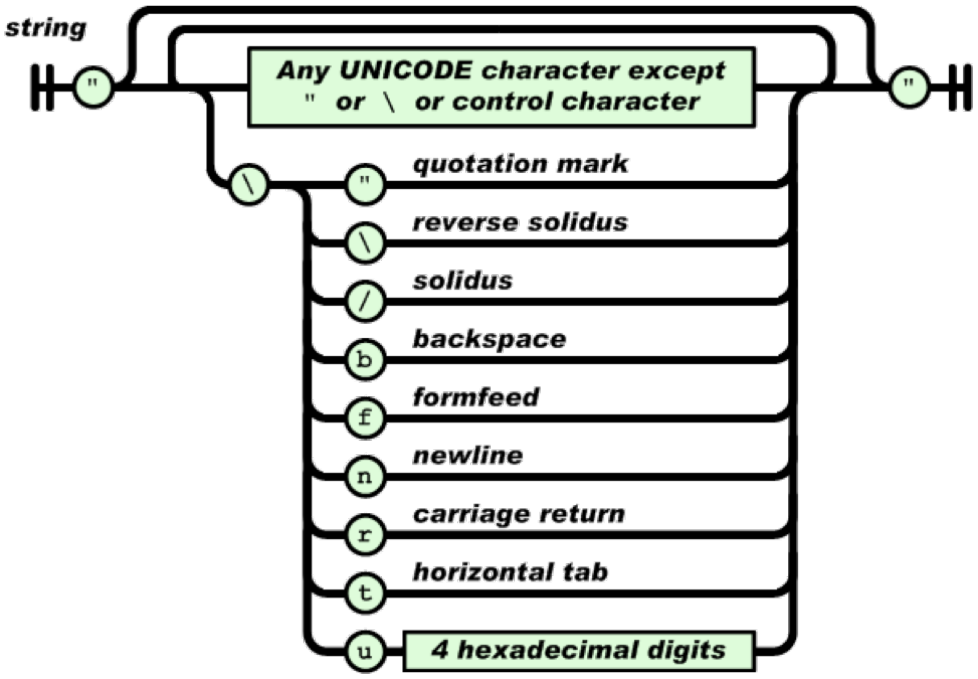


Figure 9.4

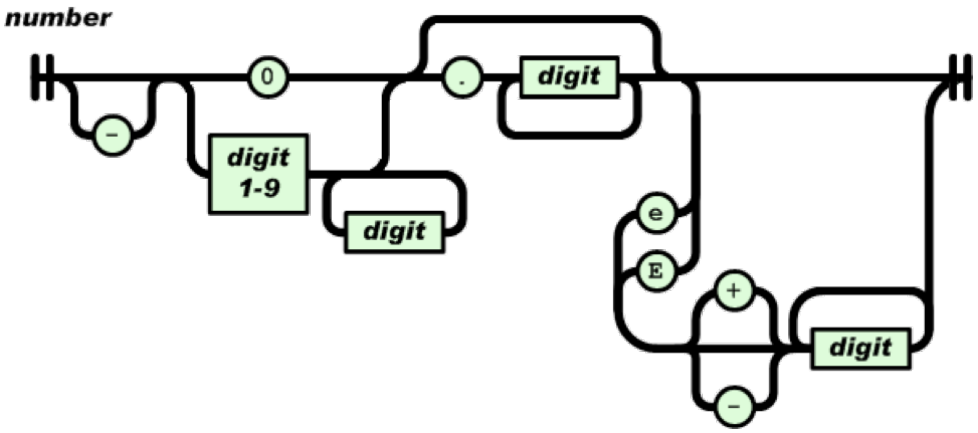


Figure 9.5

9.14 JSONLint

Linters are useful tools that are available for most programming languages. They are usually fairly simple programs that you can use to flag up programming errors. The kinds of errors that a Linter might bring to your attention can include syntax errors, certain kinds of bugs, stylistic issues, and various kinds of "code smell". Note that Linters can't (yet) check for more complicated or subtle bugs, for example, those arising from logical errors of reasoning that stem from the programmer, or from errors that arise because the problem and solution are underspecified, or because the programmer isn't sure what their goal is. Whenever you learn a new programming language it is worth looking for a linter tool to support you in using the language correctly and picking up on simple issues. Note that there is some overlap between a linter and the kinds of checks that some compilers and interpreters do but rather than going through the entire build process, which can be time consuming, a linter can process your source code and more rapidly give useful immediate feedback about certain categories of problems.

JSONLint[1] is a very useful online tool for quickly editing ad validating JSON files. You can find it here:

It's really simple to use, edit some JSON either in your own files then copy/pasting or directly into JSONLint's interface then press the "Validate JSON" button. The JSON is either valid or else you will be shown where the error lies. Once everything is fixed then you can copy and paste the nicely formatted JSON data back to your own text file and save for usage elsewhere.

[1] `https://jsonlint.com/`

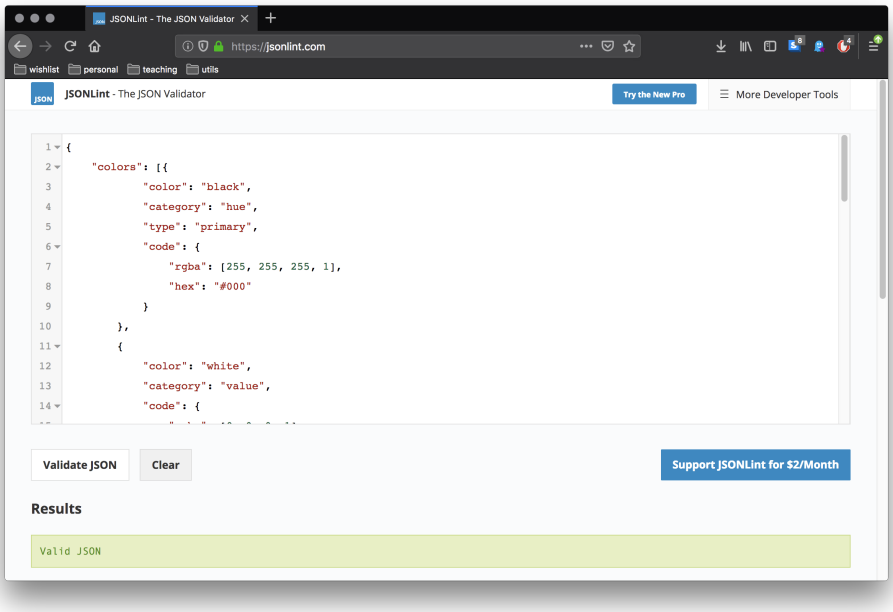


Figure 9.6

Note that JSONLint is also great for when you receive JSON from elsewhere, perhaps as the result of an API call. JSONLint will format the JSON and enable you to more easily see the structure of the data within the JSON document.

9.15 Why is JSON important?

If you are going to build APIs, or really exploit JavaScript, or share data with other APIs (or retrieve data from them) then you will need to use JSON. Really, you can't effectively use JavaScript objects without getting a crash-course in JSON anyway, but JSON also has a role beyond being the text based serialisation format of JSON objects.

There are many other data transport and representation languages, e.g. XML, YAML, RDF, but JSON occupies a sweet spot that:

- is not too complex so you can get up and running swiftly and can quickly prototype ideas.
- tooling is lightweight you don't need any specialist tools, you can just write JSON into a regular text file (although as we saw earlier a Linter tool can be useful to help avoid simple mistakes).
- is human readable you should be able to read through a JSON document and interpret the structure of the data that it describes. If a JSON file isn't easily readable then that is usually either a failure of the developer of the file to ensure that it correctly communicates the data or else means that there is some specialist knowledge required to interpret the actual data itself, unrelated to the JSON representation.

Yes, there are drawbacks to JSON (stackoverflow is a great place to find discussions between professional developers on the merits, or otherwise, of JSON), but the positives make it an easy tool to choose and use. This is probably why JSON is frequently a go to choice for data description and representation even amongst developers who are not doing web development or using JavaScript.

9.16 Resources

As usual, due to the rapid pace of change in web technologies, the best place to get up to date information about the current status of browser related data storage APIs is from the Mozilla and Google documentation.

- MDN Cookie API[2]
- MDN Web Storage API[3]
- MDN Indexed DB API Documentation[4]
- Google Developers Documentation (Indexed DB)[5]

9.17 Summary

In this chapter weve looked at the following broad topics:

- Identified the current main approaches to data storage for the web
- Distinguished between server-side and client-side data storage
- Made appropriate choices about which technology to select for a given problem

[2] `https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie`
[3] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API`
[4] `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API`
[5] `https://developers.google.com/web/ilt/pwa/working-with-indexeddb`

Chapter 10

Client-Side JS: Sound & Vision APIs

Just like in the last topic where we looked at the data storage APIs of the browser, in this unit we focus upon a specific set of APIs for doing things with audio (sound) and with graphics (vision). Basically we can use the browser's sound and vision JS APIs to create awesome experiences for our users that go way beyond text presentation.

Be aware though that each of these topics is very large and could conceivably constitute an entire module individually, so what we'll study here is merely an overview and a taster of what can be done. Practically, the only limitation in sound and vision is your imagination, because the tools that the sound and vision APIs gives us are very powerful, ultimately enabling us to create raw sounds by manipulating frequency and waveform, and to create images by placing individual pixels.

We'll deal with each sub-topic individually, starting with sound and then moving on to vision.

10.1 Sound

Sound in the Web browser is handled through the Web AudioContext object. Really the web browser, your operating system, and your hardware are handling much of the task, acting to turn your instructions into perceivable reality. The AudioContext object is our interface to the sound related features that the browser provides, and also gives us a theoretical model, based on graphs, for creating, manipulating, and outputting sounds.

Our browser provides basic functionality for:

- Playing sound files (Audio Playback)
- Creating sounds from scratch (Audio Synthesis)

We'll look at each in turn, starting with playing back existing sound files, then looking at how to create sound signals from scratch by synthesising the sound waves that make up the sound.

10.2 Playing Audio Files

If weve already got an audio file, i.e. we don't need to synthesise any new audio then we can just play that audio as is. To enable this we can use the MP3, Ogg, and Wav formats which are supported as standard.

To play a file we merely use the HTML5 ¡audio¿ element which supports, by default, the MP3, Ogg, and Wav formats. Basically all we have to do to play an existing file in a web page is to include the audio element. This is just like how if we want to include a picture on a web page we include the ¡img¿ element. So audio gets treated as though it is a distinct entity within our web page. This also means that it is exposed via the DOM to JS, so we can manipulate our audio element, but more about that later.

The audio element has a variety of attributes that we can set and manipulate which give us, and our users, some control over the resulting audio playback. For example, the controls attribute gives our users a set of shuttle controls and other user interface elements for interacting with the audio. This can include playing and stopping the audio, and moving the current playback position within the file. We can also set other attributes such as autoplay and muted. Note that autoplay is frequently prohibited in many modern browsers, requiring some user interaction with the page before play begins. As a result you shouldn't rely on the ability to autoplay an audio file when your page loads as part of your user experience. However you can have audio playback autostart at page load time, but only if the sound is muted.

10.3 An <audio> example

Let's first look at how to include the audio element within a simple HTML web page.

```
<!DOCTYPE html>
<html>
    <body>
        <audio id="my_audio_control" autoplay loop controls>
            <source src="horn.mp3" type="audio/mpeg">
            Your browser does not support the audio element.
        </audio>
    </body>
</html>
```

Notice that we have set the autoplay, loop, and controls attributes. We tell the audio control what to play by providing a ¡source¿ element defining both the location of the file to play and its type. We also added a message to display if the browser that is viewing this page does not support the ¡audio¿ element. The result is shown in the following screen capture. You should experiment with different combinations of attributes to get a feeling for how this element works.

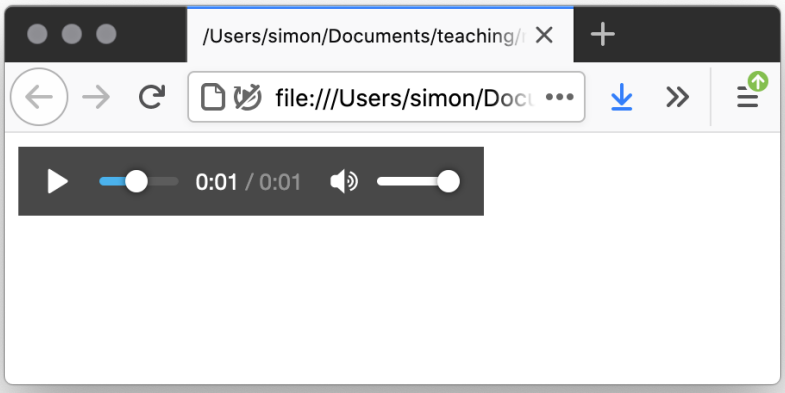


Figure 10.1

10.4 Interacting with <Audio> from JS

In the last example we relied upon our user to control the playback of our audio files. For many circumstances this is perfectly fine. When we combine this with the ability to use CSS to style the presentation of the audio element we can build user interfaces that naturally integrate the audio playback amongst our other features. However, the degree to which this can be controlled in a reliable way is currently uncertain. Many of the styling options are available through the webkit pseudo extensions and thus may be subject to change in the details of how they function.

However this does raise an interesting point. We can set the hidden attribute to hide the player, or else we can set the controls attribute to false from JS which also hides the player, in both cases causing the audio controls to be hidden. This effectively removes control of the audio from most users, but is problematic from a usability and user experience perspective. Without audio controls the user can't stop annoying audio from playing, or has to use the system sound settings to adjust sound levels, so what is the solution if we don't want to expose the standard audio controls to our users but do want to give them a good user experience? The answer, as with many aspects of web development, is to use HTML, and especially JS, to build alternative interface elements.

Let's look at a simple example of that now:

```
1 <!DOCTYPE html>
2 <html>
3     <body>
4         <audio id="my_audio" autoplay="autoplay" loop controls>
5             <source src="horn.mp3" type="audio/mpeg">
6             Your browser does not support the audio element.
7         </audio>
8         <button onclick="toggle_loop()" type= button">Toggle looping</
              button>
9         <script>
```

```
10              var ao = document.getElementById("my_audio");
11              function toggle_loop() {
12                  if (ao.loop) { ao.loop = false; }
13                  else { ao.loop = true; }
14                  ao.load();
15              }
16          </script>
17      </body>
18  </html>
```

In this example, weve merely added a button to control whether the audio loops, but the method is similar for controlling other aspects of the audio element. Although we've kept the standard user interface displayed, once we have a critical mass of functionality provided through our own user interface elements, at least so the user can start and stop audio playback, then we will be in a position to hide the default audio controls and offer only those which fit with our designs.

Interacting with other attributes through JS happens in a similar fashion. You can use the HTML standard[1] to investigate the remaining attributes and their legal values, but these include volume, loop, muted, controls, as well as methods such as play(), and pause().

Note that because the audio element is a child of the more general HTMLMediaElement, playback control methods are provided by that element rather than by the audio element itself, so we have to look to the media element API[2] for those parts.

You should now be able to build your own user interfaces to enable your users to control playback of existing audio files within your sites.

10.5 Audio Synthesis

So far we've only considered playing back existing audio from MP3 or similar media files using the ¡audio¿ element. However we have a much more powerful mechanism available to us for working with audio. Rather than playing back existing sounds e.g. from music files, we can create sound, from first principles, using JS. This means that instead of recording a real sound, a similar, or completely different sound can be artificially constructed by using the right values for the components of the sound. We'll refer to this process as audio synthesis.

For several decades synthesisers have been electronic musical instruments, frequently seen in piano keyboard form, but they do a similar function to what we will do using the JS AudioContext. Both are technically synthesisers, they allow us to create sounds by manipulating the components of sound, to get the effect that we want.

[1] https://developer.mozilla.org/en-US/docs/Web/API/HTMLAudioElement
[2] https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement

Note that the idea of "getting the effect that we want" should raise the idea in your mind that creating the audio for your site is as much about designing a user experience as any other part of web design, such as designing the visual, or interactive experience.

10.6 What is Sound?

Before we can get to grips with what AudioContext can do for us, we first need to turn our attention to the nature of sound. This is because AudioContext is built around a fundamental understanding of what sound is and how it works. By understanding this, we provide ourselves with a theoretical framework in which the operation of AudioContext makes more sense. Let's start with a working definition of sound:

```
Sound is a Movement (vibrations/oscillations) that travels (acoustic wave) through a m
```

The character of any given sound is affected by a lot of factors, but there are some fundamental features that have a great effect upon the resulting sound. These are:

- The shape of the wave. In audio synthesis we usually consider four core shapes, known as waveforms. These include the sine wave, the square wave, the triangle wave, and the sawtooth wave. We can also create custom waveforms by supplying sine and cosine values to a PeriodicWave node, but for our purposes we'll stick with the four builtins.
- We're only going to consider periodic waveforms, that is waveforms that repeat continuously until the sound stops. How often the repetition occurs is called the frequency of the wave. The frequency is usually defined in terms of the number of times, that is the number of cycles, that the wave repeats per second. Such measurements are made in Hertz (Hz) where 1Hz is defined as 1 cycle per second. Humans can generally hear sound that vibrates in the range between 20Hz and 20kHz, that is between twenty times per second and twenty thousand times per second. As we age our ability to hear at the extremes of these ranges, particularly at higher frequencies, is curtailed.
- Amplitude is a measure of how much any given wave changes during a cycle. Think of waves on a beach and consider the distance from the normal water level to the top of a wave (and also conversely the distance from normal water level to the bottom of the trough between waves). Amplitude for any given wave is a measure from the normal level, i.e. the midpoint between peaks and troughs, to the top, or bottom, of a wave. In audio synthesis we'll use amplitude as a proxy for loudness, but in reality loudness is really more complex and involves also the power (or intensity) of the sound. For our current purposes though, amplitude is a useful proxy for loudness.

We'll see that many aspects of sound depend upon this simple understanding of sound. A speaker, such as that on a radio, creates sound by turning a signal (which can be analogue or digital) in to a set of vibrations. This is achieved by moving at various frequencies and amplitudes, which in turn causes air to move, which our ears subsequently detect and our brains interpret. Conversely, a microphone is very similar to a speaker in inverse, a device that detects vibrations and returns them to a signal, initially analogue and then often converted into a digital audio signal.

Similarly, within a browser we can create audio by generating sound waves. A computer can control the frequencies and amplitudes (and other things) of multiple simultaneous sound waves to enable the resulting generated sound to make sense to us or otherwise fulfil the purpose that the sound is designed for. For example we might want to generate audio that sounds like something, that forms music, or is recognisable as a voice, or some other recognisable noise like thunder, or that we react to, like a siren.

10.7 The AudioContext & Browser Audio

Sound synthesis in the browser focuses on the construction of an audio-processing graph.

Remember: A graph is a mathematical model that is constructed from nodes that are connected to each other by edges. This basic idea is used in lots of ways throughout computing for modelling various ideas. Sound modelling is just one way to use graphs.

An AudioContext is just a graph that links audio processing modules (AudioNode) together. We create an AudioContext, which is an empty graph. We then add nodes to this graph to represent parts of our audio signal, from its creation, through any processing to change how it sounds, through to its destination, i.e. output via speakers to our ears.

Usually we would create a single AudioContext that we then reuse as needed in our site, e.g.

```
1 var audioCtx = new AudioContext();
```

In practice though, because of differences between browsers in how the audio API is implemented, we often need to use the following to create an AudioContext:

```
1 var audioCtx = new (window.AudioContext || window.webkitAudioContext)();
```

This becomes our handle to the audio context that we can manipulate and control via JS. An important property of the AudioContext object is the destination which is the place that sounds are output to (usually the computer speaker). We usually need to specify the destination to use once we've created an AudioContext otherwise we won't hear anything because there won't be an output.

We can create Audio Nodes and then connect them into our audio processing graph, represented by our AudioContext. These Audio Nodes create and shape the sounds we hear at the destination as the sounds pass through the graph. It can be useful to think of this process as a bit like sending water through pipes. Water can be diverted and made to run through various processes, perhaps UV treatment and particulate filtering, with the result being clean drinking water. Audio moving through an AudioContext graph is very similar, it can be filtered and manipulated so that what comes out at the destination is very different to what would be heard at the starting place. It is the filtering and manipulation process that we can use to our advantage to create the sounds that we want our users to hear.

10.8 Creating a Sound

Let's see how all of this comes together. We need to create an audio context graph, then add a sound source node to our graph, connect that sound source to a destination, and then get it to start sounding.

The starting place for sounds is an oscillator (these create vibrations so are the starting place for synthesising sounds). Remember back to our discussion of speakers earlier, the speaker's cone is an oscillator that responds to a signal, it then moves back and forth, it oscillates. Within our AudioContext, an oscillator is very similar, whilst it doesn't physically move, it digitally captures the idea of a starting place for our sound, something that moves backwards and forwards at a given frequency. How about the musical note A, that has a known frequency value of 440Hz. This is the default value for an oscillator but we're going to set it explicitly in our next example to show how to do that (and also to give you an opportunity to explore different frequencies).

You can type the following directly into your browser's web console (shortly we'll see how to embed this into a web page though as well):

```
1 var context = new (window.AudioContext || window.webkitAudioContext)();
2 var oscillator = context.createOscillator();
3 oscillator.frequency.value = 440;
4 oscillator.connect(context.destination);
5 oscillator.start();
```

Here we've create an AudioContext that we've assigned to the 'context' variable. We've then created an oscillator, which we've assigned to the 'oscillator' variable. We've then set the frequency value for the oscillator to 440Hz. To form our graph we need to connect our oscillator to something. As we don't need to do anything more to process our sound, we now merely connect our oscillator to our browser's default destination and the set it to start.

You'll notice that the sound doesn't stop once it's started. You'll need to refresh your page or close your tab to stop the sound. Just so you know how this relates to audio embedded in a web page, let's quickly use the exact same JS code within the context of an HTML webpage:

```
1  <!DOCTYPE html>
2  <html>
3      <body>
4          <script>
5              var context = new (window.AudioContext || window.
                   webkitAudioContext)();
6              var oscillator = context.createOscillator();
7              oscillator.connect(context.destination);
8              oscillator.start();
9          </script>
10     </body>
11 </html>
```

In this example we've literally wrapped our JS in pairs of ¡script¿, ¡body¿, and ¡HTML¿ tags to form a web page around it. That's all we need to embed our audio graph in a page.

10.9 Adding User Interaction

That last example showed us the simplest way to create a sound and output it to our speakers. It's quite annoying though as we have to close the tab or turn down the speakers to stop the sound. What we really need is a 'mute' button, which is a perfect opportunity to show how to do user interaction with our AudioContext.

```
<!DOCTYPE html>
<html>
    <body>
        <button onclick="mute()" type="button">MUTE</button>
        <script>
            var context = new (window.AudioContext || window.
               webkitAudioContext)();
            var gain = context.createGain();
            gain.connect(context.destination);
            gain.gain.setValueAtTime(0, context.currentTime);
            var oscillator = context.createOscillator();
            oscillator.connect(gain);
            oscillator.start();

            function mute() {
                if(gain.gain.value == 0) { gain.gain.setValueAtTime(1,
                   context.currentTime); }
                else {gain.gain.setValueAtTime(0, context.currentTime)}
            }
        </script>
    </body>
</html>
```

Instead of an empty web page, this time we've added a button with an onclick method defined that calls our JS. The function that we want to run when we click the button is our mute() function. This is simply an if statement that checks the current status of the gain value. We'll consider gain to be a proxy for volume. If the gain is zero then the sound is already muted so we turn it to maximum, and if it is anything other than zero then we set it to zero. Gain can have values between zero and one, so you should already be getting some ideas for how to use other HTML interface controls to give variable control over the volume of your sound. You might even be starting to think similar ideas about the frequency values. Don't limit yourself to just the HTML elements though, a nice user experience might include using keyboard, mouse, touchpad, or multi-touch to enable fine, variable control. Note that for those of you who are really inspired, there is also a draft specification for Web Midi which can enable you to plug MIDI enabled instruments into your computer and have them interact with web audio in the browser.

10.10 A Clarification on Node reuse

One misconception that affects nearly everyone when starting to use the AudioContext API is around AudioNodes and reuse. We think of AudioNodes as a tone that we can stop and restart as needed so we create a new node, play it, stop it, and then go to restart it and get an error.

AudioNodes are a little counter-intuitive. They are designed to be very lightweight and low-cost to create. So we use them once and then discard them. A useful way to think of AudioNodes is like a note played on a piano. Once youve hit the key and the notes played, you need to hit the key again to get another note. You don't start, and stop, and restart piano notes, you just create new ones as often as needed to get you through a song.

So, an AudioNode, such as an Oscillator, can only be started and stopped once. We dont, and indeed can't, start and stop then restart them. Instead we have to create a new AudioNode when we need one as they are very cheap to create. Note that one alternative approach is to have nodes for each required note or tone running continuously and to mute and unmute them as required. This can work, but is a bit of a hack and requires more effort to get the sound just right. Muting and unmuting at the wrong point in a wave's cycle can cause audible artefacts that sounds like clicks and pops as a result.

10.11 Playing chiptunes

Now let's look at a much longer example. This will bring together everything that we've looked at so far, as well as introducing some other useful functions like setInterval() and setTimout(). Neither of these are audio specific but can be used to do useful functionalities, firstly to do a task repetitively to a given schedule, i.e. after a given interval, and to do something for a given length of time, i.e. timeout after a set number of milliseconds.

The following will play a song that some of you might find quite recognisable (especially if you play videogames):

```
1  <!DOCTYPE html>
2  <html>
3      <body>
4          <script>
5              window.onload = function() {
6                  var audio = new (window.AudioContext || window.
                       webkitAudioContext)();
7                  position = 0,
8                  scale = { b: 233, c: 261, d: 293, e: 329, f: 349, g:
                       391, A: 440, B: 493, C: 523, D: 587, E: 659, F: 698,
                       G: 783, a: 880 },
9                  song = "EE-E-CE-G---g--C-g--e--A-B-BA-gEGaFG-E-CDB--C--g
                       --eA-B-BA-gEGaFG-E-CDB";
10             setInterval(play, 250);
11
12             function createOscillator(freq) {
13                 var attack = 10, decay = 250, osc = audio.
                       createOscillator();
```

```
14                  osc.frequency.value = freq;
15                  osc.type = "square";
16                  osc.connect(audio.destination);
17                  osc.start(0);
18
19                  setTimeout(function() {
20                      osc.stop(0);
21                      osc.disconnect(audio.destination);
22                  }, decay)
23              }
24
25              function play() {
26                  var note = song.charAt(position), freq = scale[note];
27                  position += 1;
28                  if(position >= song.length) { position = 0; }
29                  if(freq) { createOscillator(freq); }
30              }
31          };
32          </script>
33      </body>
34  </html>
```

There are several things we need to talk through here. We have some setup code, and then we have two functions. Let's start with setup. We create our AudioContext, then we create a scale, which is just a mapping of letters to frequencies, so if we write A then we mean 440Hz.This helps us to write descriptions of songs using musical note letters instead of specifying the frequency each time. We immediately use that mapping in our song which we describe as a series of letters (using the '-' character to give us a none-playing gap between notes when needed. Remember that many tunes are as much about when a note is played as they are about the silence between those notes). For completion, we've also defined a position variable, starting at zero, to track where we are in the playback of our song. Each time we play a note we need to increment the position so we know which note to play next.

We then have our two functions, createOscillator() and play() which control actually making our sound. We'll start with createOscillator() which merely plays a specified note for a given length of time. Finally we have the play() function which is called every 250ms by the setInterval function. This essentially means that we want to play a new note every 250ms. If there is a '-' then the if(freq) check fails and a new createOscillator() call doesn't happen which has the effect of playing 250ms of silence when needed. With play() we look up the note for the current position in the song, look up that note's frequency, then pass the frequency to the createOscillator() function. Notice how setInteval is useful for creating and playing a new note on a set schedule and conversely, notice how setTimeout is useful for limiting how long each note plays for.

10.12 Sound (Audio Context) API Overview

The Web Sound API does a lot more than we've covered here so it's well worth investigating the W3C specification: https://webaudio.github.io/web-audio-api/

Generally, when dealing with web audio, think in terms of an instance of a graph, the AudioContext, that is formed from the following:

- Audio Sources
- Audio Effects and Filters
- Destinations

Note that there are also audio data analysis and visualisation tools, audio channel splitting and merging tools, spatialization tools, as wells as support for offline and background audio processing. For now though, we'll now briefly consider each of sources, effects/filters, and destinations, in turn...

10.13 Audio Sources

There are a variety of audio sources that are supported in AudioContext graphics. These include:

- OscillatorNode The most common source for audio. A node we can start, stop and set the frequency (Hz) and type (sine, square, sawtooth, triangle, etc.)
- AudioBuffer - an audio asset that is held in memory but has been sourced from elsewhere, e.g. read in from a file.
- HTML5 audio or video elements (MediaElementAudioSourceNode)
- Media streams (MediaStreamAudioSourceNode)

10.14 Audio Effects/Filters

Once you have an audio source set up as you want it, the next thing to do, if necessary, is to process the audio coming from that source. In synthesis this "processing" is usually achieved through the application of audio filters. Think of them a bit like a water filter or coffee filter, they will let some parts of the input through to the output and capture or discard other parts. They might also do more than merely selectively pass the input audio through to the output and might even modify the input on the fly. In audio synthesis there are standard filters that are applied when working with audio to achieve known effects. The AudioContext graph supports a variety of these known filters such as:

- Biquad Filters - these are used to represent a range of audio filters for tone control and graphic equalization, e.g. low-pass, high-pass, band-pass, high/low-shelf, peak, notch, and all-pass are standard biquad filters that give different end results.
- Convolvers - for adding reverb (short for reverberation) to your audio. This is to get the effect of your soundwaves bouncing off of and being absorbed by various surfaces. Think of how the same sound can be different in various places, for example your shoes will sound different in a carpeted room compared with a wooden floored hallway, or your singing voice will be different in your garden compared with in your shower. This is reverb in action.

- Delay - refers to the time between arrival of the audio signal from the source and its onward propagation to the next node in the graph. Delay is used artistically in music production to create echo effects but also in audio production, for example to enable a signal to be sent to multiple speakers across a large space so that a signal arrives at each speaker at precisely the same time.
- Compressors – used to lower the volume of the loudest parts of a signal to prevent clipping and other distortion that might occur.
- Gain - used to increase or decrease the volume of an audio signal.

10.15 Audio Destinations

Used to specify where your audio should go once youve created and processed it. Usually this is the default which is the users speakers (AudioDestinationNode). But an AudioDestination can include other outputs, for example,

- where there are multiple audio output channels and you want to direct the output of an audio graph to a specific channel,
- if the audio should be stream elsewhere (MediaStreamAudioDestinationNode).

10.16 Howler.js

Finally, remember that the AudioContext API is quite low level. We don't always want to deal with audio from first principles, so there are libraries that we can use that build on AudioContext and bundle up a range of pre-packaged functions so that we don't have to rewrite them from scratch.

So instead of creating sound from basic waveforms and frequencies we can take a more intermediate approach by adding in a library. Howler.js[3] is an example of one such audio library for the Web. It gives you a simplified API for working with audio from JS. It is lightweight, around 7KB gzipped, and is 100% JS. You can use it for playing music, creating radio streams, doing spatial audio where sounds appear to come from specific locations, and for audio sprites.

10.17 Vision

Now let's consider a different one of our senses besides hearing. Let's consider satisfying our sense of sight. In this context, by vision, we mean creating graphics or making pretty pictures on the computer. That is, rather than merely displaying an existing picture, as we might already have done using the HTML ¡img¿ element, instead we might want to be able to write code whose outputs is the image itself.

In this section we will specifically consider 2D graphics. Although the browser also supports 3D graphics, that might be one step too far for this module, and is left as an exercise for your own self-study if you're interested.

[3] `https://github.com/goldfire/howler.js`

Producing 2D graphics generally involves drawing pixels of different colours onto the screen in specific places to create the desired visual effect. If we draw pixels differently over time then we create a form of animation (but again, producing animation is left as a path to explore yourself).

Remember that a pixel is a "picture element", and that our computer screens are made up of thousands of pixels. Generally we consider our screen to be a flat, two-dimensional space onto which our graphics are drawn. Each pixel in this space is labelled along the x and y axes and we consider the top left corner to the origin, or (0,0) position. Each axis then extends positively from 0 up to a maximum value that is equivalent to the resolution of the screen that you are using. There are some practical caveats here of course as people don't always have their browser window maximised so the maximum resolution of the screen isn't necessarily all of the space that we have to draw into.

10.18 HTML Canvas

The spaces that we actually draw into is a sub-part of our page, actually a space defined by an HTML4 element called a canvas. Just like a painters canvas, the HTML canvas element is the place onto which we draw our graphical elements. It can help to think of many of the drawing functions that we see later in the context of applying paint with a brush to a canvas. We apply paint using brush strokes, i.e. moving the bristles of the brush from place to place. Usually we choose exactly where we want a stroke to begin and end then we touch the brush to the canvas at the start of the stroke and raise it as the end of the stroke. It is a similar situation with some of the drawing functions which require us to decide where to start and where to stop our stroke. We also need to specify colour as well as the weight of the stroke as well as many other characteristics

We must add a canvas to our web pages, either into the HTML directly or through JS, in order to be able to draw a picture on our web pages. We then use a graphics rendering context, which is kind of analogous to an AudioContext, to actually make marks onto the HTML ¡canvas¿ element.

Now let's see a ¡canvas¿ element in situ on a webpage:

```
<!DOCTYPE HTML>
<html>
   <body>
      <canvas id = "my_canvas" width = "100" height = "100"></canvas>
   </body>
</html>
```

This merely defines a new canvas, gives it an ID of "my_canvas", and then specifies a size of 100 by 100 pixels. However if you load it up in your browser, you won't actually see any difference in your page, although if you add other HTML elements around it then they will be displaced by blank canvas element. So let's add a bit of style to draw a box around our canvas so we can see it:

```
<!DOCTYPE HTML>
<html>
```

```
3     <head>
4        <style>
5           #my_canvas{border:1px solid tomato;}
6        </style>
7     </head>
8
9     <body>
10       <canvas id = "my_canvas" width = "100" height = "100"></canvas>
11    </body>
12 </html>
```

Exactly as before but now with a 1 pixel tomato coloured border around our canvas. It should look something like this:

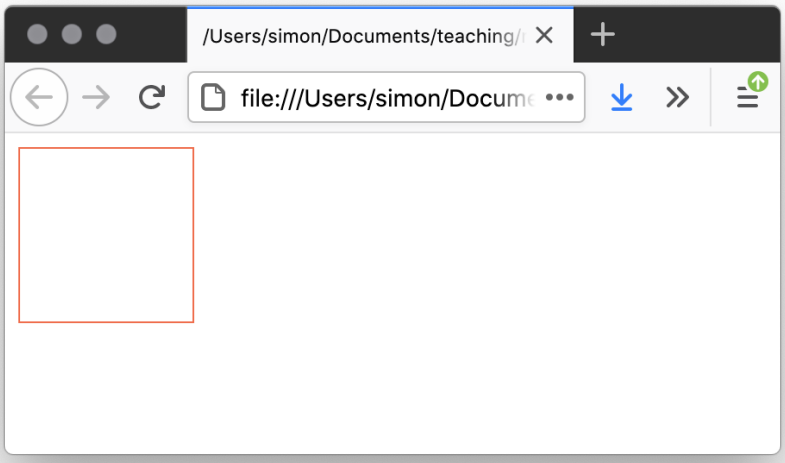


Figure 10.2

10.19 Drawing on our canvas

With that basic setup we can now add graphics to our page by drawing on our canvas. We need a 2D context and an alternative in case ¡canvas¿ isnt supported:

```
1  <!DOCTYPE HTML>
2  <html>
3     <head>
4        <style>
5           #my_canvas{border:1px solid tomato;}
6        </style>
7     </head>
8
9     <body>
10       <canvas id = "my_canvas" width = "100" height = "100"></canvas>
11       <script>
12         var canvas  = document.getElementById("my_canvas");
13         if (canvas.getContext) {
```

```
14              var ctx = canvas.getContext('2d');
15              // Add you drawing code here...
16          } else {
17              // Do something else if user's browser doesn't
18              // support the canvas element, i.e. retrieve
19              // an image file as a replacement or a message
20          }
21      </script>
22    </body>
23 </html>
```

Notice the script section which does three important things:

1. Retrieves a handle for the canvas using getElementyId()
2. Creates a 2D drawing context which we can use to actually draw onto the canvas
3. Outputs a message for the user if they are on a browser which doesn't support the canvas element (or else let's you perform an alternative function like retrieve a static image file from a server)

Our page still looks the same, like this:

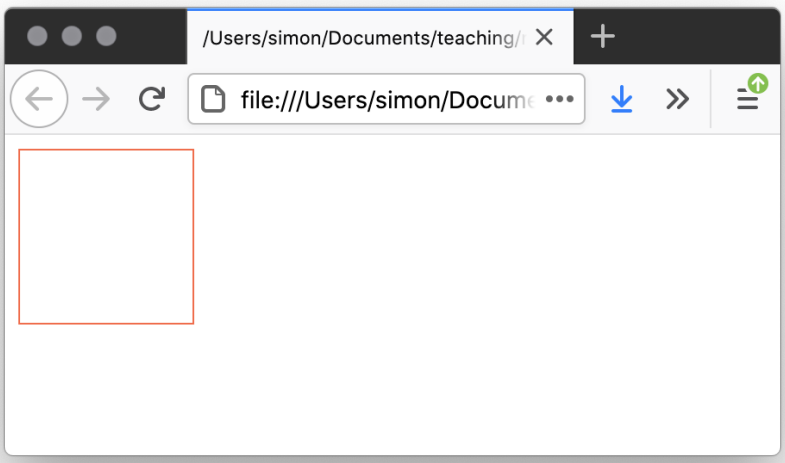


Figure 10.3

This is because we haven't yet actually drawn anything onto the canvas. We'll next look at three basic drawing functions we can use:

- Drawing Rectangles
- Drawing Circles
- Drawing Lines

10.20 Drawing Rectangles

Drawing a rectangle is as simple as giving the starting and finishing coordinates for the upper left (starting) and lower right (finishing) corners, for example:

```
<!DOCTYPE HTML>
<html>
   <head>
      <style>
         #my_canvas{border:1px solid tomato;}
      </style>
   </head>
   <body>
      <canvas id = "my_canvas" width=360 height=240 ></canvas>
      <script>
        var canvas  = document.getElementById("my_canvas");
        var ctx = canvas.getContext('2d');
        ctx.fillStyle = "tomato";
        ctx.fillRect(25,25,100,100);
      </script>
   </body>
</html>
```

Notice that in addition to creating a filled rectangle using the fillRect() function we also set the fillStyle by setting a colour. We've also simplified the code a little for this demonstration by removing the checks on whether the context could be created. In the real world you would do these checks just in case your user had a browser that didn't support the canvas element but in order to give clarity to our examples this has now been omitted. You should expect to see something like the following as a result:

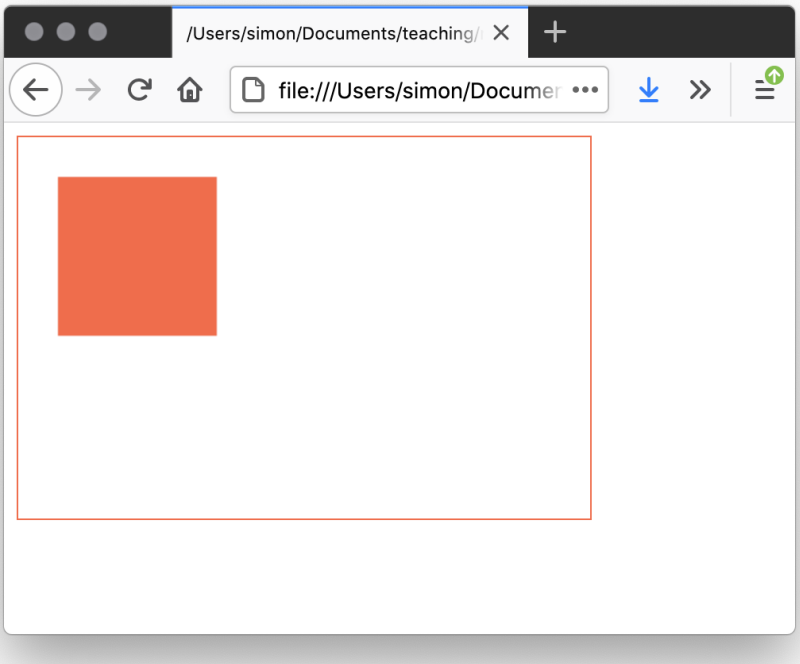


Figure 10.4

10.21 Drawing Circles

Drawing circles is similar to, but slightly more complex than, drawing rectangles. This is because we are actually drawing curves, or arcs, which just happen to begin and end in the same place. In the following example we've omitted the HTML boilerplate and just presented the JS code but it should drop straight into the HTML structure we previously built.

```
<script>
        var canvas  = document.getElementById("my_canvas");
        var ctx = canvas.getContext('2d');
        ctx.strokeStyle = "green";
        ctx.lineWidth = 1
        ctx.beginPath();
        ctx.arc(75,75,50,0,Math.PI*2,true);
        ctx.moveTo(110,75);
        ctx.arc(75,75,35,0,Math.PI,false);
        ctx.moveTo(65,65);
        ctx.arc(60,65,5,0,Math.PI*2,true);
        ctx.moveTo(95,65);
        ctx.arc(90,65,5,0,Math.PI*2,true);
        ctx.stroke();
</script>
```

Notice here that we used the strokeStyle instead of fillStyle to set the colour. We've also specified a lineWidth or 1 pixel. Then we indicate that we want to create a path, the

specific movement of our brush across the canvas. After that we use a series of arcs and movements (moveTo) which builds a sequence of brush strokes on the canvas, basically defining the path that we started. What is happening is that we are making a mark on the canvas by using the arc() function then lifting the brush and moving it to another location to make another mark, again using the arc() function, and so on. After we have defined all of the arcs that we want to draw then we use the stroke() function to actually make the marks onto the canvas.

Note that drawing a single circle onto our canvas is actually as simple as this:

```
var c = document.getElementById("my_canvas");
var ctx = c.getContext("2d");
ctx.beginPath();
ctx.arc(100, 75, 50, 0, 2 * Math.PI, false);
ctx.stroke();
```

However our more complex example yields a slightly more interesting result:

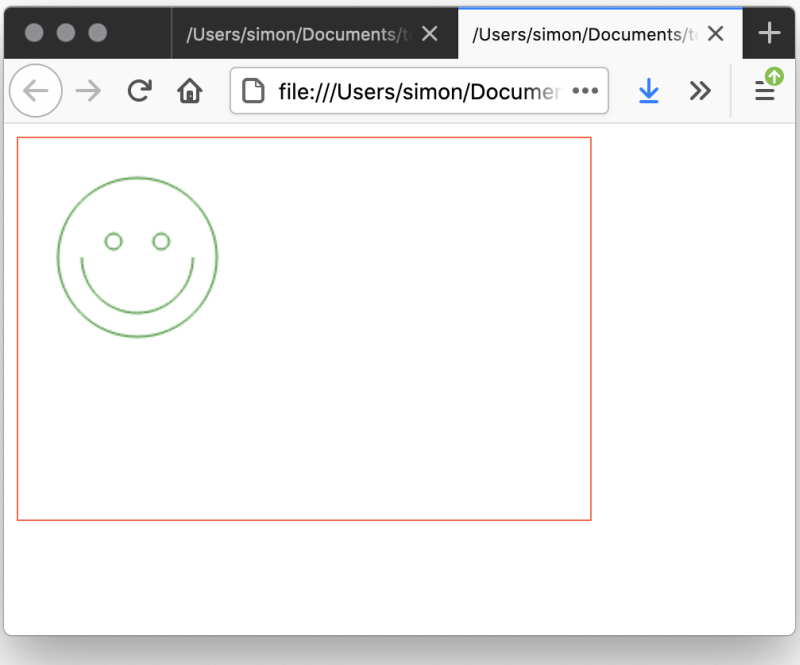


Figure 10.5

The arc() function parameters are as follows:

```
context.arc(x,y,r,sAngle,eAngle,counterclockwise)
\end{lstlisting
}
\paragraph{} where x and y are the coordinates for the center of our
   circle, r is the radius, sAngle and rAngle are the starting and
   finishing angles, and counterclockwise indicates the direction in
   which to draw the arc. The trickiest aspect of using arcs is working
```

```
    out the start and finish angles relative to the centre. These are
    measured in radians so start at 0 and grow to 2 * pi.
5
6
7 \section{Drawing Lines}
8 \paragraph{} Finally, let's consider how to draw lines. Again the tricky
     part is deciding where you want your lines to start and end and
    determining the correct corresponding coordinates for the start and
    end points of our stroke.
9
10 \begin{lstlisting}
11       <script>
12         var canvas  = document.getElementById("my_canvas");
13         var ctx = canvas.getContext('2d');
14         ctx.strokeStyle = "green";
15         ctx.lineWidth = 1;
16         for (i=0;i<10;i++){
17             ctx.lineWidth = 1+i;
18             ctx.beginPath();
19             ctx.moveTo(5+i*14,5);
20             ctx.lineTo(5+i*14,140);
21             ctx.stroke();
22         }
23         </script>
```

This time we've include a little bit of additional JS to draw a series of lines using a for loop, so that each iteration through the loop creates a new line of increasing size. Again we're creating a path that is defined using a sequence of moveTo() and lineTo() function calls before finally calling stroke() to actually make our marks on the canvas. This is a little different to the arcs before in that we are using moveTo to specify the starting points then lineTo to move from the starting point of our line to the end point. Within both lineTo and moveTo we are merely specifying an x and y coordinate for a location on the canvas. We should see this result:

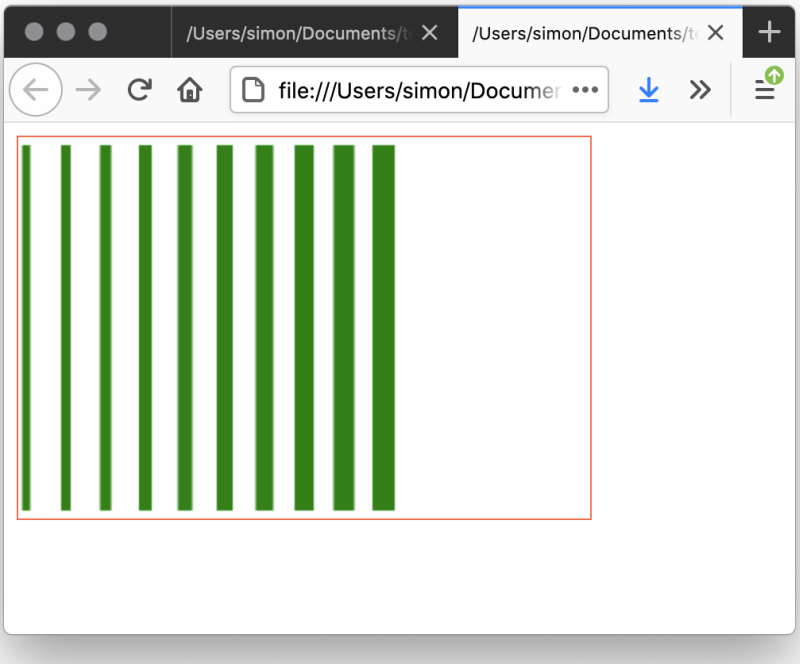


Figure 10.6

10.22 Summary

So far in this topic we've considered:

- How to play existing audio files on our pages
- How to create sounds from scratch using the AudioContext, and
- How to create 2D graphics from scratch using the HTML canvas.

There is an awful lot more to both of these topics, sound and vision, each of which could be covered in an entire module, or even an entire degree programme. As a result, this unit should be considered as merely the starting point and the briefest of introductions and overviews. However, it should be sufficient to help you understand how both sound and vision fit into the context of a web document and start giving you ideas for how they can be exploited.

Chapter 11

Deployment

Finally, we can now round out our study of web technologies by re-considering the Web in the context of all of the topics that we've studied so far. Although you will already, I hope, have deployed the sites that you've already built during the module, we're now in a position to consider the idea of deployment in more detail.

We'll now look at the various options we have for deploying our sites and consider how some options give us opportunities to move functionality to the "server side". Whilst we haven't studied server-side development in this module, I want to leave you with some directions for future paths to explore in terms of server-side development so that you can continue your web technology journey.

11.1 Making Our Sites Available to Others

At some point during the web development process we are probably going to want other people to be able to view and use our sites. We are going to want to publish our site to the Web. This is the process of moving from local development, where the site is running entirely on our local machine, to a hosted web deployment, where the site is running entirely on a web server that is accessible on the Web. As a quick aside, by accessible on the Web, we mean that anyone with a browser and an Internet connection can type in the web address and retrieve the pages that make up your site.

To better understand what is happening when we deploy (or publish) our websites we need to consider servers and clients, and how these relate to web servers, web clients and the HTTP protocol that facilitates communication between web servers and web clients.

On a more practical note we'll also consider some options for deploying our websites which range from free (like GitHub Pages or NeoCities) through to paid hosting of various kinds.

Let's start though by considering the difference between static and dynamic sites as the distinction will affect the remainder of the unit.

11.2 Static Sites Versus Dynamic Sites

We should also, before we go any further, distinguish static sites, or more strictly, static hosting, from dynamic sites and dynamic hosting. Usually we would consider the kind of development that we've done during this module as static site development. This is because the server to which our pages are deployed doesn't have to manipulate them in any way before returning them to any client that requests them. The pages are statically hosted hence the site is referred to as a static site. "But what about our JavaScript? Isn't that dynamic?" I hear you ask. Yes, the site itself may well be dynamic and support any amount of user interaction but if that is entirely handled within the browser, on the client side, then it is still considered a static site. The notion of a static site is related to whether the server must do any work itself to generate the pages that are returned to the browser. As a result then a dynamically hosted site is one where the web server itself either generates, or alters, the pages that are served to the user, or else delegates those tasks to some other software. If any work is done on the web server beyond merely returning the pages to the requesting client then the site is deemed to be dynamic as a result. Note that a dynamically hosted site can still serve a web site which includes no actual user-facing interaction, e.g. pure HTML and CSS.

11.3 Local Web Development

So far, we've mostly done local development. Although if you've deployed your site to Github Pages then you've also done one type of web deployment. However, we've not really considered in much detail what this actually means.

Schematically, local development for use has been something like this:

1. Create text files (Saved as .html, .css, .js)
2. Load text files directly into the browser (for example by double-clicking on the index.html)
3. Interact with the site in the browser

Most of our site should work just fine under these circumstances, and the basic set up is illustrated in the following Figure in which the browser requests a file (web document) from the disk (local hard-drive) which is returned to the browser for rendering as a web page.

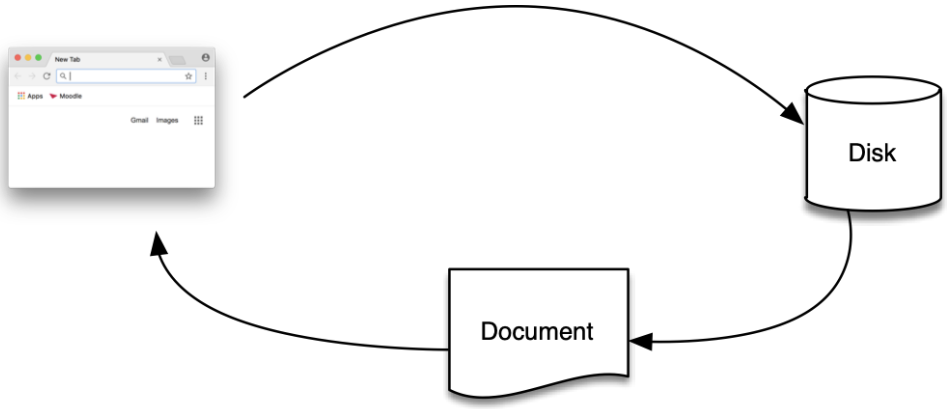


Figure 11.1

This is definitely using web technologies, but the resulting site is not part of the Web. This is because the site hasn't yet been deployed or published.

11.4 Deployment

In order for our site to be available on the web, we need to store our web pages where they are accessible to our users. For our sites to be accessible we mean that a client (other than ourselves on our local machine) can connect to the storage location and request specific pages and "use the site". Note that there are other meanings of the term "accessible", usually in the context of the accessibility of the site, how usable the site is for different categories of user, but we'll leave that aspect to one side for now. The accessible location is known as a server, or more strictly, a web server. Note again, that the term "server" is overloaded terminology: server can refer to a hardware server, or a software server but can also refer to different types of software, for example a web server or a database server amongst many other types of software server. For our purposes, when we refer to a server in this unit we are referring to a web server.

We could install web server software on our local machine and then go through the process of exposing that web server to the wider world but that raises many issues about security and requires lots of administrative skills that we haven't considered so far in this module and which are a bit out of scope. That said, you should definitely, at some point, consider trying to deploy a web server so that you understand the process of turning a regular computer into a web server. For the rest of this unit though we are going to consider using third party services and hosting providers.

11.5 Server Hardware

A hardware server is a computer that is connected to a network, usually the Internet and which is running server software. There are many kinds of server software but we're only concerned with web server software for now. The hardware server accepts connections from other Internet users (often called clients) and then performs the tasks specific to the kind of server software it is running.

Hardware servers can be basically any computer, from a Raspberry Pi right through to huge clusters, mainframes, and supercomputers, and including nearly everything in-between. Note also that a Raspberry Pi is not the minimal computer required to run a web server. Many Internet of Things (IoT) devices run tiny web server software to provide user interfaces to their owners for configuration.

11.6 Server Software

Server software runs on a computer. Conveniently, the act of running server software turns the hosting hardware into a hardware server by extension. Server software listens for incoming connections and then responds appropriately to those connections.

For the Web, server software is usually, primarily, HTTP server software. This is merely a piece of software that implements the HTTP protocol so that:

1. It can listen for incoming HTTP requests, and can
2. Respond appropriately to those requests by sending responses.

There are lots of implementations of HTTP Server Software, including:

- NGinX,
- Apache,
- IIS,
- Lighttpd ("lighty")

Installation and maintenance of HTTP/Web server software is usually the domain of system administrators rather than developers but you should still have some awareness of what happens after your site is designed and built, the process of making your site available on the Web.

As a rule, an HTTP server needs to be connected to the Internet and have a valid, reachable IP address. The server listens for connections. The default port for web connections is port 80. This is so common that browsers connect automatically to port 80 when you type in a web address. Web servers can also listen on any other valid port from the range of 65536 available ports on each hardware server. Many of these ports though are generally reserved for services other than the Web. A secondary standard web port is port 8080 which you might occasionally see in the wild, usually because it is specified in the web address of a given site.

11.7 Clients & Servers

The "Client/Server Architecture" is a long established approach for organising communication between different devices and also between different pieces of software. Client/Server Architectures are used in lots of places in computing but well assume from now on that we are dealing solely with HTTP servers.

With an HTTP server, a client makes a request to a specific server (using a web address to identify which one). The server is already listening for requests. When a request is received the server determines what to do according to the HTTP protocol which specifies the allowed responses. The server then sends an appropriate message back to the client. That constitutes a single request-response cycle. The client can subsequently initiate additional communications by sending new requests.

Note that this relationship of client initiating a request and the server responding is fairly standard, unless youre using the XWindow system in which case the relationship is back to front.

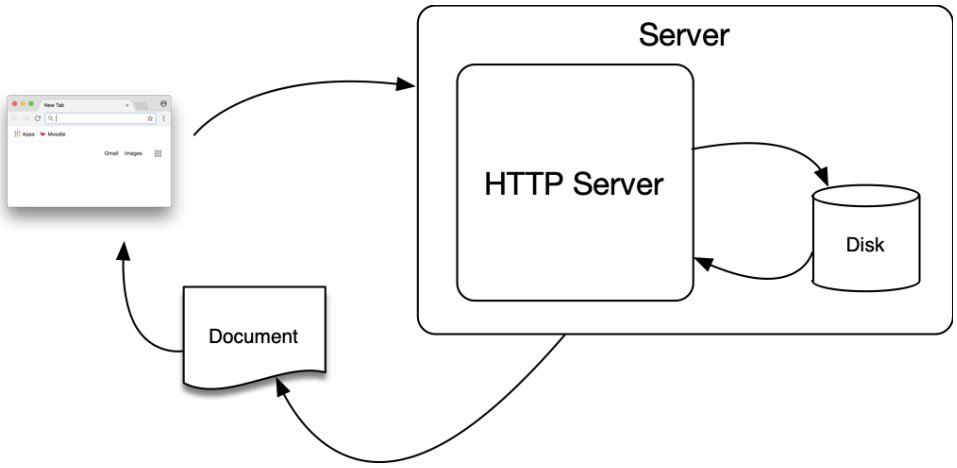


Figure 11.2

The server is a nexus. Multiple clients can connect to it with their own individual requests and the server will respond appropriately within the rules set down by the communication protocol.

11.8 HyperText Transfer Protocol (HTTP)

The HyperText Transfer Protocol (HTTP) is a formal definition of how web servers and web clients can interact. It defines how (primarily text) messages for transferring HyperText should be:

- formatted and transmitted
- responded to by servers and clients upon receipt of a given message

HTTP is a core standard of the Web along with HTML. The relationship between the two is straightforward, HTML defines how to structure and inter-link web documents and HTTP defines how to move those web documents around so that they can be read and their links navigated.

Technically, HTTP is a stateless, application layer protocol for communicating information between distributed systems.

11.9 Deploying our own sites

We have a number of options when it comes to deploying our web sites. We can:

- Run a Local web server on our development machine.
- Use a free static hosting service.
- Use a Dedicated Web hosting service
- Use a Virtual Server
- Use a Hardware Server
- Use a Cloud Host

We'll now consider each in turn...

11.9.1 Local Web Server

This is not really deployment but can be useful when developing a web site. It means that we can get our site to act during development as if it was publicly hosted. This can occasionally help us to identify some bugs that are only obvious after deployment. Note that there are also browser functionality limitations as a result of the browser security model which limits what a browser can do when reading files from the local file system, so running a local web server is useful when you run into these limitations.

To run a local web server you should choose some appropriate software. NginX is a good choice, install and run NGinX (or another web server) locally on your development machine then connect to that local web server using the localhost or 127.0.0.1 address, accessing whichever port was set up by default for your installed web server (usually port 80 or 8080 but can be other ports).

This is good for testing your sites and helps answer questions like: does my site behave the way it is supposed to when served up by a server? Do the links on my site work correctly?

A local web server can be made available to other Internet users (but this gets complicated fast and is outside of the scope of this module).

11.9.2 Free Static Hosting

There are many free static hosting services out there on the Web. These are sites that provide you with free web space for hosting and serving reasonably simple sites. These usually restrict you to plain HTML, CSS, and JS pages and have very limited, if any, server side dynamic functionality.

Github Pages[1] is one such hosting service, albeit really oriented towards the needs of the open-source software community. This includes static hosting direct from your GitHub repository. When you push updates to your Git repository in GitHub the changes to your site are automatically deployed and made available to all web users. We've already seen how to deploy to GitHub pages during earlier lab activities. For more information visit:

NeoCities[2] is another free hosting service where you can deploy your static sites. This is an "old School" style of free, static site hosting which has been designed to be reminiscent of the Web of the 1990s. It is named for an older web host service called Geocities (which is, incidentally, where I published my own first ever website back in 1998). For more information visit:

There are many free static hosting providers. Both GitHub Pages and Neocities have been around for a while but the hosting eco-system is very dynamic and you should investigate what is currently available should you want to try something different. You should also be aware that many of the commercial hosting providers also offer free-tiers amongst their services which can give you similar free, high-quality hosting for smaller sites.

11.10 Dedicated Web Hosting

Many sites provide web hosting for a price. Often such hosters provide additional features and guarantees which can be useful if the site that you are deploying is important.

The process is generally as follows:

- Sign up for an account,
- Pay the fees (start small/sometimes free) escalate rapidly depending upon bandwidth/features
- Upload your pages

There is often some limited control over the specific web server software and frequently some additional choices of pre-packaged web tools for deploying Wordpress, Joomla, &c. Although out of scope for our purposes right now, these providers can also sometimes offer limited server-side dynamic hosting as well.

Amongst the providers in this sphere are 1&1 Ionos, RackSpace, Fasthosts, and many, many more

[1] `https://pages.github.com/`
[2] `https://neocities.org/`

11.10.1 Virtual Server

This option gives you a complete virtual machine on someone elses hardware server. This is connected to the Internet and you will pay for CPU, RAM, Storage, and bandwidth, according to your needs. This can start very cheap. There are some free virtual server hosts which offer their lower priced virtual servers from 2.99/month but prices can escalate very rapidly.

A virtual server is basically a complete remote computer and you will often get a basic install of a Linux machine (although some hosts support other operating systems). You get almost complete control over your machine, but now have to assume much more of the administration of your machine and its operating system. You will usually start with a basic Linux machine and must build up the software side of your server from there, installing all software, languages, web servers, databases, etc. as needed.

It is usually easy to backup and move your server to other places and gives you flexibility, perhaps to keep an "in house" clone of your virtual server locally for development and offline testing.

Again there are plenty of providers available and the market is quite dynamic. Currently providers include the following: Fasthosts, 1&1 Ionos, Digital Ocean, Linode, and OVH.

A virtual server is a good and useful middle ground for more dynamic sites or ones where you need to exert more control over the way that you site is served and administered. You can easily scale up by purchasing a virtual server from a higher tier e.g. more RAM, CPU, Storage, Bandwidth, etc depending upon your needs. This is often possible with minimal effort and simple reboot.

11.10.2 Hardware Server

One of the more expensive, but perhaps most flexible solutions for web hosting is your own hardware server. This is an actual physical machine, located in your own premises or in a hardware hosting facility. This is Hardware that you build, buy or rent. It must be connected to the Internet and this connection needs to be through a reliable network with a dedicated IP address.

All administration must be done by you or others in your team. You must still build up the server software installation just like with a virtual server. You must also consider back-up and have a plan for expansion if your server can't keep up with the popularity of your site.

11.10.3 Cloud Host

Cloud Hosting is sometimes a fancy name for a virtual server but can also refer to the idea of computing as a service and the provision of "Web services". Again, there are many options for web service based cloud hosts, Amazon Web Services and Google Cloud Computing are two of the biggest providers.

Cloud hosting is scalable to the degree that you can afford and offers great flexibility but, especially at the more expensive end, the balance of advantages and disadvantages between a cloud hosted environment and a hardware setup can change and it can become cost-effective to manage your own infrastructure once more.

Ultimately though, you should remember that a cloud machine is basically, someone elses computer, with the attendant security and privacy issues associated with that. Note that this also affects all types of hosting. Before choosing a host, and definitely before deploying your site you should be considering where the server is located and also the associated legal jurisdiction. Does this have any effect on the type of site you intend to deploy? Whilst the Web is global, the machines that provide the functionality for the Web are still physically located somewhere in the world and are thus beholden to the laws that pertain in that location.

11.11 Web Standards

Now let's take a final detour to consider web standards. These are the publicly agreed documents that underpin the Web. They define the protocols, like HTTP, and they define the languages, like HTML5, and CSS3, of the Web.

Part of the success of the web is due to the goal of enabling non-experts to publish to the web. Clients and servers are quite relaxed about what they will accept, for example, HTML can be quite malformed before it becomes un-displayable. Usually we will still get the text content displayed as the most basic fall back. A major part of the success of the Web stems strongly from the existence of basic agreements on how all the parts work together. There are written agreements (standards) that describe these agreements. There are bodies (groups of people and organisations) that engage in multi-year negotiations to design and document standards for how the Web works. The role of these bodies is usually to develop new standards (to govern new circumstances) or enhancing existing standards (because we rarely get anything perfect the first time around).

There are three important bodies involved in standardising the web. These are:

- The Internet Engineering Taskforce (IETF)
- The World Wide Web Consortium (W3C)
- The Web Hypertext Application Technology Working Group (WHATWG)

11.12 The Internet Engineering Task Force (IETF)

The Internet Engineering Task Force (IETF) is an organisation who develop and promote open standards for the Internet. Whilst this isn't focused specifically on the Web, the Internet underpins most of the functionality of the Web, so what happens with respect to Internet protocols is of huge importance.

The IETF is primarily known for the TCP and IP standards. These are:

- TCP - The Transmission Control Protocol. A protocol for transmitting packets of data between computers.
- IP - The Internet Protocol. A protocol for uniquely addressing machines on the Internet (hosts) and enabling them to find each other.

The IETF also defines the standard for HTTP (HyperText Transfer Protocol) so is an important organisation from the Web perspective.

In summary the IETF controls how computers find each other (addressing), how computers move data between each other(transmission), and how a hypertext system can be built on top of a system that already offers both addressing and transmission.

Note that all of these layers could be replaced with different approaches. No single layer is necessary in the form that it currently exists and each will be enhanced, and possibly replaced, over time.

The surface features of the Internet and Web stem from the aggregation of these underlying standards providing sufficient functionality for higher level needs, like transmitting HTML from a server to a browser, but other approaches could be taken, i.e. using different addressing protocols or different transmission protocols.

11.13 The World Wide Web Consortium (W3C)

The World Wide Web Consortium (W3C) is another international standards organisation, like the IETF. This time though the focus is on the needs of the Web specifically rather than the wider Internet. The W3C has traditionally taken a quite conservative and unwieldy approach to standardisation. The standardisation process involves multiple stages:

- Working Draft - After discussion of ideas a WD is published for review. Can implement WD standards but likely to be huge future changes
- Candidate Recommendation - When standard meets initial design goals the wider community is solicited to feedback on practicality of implementation
- Proposed Recommendation - Having passed through WD and CD stages a PR is put to the W3C advisory council for approval
- W3C Recommendation - A standard that has been endorsed by W3C membership after extensive development, testing, and evaluation
- Revisions - Updates or extensions published until sufficient for new edition of the standard

11.14 The Web Hypertext Application Technology Working Group (WHATWG)

The Web Hypertext Application Technology Working Group (WHATWG) was established in 2004 as a response to the slow pace of development of W3C web standards. Primarily the decision by W3C to focus on XML-based technologies instead of HTML was seen as a mis-step by many. Web developers are famously pragmatic and XML introduced huge complications without obvious immediate benefits. Thus the WHATWG was born to work around the W3C. The central and steering members include: Apple, Mozilla, Google, and Microsoft, so some of the biggest commercial organisations in computing and technology. However contributors to the WHATWG can be anyone who joins the WHATWG mailing list.

The WHATWG focus is upon the following:

- The HTML Living Standard. This is essentially the development of HTML 5 and beyond. HTML 4 is now adopted by W3C as the starting point for new HTML development. The notion of a living standard means that rather than a single, defined standard, the move is now towards continuous changes as the Web, and the way it is used, evolves. The living standard also incorporates additional APIs beyond pure HTML, e.g. WebSocket, WebWorker, LocalStorage, etc.
- DOM standard defining how the Document Object Model works so that all browsers can operate to a shared model.
- Fetch standard defining requests and responses and the process that unifies them so that there can be a single agreed way to access remote APIs from a browser instead of using XMLHTTPRequest or some other similar hack.
- Plus a whole host of additional ancillary standards including the web worker API, Microdata vocabularies, storage API standards, data streaming API standards, Encoding standards (e.g. UTF-8), MIME-type sniffing, URL parsing standards.

PThere are some benefits to having the WHATWG in addition to the W3C. The WHATWG assumes some responsibilities that the IETF has traditionally worked on alone, e.g. URL standards, HTTP standards by replacing RFCs with WHATWG living standards. The aim is to assume responsibility for developing standards more rapidly and pragmatically and this has led to improvement. The disastrous direction of HTML during the XHTML years has been addressed, replacing the complex HTML4 standard with a streamlined, focused, and more understandable HTML 5 standard.

11.15 Summary

We've now completed our survey of client-side web technologies by considering how to deploy our websites so that they can be found by Web users. During this process we've looked at moving from local development to web development. Weve investigated servers and clients, and how these relate to Web servers and the HTTP protocol. Weve also considered the various options for deploying our websites, from free hosting through to more expensive cloud hosting. Finally, we introduced the three important organisations that

set the standards which underpin the Internet and the Web. Without these organisations there wouldn't be the agreements that enable the Web to work the way it does. As your career will probably outlast the current Web standards, it is well worth knowing to whom to turn to find out what new ideas and technologies are on the horizon.

Chapter 12

Conclusion

We've surveyed a whole lot of topics so far which should lead to a number of conclusions. Firstly, we can't learn everything about the Web from one book. Instead, this one book is just a starting place, but the onus is on you to keep up-to-date with new developments in the field. The second take away is that the Web is a fast moving collection of technologies that have changed over time, and will continue to change. Ultimately, the Web is a reflection of humanity's desire to share information, and to find better ways to use technology to do so. Perhaps that story started when we daubed pigments onto cave walls, or scratched images onto rocks, or perhaps it was with the development of writing systems, or event he printing press. Regardless, that story has not finished yet, and perhaps never will.

However, if you've liked the topics covered so far in this book, which has focussed on the client side of Web development, perhaps you'll enjoy the follow on book that details server-side development and considers the future of the web.