Overview

This page contains the exercises for Part 2 (the last 5 weeks) of COMSM0085 Software Tools. As with Part 1, each week usually contains two workbooks -- sets of exercises which are supported by videos to watch, or by set reading materials (more prominent in this part of the unit). The exercises will require you to do some work within your Linux VM in Vagrant, which we'll be supporting in the labs, but which you should attempt before the lab to get the most out of the unit.

Part 2 focuses on web technologies. The Web is one of the largest software ecosystems ever built, and a lot of programmers will either work on web technologies or make use of them to present their other work. We'll be covering the basics: how to write webpages, style them using CSS, make them interactive using Javascript. Then we'll look at ways to treat websites and webpages as parseable resources, and conclude by looking at the kinds of encryption needed in common web transactions.

HTTP

HTTP, or HyperText Transfer Protocol, is one of the basic protocols that make up the internet - and usually the lowest level that a web developer needs to concern themselves with frequently.

Videos

Video	Length	Slides
HTTP	26 minutes	slides
The Internet	15 minutes	slides
URLs	7 minutes	slides

Reading

- RFC 7230 - this is a comprehensive outline of the protocol for HTTP, but you can focus on sections 2.1, 2.2, 2.7 and 3.

Exercises

- Setup
- Exploring HTTP
- Online research
- A server in Java

Optional

You can read the RFC for the internet protocol IP at RFC 791.

You can read the RFC for the transmission control protocol at RFC 793.

You can read the RFC for URIs at RFC 3986.

If you want to, you can read further about HTTP (including the new HTTP/2 standard) on the Mozilla Developer Network.

Set-up

From this point on we will be running web servers on our development environment, and connecting to them with web browsers. To get this to work we need to set some things up first, so please follow the instructions for your environment:

Lab machine, native

These instructions apply if you are working directly on a lab machine (not using a Vagrant VM).

You do not need to set up anything special. As you are not root, you can only serve from TCP ports 1024 or above, but that is not a problem - we will use port 8000 by default.

Proceed directly to exploring HTTP.

Lab machine, with Vagrant

If you want to run a server inside a Vagrant VM and connect to it with a browser directly on the lab machine (outside the VM), then you will need to tell Vagrant which ports within the VM to make available outside the VM. Open your Vagrantfile and add the line

```
config.vm.network "forwarded_port", guest: 8000, host: 8000
```

in the main block (just before the `config.vm.provision` line will do), then restart your VM if it is already running. This tells Vagrant to open a server on port 8000 on the host (the lab machine), and forward any traffic it receives to port 8000 on the VM.

You can now start the VM, run a server inside the VM, and connect to it from the browser on the lab machine.

Proceed to exploring HTTP and run the server (the `nc` command) inside the VM, but use the browser on the lab machine directly. Note that the `http-response` file mentioned on that page needs to go in the correct folder inside the VM, where you started the server.

Your own machine, without Vagrant

If you have a system that supports the `nc` and `wget` commands (Mac OS, Linux or WSL should all offer the option to install them) then you can proceed directly to exploring HTTP.

You might get one or more firewall warning pop-ups, which you will have to "allow".

Your own machine, with Vagrant

Configure vagrant as in the section "Lab machine, with Vagrant" above, that is add the line

```
config.vm.network "forwarded_port", guest: 8000, host: 8000
```

to the Vagrantfile and restart Vagrant. You can now run servers in the VM and clients (such as your favourite browser) on your own machine.

You might get firewall warning pop-ups, which you will have to "allow".

Proceed to exploring HTTP. If you don't have `wget` on your own machine, you can open a second terminal and log in to the VM from there, then run both `wget` and the server inside the VM that way - then later on, when you're using a browser instead of `wget` for the client, you can just open the browser on your own machine.

Over SSH

For security reasons, you will not be able to run a web server on a lab machine (whether in a VM or not) and connect to it from your local web browser. If you have no other option than to use a remote connection, you should use x2go to connect to a lab machine with a graphical user interface, and run both the server and browser through that. This software uses ssh internally so you need to connect to `rd-mvb-linuxlab.bristol.ac.uk` using `seis.bris.ac.uk` as the proxy server.

Exploring HTTP

Before you start

To avoid an annoying problem, before we run our own server we are going to check that no-one else is already using the TCP port we want.

Run the command

```
wget localhost:8000
```

on the machine where you want to run the *server*. It should time out after a couple of seconds (or instantly) with "failed" and some error message.

If it succeeds (shows "Connected" or "200 OK" anywhere), then someone or something else is using port 8000.

Next, run

```
netstat -tan
```

and check that there is no line with the value "8000" under Local Address and a state of LISTEN or ESTABLISHED. If you get a lot of lines, `netstat -tan | grep 8000` will help. It does not matter if 8000 appears in the Foreign Address column, but there must be no-one using port 8000 under Local Address.

`netstat` should work on every OS including Windows (Microsoft has written their own version), but `wget` doesn't exist by default on Windows machines although it can be downloaded for free.

If either `wget` or `netstat` suggests port 8000 is in use,

- If you are on a lab machine, it could be that another student is using port 8000 (maybe they are logged in via SSH).
- You might have other software (such as a web development package) already using that port.

You can try port numbers 8001, 8002 etc until you get one that is definitely not being used and then replace 8000 with that in all further exercises today, or (on your own machine) you can stop whatever program is using the port first.

Note: if you have vagrant running and set up as described on the last page, and you run `wget localhost:8000` on your own machine (not the VM), then vagrant will be using port

8000 and `wget` might block waiting for a reply. This is the correct behaviour, and this is why I asked you to run `wget` on the machine where you want to run your *server*, not the *client*. If you want to do the following exercises with both server and client on your own machine, then you need to stop the vagrant VM first or remove the 8000 line from your vagrantfile.

A basic HTTP server and client

Open a terminal on the machine where you want to run the server, and download the file http-response to the folder where your terminal is open (for example with `wget`). Then run the command

```
nc -l -p 8000 < http-response
```

Or if you are on a Mac:

```
nc -l 8000 < http-response
```

This calls the tool `nc` ("network cat") and tells it to listen on TCP port 8000 (`-l -p 8000`), e.g. to run a server there. When a client connects, `nc` will by default read from its standard input and write this to the client, so we use a shell redirect to make it read from a file instead. The file contains some standard HTTP:

```
HTTP/1.1 200 OK
Content-type: text/plain
Content-length: 18
Connection: close

Hello over HTTP!
```

(If you want to type this out yourself or copy-paste, you **must** save the file with CRLF line endings and put two newlines after the hello message.)

`nc` will block until you connect a client. Open another terminal on the machine you want to run your client and run

```
wget -q -S -O - localhost:8000
```

You have now made a HTTP connection from the wget client to the nc server. The server terminal should print out the HTTP request it got from the client (this is built into wget) and the client should print out the response from the server (which comes from the file).

Connecting with a web browser

Run `nc -l -p 8000 < http-response` again on your server machine (kill the old one with `Control+C` first if it hasn't terminated already).

Open a web browser on your client machine (e.g. your own PC/laptop, or a lab machine). If you're using chrome/edge or firefox, open the debug tools (`F12`) and go to the network tab before you open the page.

Navigate to `localhost:8000`. You should see the line "Hello over HTTP!" and `nc` will print the HTTP request from your browser. You can then close it with `Control+C` if it doesn't terminate by itself. In your browser's debug tools, on the network tab you should see a file `localhost`, click this and then select *Headers* in the details that appear.

This details page shows you the HTTP request details, including the response code (`200 OK` in this case) and all the headers sent in both directions. This screen is one of the first places you should look at when something web-based that you've coded is not working correctly or at all, as HTTP is normally the lowest layer you need to care about.

In particular, the `Content-type` header is a common source of mistakes: if you want to serve e.g. a script or a stylesheet but you don't set the correct type header, your browser may download the file just fine but won't do anything with it. Similarly, whether you send `text/plain` or `text/html` determines whether the browser will interpret a file you navigate to as HTML or not.

A web server in C

On your server machine, clone the repository `https://github.com/emikulic/darkhttpd` which contains a single-file web server in just under 3000 lines of C. You do not need to understand this code in detail, but it can be educational to try to understand some of what it does internally.

Compile the program either with `make` or simply `gcc darkhttpd.c -o darkhttpd`. The server's job is to serve files within a folder, so make a subfolder called `web` and then run it with `./darkhttpd web --port 8000`. You can stop it again at the end of the exercise with `Control+C`.

You can now experiment with the following:

- Create some files (plain text, HTML, image etc.) in `web/`.
- Access them from your browser with `localhost:8000/FILENAME`, for example `web/hello.txt` would become `localhost:8000/hello.txt`.
- Observe the HTTP headers in your browser's developer tools, and the server logs printed in the server terminal.

This particular server is written so that if you try and access a folder instead of a file, for example just `localhost:8000` which has the implicit path `/` which maps to the `web` folder, then it shows a list of files in this folder as clickable links.

Pay particular attention to how the `Content-type` header is sent depending on the file extension. From the browser's point of view, that header is what counts: if you send a file with `.html` extension but set `Content-type: text/plain`, the browser would not interpret it as HTML. This makes it possible to serve URLs with no "extension" at all like `/students` for a database application, and still have the browser understand what to do.

From the server's point of view, this server (like most other servers that simply serve files from a folder) has chosen to use a file's extension to determine what `Content-type` to send; this is implemented in the `default_extension_map` starting at line 333 of the source file.

You can try this out for yourself if you want: make a HTML file called `one.html5` in the web directory and access it with the browser. (`.html5` is not an official extension, it's something I just made up. The correct one to use for proper websites is `.html`.) The server won't send a `Content-type` header at all, since it doesn't know this extension, so the browser will either display it as a plain text file or download the file instead of showing it.

Edit the map in the source file and change the entry for `text/html` to read `" html htm html5"`, keeping the space at the start of the string. Recompile the server.

Rename the file to `two.html5` (I'll explain why in a second) and restart the server and try and open the file in the browser. This time, the file will display as a HTML page.

Why did you have to rename the file? Because the browser will otherwise try and be clever if it thinks you're trying to access a file that you've just downloaded already. The server sends a `Last-modified` header, and if the browser notices you're asking for a file you have just downloaded, with the same timestamp as the HTTP header indicates, then your browser might ignore the rest of the request and not see that another header has changed. Changing the file name forces the browser to look again, as it thinks it's a different file now. Deleting the downloaded file, or editing the file on the server, would both have the same effect.

There is a moral here: if you're doing web development, and you are trying to find out why a change you made isn't showing in the browser (even if you refresh the page), it might be a cache problem. On the network tab of the developer tools window, there should be a checkbox to "disable cache" which you might want to turn on in this case and reload the page again, to exclude the cache as the possible source of the problem.

HTTP/URL research exercises

Research online what the following are and what they do:

- The *fragment* part of a URL.
- The `Accept` header sent by the HTTP client.
- The `User-agent` header sent by the HTTP client. What does your browser send?
- How do you encode a path that contains a space in an URL? Which other characters are "special" and need to be treated this way in paths?
- Which web server is the University of Bristol using for `www.bristol.ac.uk`, according to the headers? Read up online on this server and the organisation of the same name that maintains it.

A server in Java

From now on, we will be using the Java language and the Spring Boot framework to run real web servers, as you might do in a software engineering project.

Compile and run

Make sure you have java and maven installed as we did earlier in the unit. Clone the repository `git@github.com:cs-uob/COMSM0085` if you have not done so already, and navigate to the folder `code/server01`. In this folder, run `mvn spring-boot:run` to compile and run the sample application. The first time you do this, it might download lots of files.

This runs a web server on port 8000.

Explore the application

The `pom.xml` file tells maven that this is a spring boot project, and what the project is called (`softwaretools.server01`).

Under `src/main/resources` you find two files. First, `application.properties` which is a spring file configured to run on port 8000 (the default would otherwise be 8080). Secondly, a HTML file that the application can serve.

Under `src/main/java` is the source code. This is only two files, but of course the whole spring framework is active when the application runs. `Server01Application` is the main class, but this just contains boilerplate code for now.

`Controller.java` is the interesting one: in application development, *Model - View - Controller* is one of several patterns to structure an application, where a *Controller* is a piece of code that does the heavy lifting part, for example replying to a HTTP request.

Spring works with annotations, special classes whose name begins with an `@` sign. When the server starts, spring scans all files for annotations and sets up code to deal with them. Here we can see:

- `@SpringBootApplication` (on the `Server01Application` class) tells spring that this is the main file to run.
- `@RestController` tells spring that this class contains methods to deal with HTTP requests. It is so named because spring has libraries to make implementing the REST principles particularly easy.

- `@AutoWired` on a field tells spring that this field is spring's responsibility to set up when the application starts. The `ResourceLoader` is part of spring and lets you access files in `src/main/resources`.
- `@GetMapping(PATH)` tells spring that this method should be called to reply to a HTTP GET request for the provided path (there is of course also a `@PostMapping` and so on).

The `mainPage` method, which is called when you load `localhost:8000`, shows the basic way to reply to a HTTP request: set up any headers you need - in this case the content type - and return a `ResponseEntity` taking three parameters: the HTTP body of the response to return (this will be shown in your browser), the HTTP headers to set, and the response code which is 200 (OK) here.

The `htmlPage` method replies to requests for `localhost:8000/html`. Here we want to serve a file, so we use the resource loader to get it from the classpath which includes the `src/main/resources` folder (or rather, the version of it that ends up in the compiled JAR file). It also shows a second way to create a `ResponseEntity` using the *builder* pattern.

Exercises

- Compile and run the application, and test both `localhost:8000` an `localhost:8000/html` in your browser. Observe both the headers in the developer tools, and the log messages that spring prints out for each request.
- Add a method that replies to requests for `localhost:8000/bad` with a HTTP `404 NOT FOUND` error. The body of the page can be a simple string with a message of your choice. Stop and restart the application, and check that you get the correct error message when you try and open this page.

HTML5

HTML5, officially introduced in 2014, is the modern markup language for writing websites.

It gets rid of a lot of historical annoyances and inconsistencies from previous versions of HTML and XML and contains a lot of new standard tags and attributes for features that individual sites had built with JavaScript or Adobe Flash in the past. For example, HTML5 has a `<video>` tag for a consistent way to include videos that work across different platforms and without plug-ins, and a standard way of writing a date picker or "calendar" widget.

Videos

Video	Length	Slides
HTML5	16 minutes	slides
HTML elements	9 minutes	slides

MDN

For the web part of the unit, we will be reading some of the documentation on the Mozilla Developer Network (MDN) for some workshops, to complement the videos and exercises.

These pages are pitched at a lower level of technical expertise than CS students, so you should get through them very quickly. Many of them also contain interactive examples to try out. If your future career involves web development, you will be consulting documentation like the MDN HTML reference guide a lot - for example, to read through all the attributes you can put on a form tag.

Please read the following pages for this workshop, as well as watching the videos.

- MDN: Getting Started with HTML The very basics, and the HTML5 page template.
- MDN: HTML Text Fundamentals A recap of paragraphs, headings, styling, semantic tags.
- MDN: Creating Hyperlinks All about the `<a>` tag.
- MDN: Document and website structure HTML5 semantic tags for structure.
- MDN: HTML5 input types More on using the most appropriate controls in your forms.

Exercises

- Basic HTML5
- Templates

Basic HTML5

For your first exercise, create a page called `index.html` and add the HTML5 template code to it.

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="utf-8" />
   <title>A web page</title>
 </head>
 <body>
    <h1>An example web-page</h1>
    <p>A paragraph of text here, perhaps with <a href='./another_page/'>a
link</a></p>
 </body>
</html>
```

The name `index.html` is a convention: most web servers, if you give an URL that looks like a "folder" e.g. `example.com/pages` (or `example.com/pages/` with slash on the end) will serve the file `index.html` from within that folder, if there is one.

Create a page

Create content in your page to make it look like the following, then open it in your browser. All body text, but not the bullet list, is contained in paragraph tags.

http://localhost:8000

404

File not found

The site configured at this address does not contain the requested file.

If this is your site, make sure that the filename case matches the URL as well as any file permissions. For root URLs (like `http://example.com/`) you must provide an `index.html` file.

Read the full documentation for more information about using **GitHub Pages**.

GitHub Status — @githubstatus

Validate your page

There are various libraries and services that check if your page is valid HTML5.

At validator.w3.org you can enter an URL, upload a file, or paste the contents of a file and validate it. Check that the page you have just written is valid HTML5.

Your browser's developer tools can also check the page you are viewing. For Chrome/Edge, to to the *Console* tab and, if the button at the top does not read *No Issues* but shows a number with a blue (or even yellow or red) icon, then click it to open the *Issues* tab and see what the browser is complaining about.

You can check the sample solution here but you might have a different, equally valid solution.

Templating

The bread and butter of many developer jobs is building web applications that let people interact with data in a database. In a university for example, such an application might let students see the units they are enrolled in, their teaching and exam timetables, and their grades. Academics would be able to add new grades, and staff in the timetabling office would be able to create and edit timetables.

In such an application, pages need to be somewhat dynamic: the "student profile" page will show data for the particular student that you are looking at. Rather than generating these HTML pages "manually", for example by pasting strings together, it is much cleaner and usually more efficient to use a templating library. You could define one "student profile page" template, and then when a student requests their profile page, the server uses the template to create the page with this student's data. Creating a particular page from a template and some data is called *rendering* the page.

This workshop's example application is `code/server02` in the unit git repository. You build it with `mvn spring-boot:run` like the previous one, then access `localhost:8000`.

The application is a very minimal university database with students, units and grades. The `Database` interface in the `model` folder, together with the Student and Unit classes, form the data model - you can search for all students or all units in the database. In a real database of course, you would be able to search for an individual student by ID as well.

The file `Templates.java` sets up the thymeleaf template engine. The `@Component` annotation tells spring to manage this class; other classes that need templates can request it by declaring a field of the right class with `@Autowired`, as you can see in `Controller.java`.

The main page is served up as before as a HTML file from the classpath in `src/main/resources/web`.

Have a look at the process when `/units` is requested.

- First, the method `unitsPage` accesses the database and loads the list of units.
- Next, it creates a `Context`, a thymeleaf object that lets you pass values to a template. Here we add one value with key `units`, containing the list of units.
- Finally, we render the template with the context and return this. If a method in a spring controller returns a string, then the string is assumed to contain a HTML page and spring correctly sets the HTTP status code and headers.

The template itself is in `src/main/resources/units.html`. Anything with a `th:` prefix is rendered by thymeleaf:

```
<ul th:each="unit : ${units}">
    <li>
        <a th:href="'/unit/' + ${unit.code}" th:text="${unit.code}"></a>
        <span th:text="${unit.title}"></span>
    </li>
</ul>
```

The `th:each` attribute is the equivalent of a for loop in Java, in this case `for (Unit unit : units)` - in thymeleaf, unlike plain Java, you do not need to give the loop variable a type but you do need to use the `${...}` syntax whenever you are accessing a variable from the context. The `th:each` attribute renders its own tag once (in this case `<ul>`) and then renders everything inside the tag once per pass through the loop, so you get one `<li>` item per unit.

To create an attribute, you put `th:` in front of the attribute name, so the `th:href` creates the `href` attribute of a link (check the result in your browser). Thymeleaf attribute syntax is that you need to put single quotes around strings, `+` like in Java to stick strings together, and `${...}` around variables.

The syntax `${instance.attribute}` loads an attribute from an instance of a Java class by calling its getter. In this case `${unit.code}` ends up calling `.getCode()` on the unit instance rather than accessing the field directly.

The `th:text` attribute is special and creates the body of a tag. For example, the span tag here creates a `<span>` whose body (between the opening and closing tags) contains the unit's title. Thymeleaf, unlike some other templating systems, doesn't let you include variables everywhere - some template engines let you just write `${variable}` anywhere in the HTML page, but thymeleaf only lets you do that in an attribute of a tag so if you simply want some text, you need to make a placeholder tag (`span` is a good choice here) and put the variable in its `th:text` attribute.

If you click on a unit, you get to the details page for that unit (that currently doesn't have any more detail than the code and title). When you send a HTTP request for `/unit/UNITCODE` , for example `/unit/COMSM0085` , this is handled by the `unitDetailPage` method.

Note that the `@GetMapping` contains a pattern `/unit/{code}` with a parameter to match all URLs of this form, and the parameter is passed to the method as a parameter that is annotated with `@PathVariable` to tell spring how to deal with it, namely to fill its value from the path of the HTTP request.

The method first finds the unit in the database and returns a 404 error page if there is no unit with that code. (In a real database, there would be a "find unit by code" method, you would not have to load all units just to find one of them.)

After this, we call thymeleaf on the `unit.html` template (this is the one without 's' in its name) to render the unit page.

Exercises

Basic exercise: Rewrite the units list page to show the units in a table instead of a list. The table should have one row per unit and three columns: the first column should be the unit code, the second should be the unit title, and the third column should contain a link with text "details" that takes you to the details page for the unit. Also make a header row in the table with headings 'code', 'title' and 'link'. (This is purely a HTML and templating exercise, you do not need to change any Java code for this.)

Intermediate exercise: Implement controller methods and templates for listing all students and viewing an individual student's details, analogous to the ones for units (you can choose whether you make a list or a table for the students - there are currently only two). You only need to show a student's id and name for now, not the grades. Note that the id is an integer, not a string. You can copy-paste the Unit controller code and templates and make the necessary changes to show students instead, but make sure you understand what the bits do that you're changing. The student class is in the `src/main/java/softwaretools/server02/model` folder.

Intermediate exercise: The student class contains a method `getGrades()` that returns a list of pairs (unit, grade). The unit is a unit object, and the grade is an integer. On your student details page, make a table listing the titles and codes of all the units the student has taken, and the grades they got.

CSS

CSS, or Cascading Style Sheets, allows you to set what your page looks like, separately from creating the content.

Not to be confused with CSS Bristol.

Videos

Video	Length	Slides
CSS part 1	14 minutes	slides
CSS part 2	10 minutes	slides

MDN

For this workshop, please read the following pages:

- CSS first steps including the 5 subpages linked under "Guides" - they are all meant for beginners so you should be able to get through them quickly.
- CSS selectors
- The box model
- Values and units

Exercises

- Styling Text
- Frameworks

Styling Text

Under resources/sometext.html you have a basic text-based HTML document with some information about your degree. Download this file and open it in your browser, you will see it displayed with the default stylesheet (and any customisations you might have applied, such as selecting a default font and size).

Reading Mode

Many browsers have a *reading mode* which is meant to make text as east to read as possible. Let's simulate this - create a new link to a stylesheet in the header (call it something like `sometext.css`) and insert the following:

```
body {
    margin: 0 auto;
    max-width: 40em;
    line-height: 125%;
}
```

This already looks much better, but you can play with the parameters to get something that's perfect for you.

- The `max-width` is doing most of the work here, following the design principle (that's fairly well replicated with studies) that most people find text easier to read if lines are not much longer than some limit, quoted variously as around 60 characters. Since an 'm' is wider than the average character and `em` rather than e.g. `px` is the correct thing to use here (as it's relative to the font being used), I went for 40em.
- The `margin: 0 auto` centers content horizontally if it has a fixed width. The space around the text both gets the left margin away from the edge of the window, and gives the eye an area around the text free of distractions.
- Increasing the line height follows the same principle, sometimes known as giving the text *room to breathe*. Experiment with differnt line heights if you like to see which one suits you best. Note: it's possible to overdo this. Double-spacing (`line-height: 200%`) was common for dissertations in the typewriter era, mainly to give proofreaders and markers space to write comments between the lines. This is considered bad design nowadays, and you should not do it.

If you want, you can try a few other things:

- Change the `font-size` . The default is `16px` but depending on your screen size, resolution and distance from your eyes you might want a larger one.
- Set a different `font-family` . The values `serif` and `sans-serif` give default choices across all operating systems, you could set this to a more specific font on yours such as

`Calibri` (if on Windows) but bear in mind this will not work for Mac users.
- Set a `background-color` on the body, either a light but not quite white one (pale yellow/brown is often used), or go dark mode with a dark backround and a light `color` for the font itself.

Advanced note

Tricks like this - which you can code as a browser extension if you want to - might be particularly helpful for visually impaired or dyslexic students so you can force any website to use fonts and styles that work for you.

If you are designing for dyslexic people other than yourself, the best you can do is give them a way to configure the fonts and colours themselves (for example with a menu that lets you change them via JavaScript) - different dyslexic people might have different preferences that work for them, for example one person may prefer serif fonts to sans-serif or the other way round, there is no one best design for everyone.

There are statistical trends which mean that "use sans serif fonts" is good advice if you have to pick one font (for example for printed material) but one of the advantages of the web is that you do not have to choose one font in advance, you can let different users select different preferences.

Starting from Scratch

Uncomment the line that includes `reset.css` in the HTML file (comments go from `<!--` to `-->` , a relic of when HTML was connected with XML). The file is at resources/reset.css so download it to the same folder as your HTML file.

You now have almost no styling at all: headings, lists, paragraphs all just look like text.

In your own stylesheet, which you should include *after* `reset.css` , either try and recreate the default settings or try out ones that look good to you.

Headings are meant to stand out among the paragraphs, and the extra white space around them (padding or margin) is as important as the larger font size and bold font. You could try leaving headings the same size as the body font but still `font-weight: bold` and use just padding to make them stand out. (Since a heading relates to the text after it not the text before it, it is good design to have a greater margin/padding above the heading than below it.)

The default body font size is `font-size: 16px` . If you want to make the headings larger, you should not set an absolute value (such as 24px) but one relative to the default size. Since the default size comes from the `<html>` and `<body>` elements, you have two ways to do this: either with a percentage e.g. `h1 { font-size: 150%; }` which makes headings 1.5 times as big as the paragraph font, or using the `em/rem` units (setting a font size in ems refers to the em size of the parent unit).

In the browser, open the developer tools and type the following in the console and press ENTER:

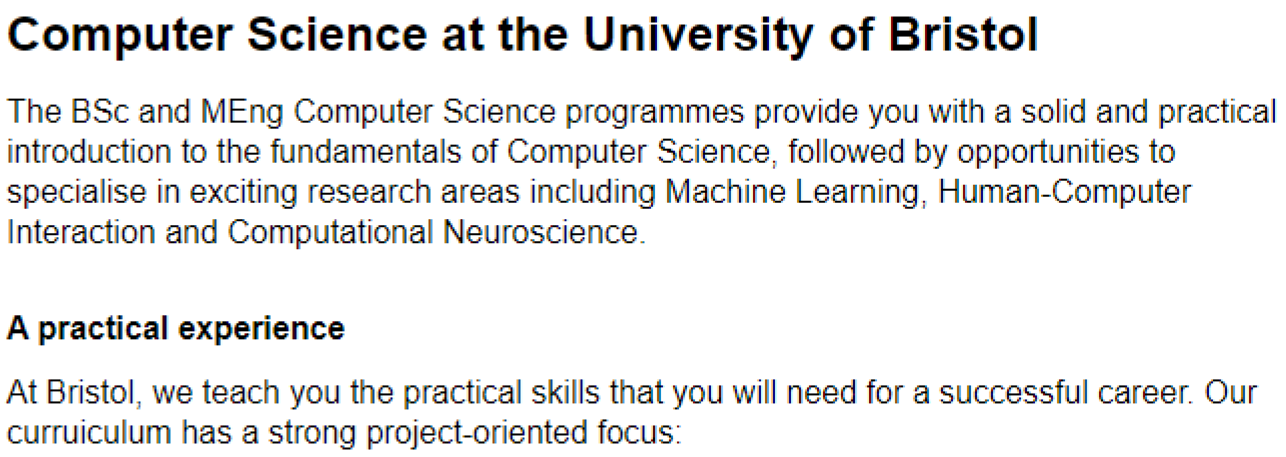
```
document.body.style.fontSize="24px";
```

(Since this is JavaScript, the property is called `fontSize` not `font-size` as dashes are not allowed in identifiers.)

This increases the body text size from 16px to 24px (if you have manually set a different size in your stylesheet, you might need to set it to `"24px !important"` to override this). If you had set your headings to fixed sizes like 18px, then headings would now be smaller than the body text - if you have correctly used relative sizing, then headings will now still be 1.5 times as big as the body fonts or whichever multiple you set them to be. That is the right way to do it.

Vertical Rhythm

Here is another design principle worth knowing about. Look at this image, part of a screenshot of one way to style the text on the page:



Computer Science at the University of Bristol

The BSc and MEng Computer Science programmes provide you with a solid and practical introduction to the fundamentals of Computer Science, followed by opportunities to specialise in exciting research areas including Machine Learning, Human-Computer Interaction and Computational Neuroscience.

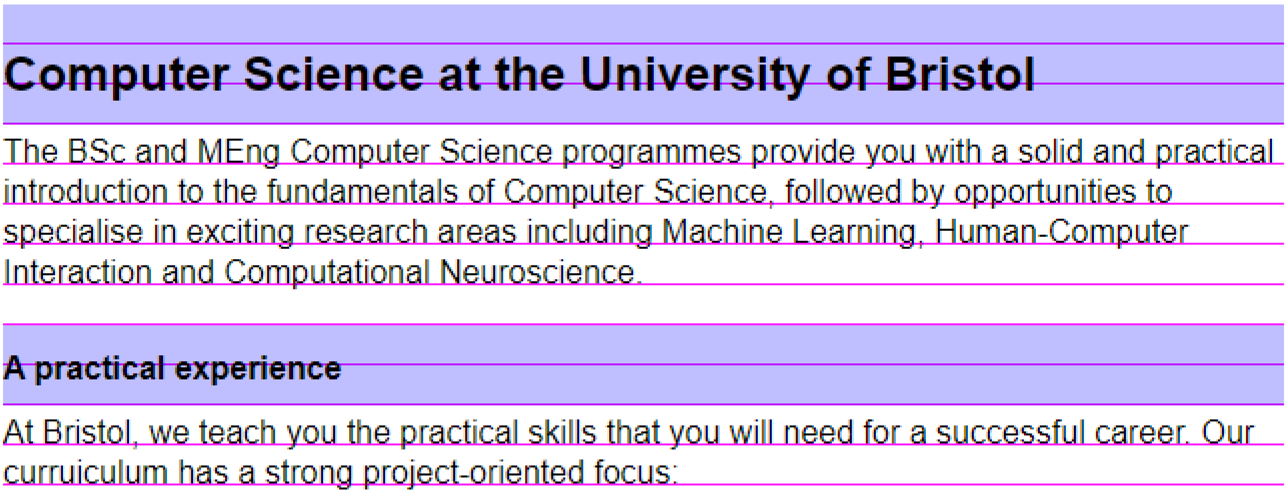
A practical experience

At Bristol, we teach you the practical skills that you will need for a successful career. Our curruiculum has a strong project-oriented focus:

The principle here is that we imagine a grid with 20px high lines, and lay out everything according to that. Under resources/baseline.png there is an image 1px wide and 20px high with the last pixel coloured pink. Download that image to the folder with the HTML file and set

```
body {
    background-image: url("baseline.png");
}
```

This will make the image repeat horizontally and vertically, and you can now see the design behind this approach:



Computer Science at the University of Bristol

The BSc and MEng Computer Science programmes provide you with a solid and practical introduction to the fundamentals of Computer Science, followed by opportunities to specialise in exciting research areas including Machine Learning, Human-Computer Interaction and Computational Neuroscience.

A practical experience

At Bristol, we teach you the practical skills that you will need for a successful career. Our curruiculum has a strong project-oriented focus:

The baselines of the paragraph text all sit on the grid lines. The important thing here is that when a heading interrupts the flow of text, the total height of the heading is an exact multiple of the grid height, even if the font size is different.

Try and recreate this layout or a similar one using vertical rhythm, starting from the reset stylesheet so you have to do everything yourself from scratch. Notes:

- The aim is that all paragraphs have their baselines on the grid, even if there are other elements (headlings, lists) earlier in the document.
- This means that the total height of headings must be an exact multiple of the grid size; you can assume that the text in a heading does not require more than one line.
- You should set all sizes, including margins and padding, in terms of relative units (either ems/rems or percentages).
- The base font size here is the default 16px. Start with the max-width and line-height from the first example, and modify the line height if necessary for your font to get the lines of paragraph text the right distance apart.
- The paragraphs themselves have a margin at the bottom (the empty line after the first paragraph) that also appears between paragraphs of text (not shown in the image) when one `<p>` tag follows another. You might also need to give them a small top padding to get the text on the lines, not between them.
- The shading around the headings is `background-color: rgba(0, 0, 255, 0.25);` to be semitransparent and let the lines show through, to see that while the heading text itself doesn't have to sit on a grid line, the total height of a heading including padding does respect the grid. However this shading is just to motivate the layout: if you want an actual coloured background, then you should set a `padding-left: 0.5em;` or something on the headings to make the text not start just on the edge of a coloured box. This improves readability.
- You will also need to style the `<ul>` list to have some spacing before and after, and to keep everything on the grid lines. You get the bullets back with the following code - or one of many other options listed on MDN:

```
ul {
    padding-left: 2em; /* the bullets appear in the padding */
    list-style-type: disc;
    list-style-position: outside;
}
```

- Don't forget to style the link. `text-decoration: underline` is one option; or you can read more on MDN about styling links including how to make a link someone has visited before look different to a new one.

Using a CSS Framework

Now that you know some CSS, you could write your own framework - one or more stylesheet files that implement your idea of good design. In this exercise we will look at some examples.

Milligram

Milligram is, in its own words, a "minimalist CSS framework".

Download the page1.html file and open it in the browser: it contains some text and a sign-up form for a CSS conference.

Add the following to the page head:

```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Roboto:300,300italic,700,700italic">
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.css">
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/milligram/1.4.1/milligram.css">
```

- The first one loads Google's Roboto web font, which the designers of Milligram selected as their default. Because it is loaded as a web resource, it should work across different browsers and operating systems, you don't need to install it first.
- The next line loads `normalize.css`, an alternative to `reset.css` that provides a consistent stylesheet across browsers. Try out the page with just the first two stylesheets and see how it looks now - the font won't have changed because no-one has set `font-family` yet, that will happen in the next stylesheet, but the margins will be slightly different.
- The third one adds Milligram itself. Your page now uses a different style, for example the form fields and labels are laid out vertically and the register button is purple.

Milligram chooses to style the whole page by default, but you can customise this further. One of their features is a *container* that gives its content a fixed maximum width.

- Add `class="container"` to the `<main>` element, save and reload. See how the page behaves when you make the window wider and narrower. The page will always have some left/right padding, but the form adapts to the window size.
- In the developer tools, activate mobile mode a.k.a. *toggle device emulation*, on chrome/edge this is the second icon from the left in the top left of the developer panel. The text looks much too small now!

- This is due to a number of factors including higher pixel density on mobile screens. Milligram can handle mobile devices, but it needs the following line in the header:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

This line, which is often (but not always) good practice to include in a HTML5 page anyway, tells a mobile browser (or a device pretending to be one) to adopt sensible defaults including larger fonts depending on the pixel density and no horizontal scrollbars. You can read more about this on MDN.

On the milligram documentation page you can read more about the elements that Milligram styles for you, and how you can customise this (e.g. buttons).

Have a look in your browser's developer tools at how Milligram styles the page: select an element in the *Elements* tab (or right-click it and choose *Inspect*), then look at the styles that appear in the *Styles* area. How does Milligram set the following?

- Size of heading fonts
- Form fields take up the full width of the container
- Form labels and fields appear below each other
- Labels are closer to their own field than the field above
- Size and centering of everything in a container on wide enough screens

Bulma

Bulma is a larger but still fairly lightweight CSS framework. Unlike Milligram, it only styles parts of the page you tell it to (but it sets a default font), and it has some more components to build things like menus or panels.

Start with page2.html, have a look at the raw page and then add the following to the header (the viewport tag is already there):

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bulma@0.9.3/css/bulma.css">
```

Advanced note

Bulma, like most frameworks, comes in two formats: `bulma.css` is the CSS file with human-readable formatting, and `bulma.min.css` (at the same location) is the minified version with all unnecessary whitespace removed and so is slightly quicker to download.

The bulma sources on github are actually in SASS format, a language that compiles to CSS but offers extra features in the source file like nested blocks and variables. (After SASS was created, variables were introduced in CSS too, but with a different syntax.)

Bulma works with custom class names. For example, we are going to make the first heading into a banner that spans the whole page width - called a *hero* in web design circles. Bulma has classes for this.

- First, add `class="title"` to the h1 heading, save and reload the page.

This tells bulma to style it as a title/heading element. On the title element page you can see that the correct way to get a top-level heading is with the classes `"title is-1"`, whereas a `"title is-2"` would be a second-level heading (slightly smaller font):

```
<h1 class="title is-1">Title 1</h1>
<h2 class="title is-2">Title 2</h2>
```

This looks like you are saying the same thing twice, and you are, but it lets you keep the structure and the styling separate - tag names give structure, class names give styling. Note that this method is a convention by the bulma designers, which other designers might not agree with, not part of the HTML/CSS standard.

Now to the hero part. According to the bulma documentation for the hero component, we need a container with class `hero` and a direct child with class `hero-body`, so

- Add class `hero` to the header tag.
- Add a child div `hero-body` inside it that wraps the `h1` tag.
- Set the classes `title is-1` on the h1 tag itself.

You now have an eye-catching element at the top of your page.

Bulma also lets you give an element the class `content` to tell it to style everything within that tag based on the tag names. Class `section` creates a section with spacing before/after, and class `container` makes something fixed width and centered.

- Add class `content` on the `main` tag. Reload the page and observe the changes.
- Add a `div` inside main that wraps all the content inside, and give it class `container`. Reload again and observe the change.
- Notice that the first h2 heading text touches the bottom of the hero area, which is not good design - text needs space between it and edges of coloured areas. Create a section tag as the direct child of the container, and give it class `section` so your code should look like this:

```
<main class="content">
    <div class="container">
        <section class="section">
            ... content ...
```

For the next exercise, style the form.

- Put another section tag around the form and its heading (so the container tag will have two direct children, both of them sections).
- Using the bulma form documentation, add tags and classes to the form as necessary so it looks similar to the Milligram one (full width fields, labels immediately above the fields, coloured button with rounded corners). You don't have to make the button purple, use `is-primary` to make it blue.

Finally, add and style a bulma component of your own choice on the page.

Bootstrap

The most popular CSS framework is bootstrap, developed by Twitter. If you are interested and have some spare time, you can have a look at it - it has lots of different components and options to customise the styling further with themes.

Bootstrap includes a JavaScript file as well as a stylesheet: for some components like tabbed pages, you need the scripts to make clicking on a tab switch to the correct content.

CSS grids

Videos

Video	Length	Slides
Design	13 minutes	slides
CSS grids	7 minutes	slides
Responsive Layout	3 minutes	slides

MDN

Please read the following pages, and play with the interactive examples:

- Normal flow
- CSS grids
- Media queries

Exercises

- Introduction
- Curriculum exercise
- Trees (responsive layout)

CSS grids introduction

To create a CSS grid, you place a container element (usually a `<div>`) and style it `display: grid`. All its direct children will now be laid out in a grid, and you can configure the general layout on the parent element:

- `grid-template-columns` defines the number and widths of the columns.
- `gap` (which should have been called `grid-gap`, but is not) defines the gap between different grid cells.
- For a gap around the outside of the grid, give the container a margin (or padding).

On the child elements, you can set the placement rules for each one and the browser will build the grid accordingly:

- By default, each child takes up the next free 1x1 space.
- `grid-row` and `grid-column` modify this:
  - `span N` makes a child element N cells wide/tall.
  - `M / N` positions the child absolutely from dividing line M to dividing line N (you can overlap or stack elements on top of each other this way if you want to).

There are many more things that you can do with grids, and your best resources if you need to look this up are:

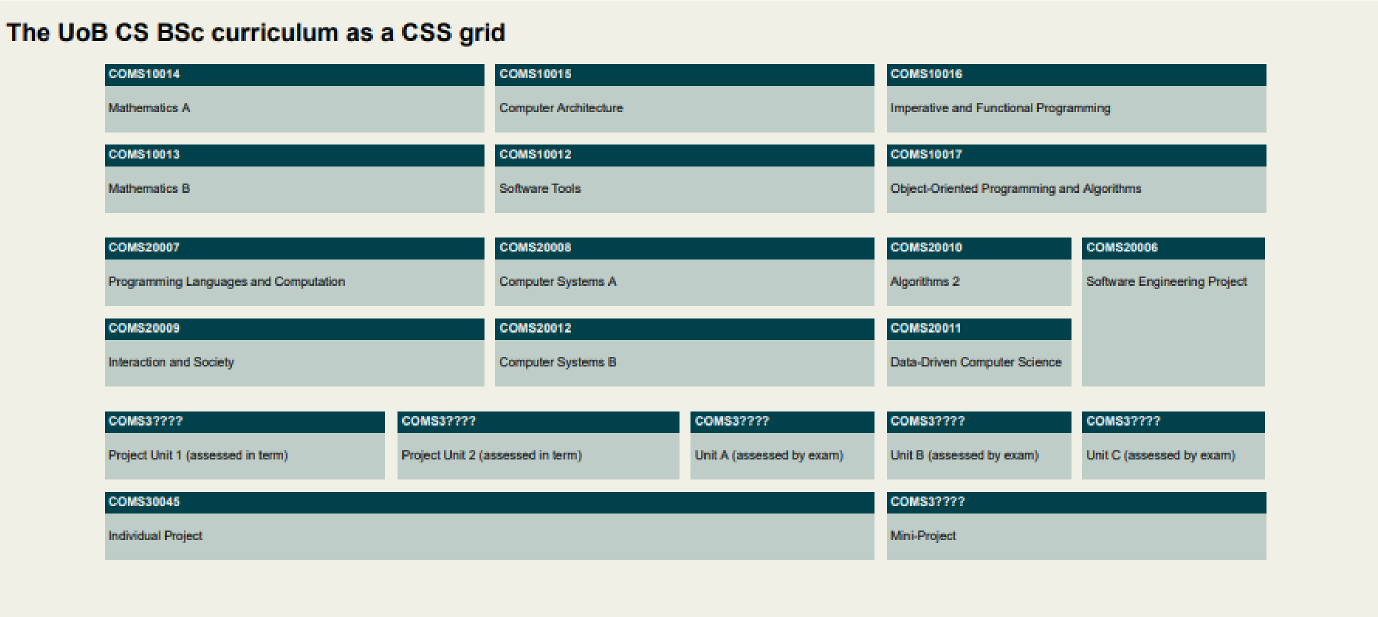
- Grids on MDN
- CSS-tricks complete guide to grids

Exercise: Curriculum

Download the file curriculum.html and open it in an editor.

You can see that it references a stylesheet file `curriculum.css` in the same folder, so create that file.

Your task is to write a stylesheet to make the curriculum look like this (for this exercise, we do not care about responsive layout):



The UoB CS BSc curriculum as a CSS grid

COMS10014
Mathematics A

COMS10015
Computer Architecture

COMS10016
Imperative and Functional Programming

COMS10013
Mathematics B

COMS10012
Software Tools

COMS10017
Object-Oriented Programming and Algorithms

COMS20007
Programming Languages and Computation

COMS20008
Computer Systems A

COMS20010
Algorithms 2

COMS20006
Software Engineering Project

COMS20009
Interaction and Society

COMS20012
Computer Systems B

COMS20011
Data-Driven Computer Science

COMS3????
Project Unit 1 (assessed in term)

COMS3????
Project Unit 2 (assessed in term)

COMS3????
Unit A (assessed by exam)

COMS3????
Unit B (assessed by exam)

COMS3????
Unit C (assessed by exam)

COMS30045
Individual Project

COMS3????
Mini-Project

(If you are a MEng student, mentally replace "individual project" with "group project" in 3rd year and then 4th year should follow the same layout as 3rd year, but you don't have to edit this in the file for this exercise.)

The units are coded like this in the file:

```
<main>
    <div class="unit y1-tb1 cp20">
        <b>COMS10014</b>
        <p>Mathematics A</p>
    </div>
```

Each unit is a `<div>` with class `unit` and further classes declaring its teaching block and weight in credit points. SPE in year 2 is special and is the only unit declared as `y2-tb4` with no CP class. By the way, why did we choose `cp20` and not `20cp` as the class name?

We are being pedantic here about separating content/structure (in the HTML file) and styling/layout (which you will write in a CSS file). There is no hint in the HTML file that the aim of the exercise is to lay the units out on a grid!

Advanced note

> Our class names are purely semantic - they describe attributes of the units (academic year, teaching block, credit points), not how they should be displayed. There is some debate among web developers and designers just how far you should take this separation; we are being absolutely purist for this exercise, but even bootstrap, one of the most widely used web libraries, has classes like `.col-4` to make something four columns wide. In practice the answer to how you separate concerns between HTML and CSS is really a case of "it depends".

Some general styles that you can apply in your `curriculum.css`:

```
body {
    font-family: sans-serif;
    background-color: rgba(112, 145, 53, 0.1);
}

.unit {
    background-color: rgba(0, 67, 79, 0.2);
}

.unit b {
    display: block;
    background-color: rgb(0, 67, 79);
    color: white;
    padding: 5px;
}

.unit p {
    padding: 5px;
}
```

This sets the colour scheme (derived from the official University of Bristol one), and some sensible padding/margins. The `<p>` elements already come with a default margin that we don't want to change here, but note that we applied the background colour for the text boxes to the `<div class="unit">` element itself, not the `<p>` elements otherwise the margin of the `<p>` elements would not be coloured in. The 5px paddinjg that we declared everywhere creates some space between the text and the edge of the boxes, which makes the text easier to read.

We used `<b>` tags for the title bars that are normally inline elements, but we redefined them to be block elements here (this is allowed) so they span the whole width of their grid cell.

Exercise

- The grid should be applied to the `<main>` element, with 12 equally wide columns and a maximum width of 1500px overall; you can center the grid for even wider screens with `margin: 0 auto;` on the main element.

- The gap between grid cells is `15px`.
- 10CP units should take up 2 of the 12 columns in width, 15CP units 3 columns, and 20CP units 4 columns. The 40CP project is 8 columns wide.
- All units are one row high except SPE which is 2 rows high.
- The aim of the exercise is not to write an individual rule for each unit! Try and use as few rules as necessary, you can do it with one rule per individual class that appears in the source file (you can even do it with a couple of rules less if you really understand grid layout).
- The main challenge will be getting the SPE unit placed correctly, and since this disrupts the normal flow of the layout, writing the necessary rules to get everything else after that to go in its correct place again.

Extra challenge:

- Notice that the gap between the academic years is twice as large as that within years (e.g. the vertical space between COMS10013 and COMS20007 is twice that between COMS10014 and COMS10013).
- This can be achieved by leaving an empty grid row between the years, thus doubling the grid gap. That is, Year 1 lives in rows 1 and 2; row 3 is empty; Year 2 lives in rows 4 and 5 and so on - the empty row 3 has a height of 0 and no elements, but there are two gaps between rows 2 and 3, and 3 and 4, that separate the Year 1 from Year 2 units.
- Alternatively, you could add extra margin (why not padding?) to the bottom of all TB2/TB4 units to achieve the same effect.

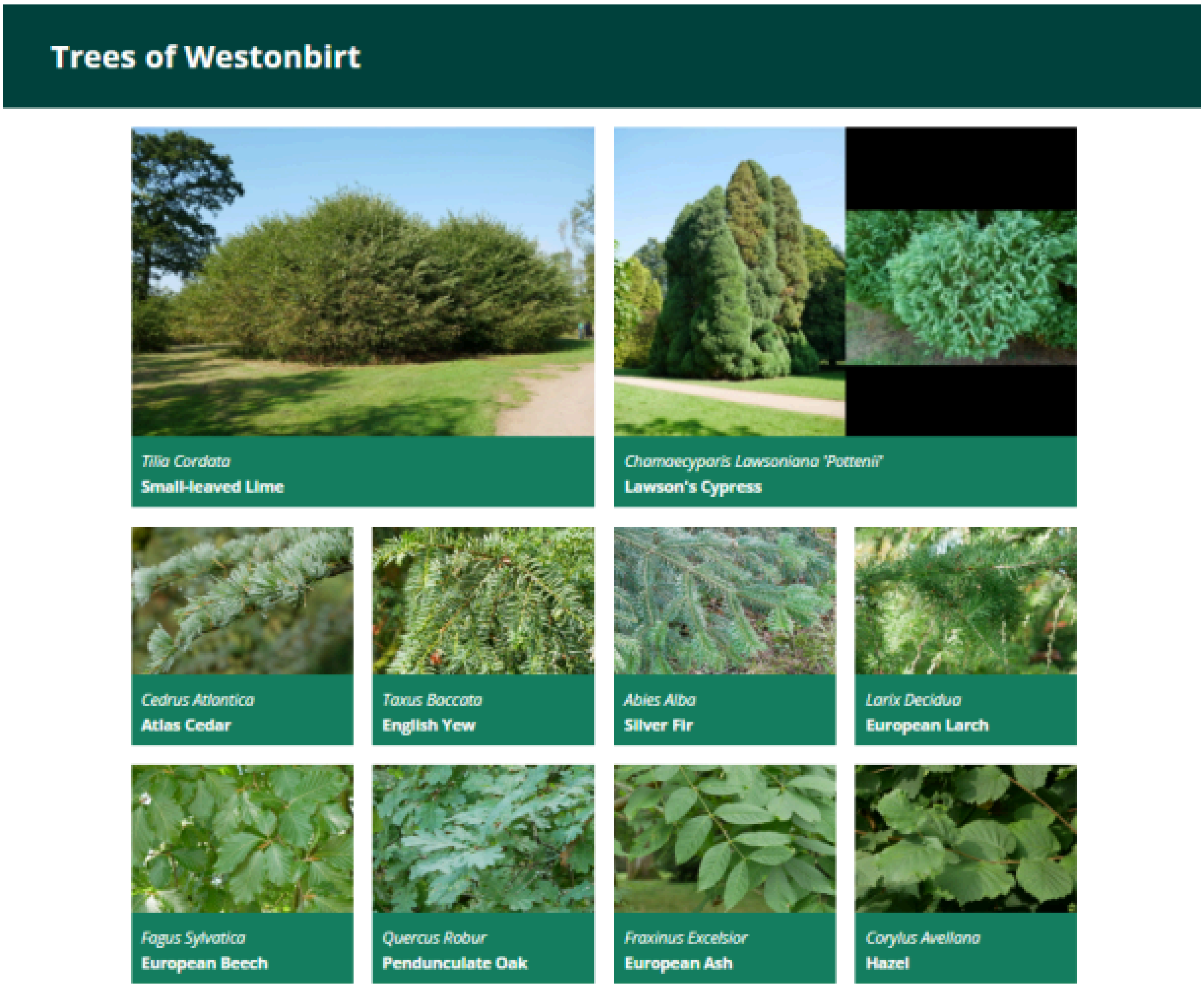
Responsive layout: trees exercise

Download the trees.tar.gz file for this exercise and extract it (`tar -zxvf FILENAME`) to a folder, then open `trees.html` in your editor. Create an empty `trees.css` in the same folder and open that too.

The file contains images and English and Latin names of some trees at Westonbirt, the UK national arboretum.

Your task is to write a stylesheet to create a responsive grid layout.

On media that is 600px or wider, the page should look like this:



Trees of Westonbirt

Tilia Cordata
Small-leaved Lime

Chamaecyparis Lawsoniana 'Pottenii'
Lawson's Cypress

Cedrus Atlantica
Atlas Cedar

Taxus Baccata
English Yew

Abies Alba
Silver Fir

Larix Decidua
European Larch

Fagus Sylvatica
European Beech

Quercus Robur
Pendunculate Oak

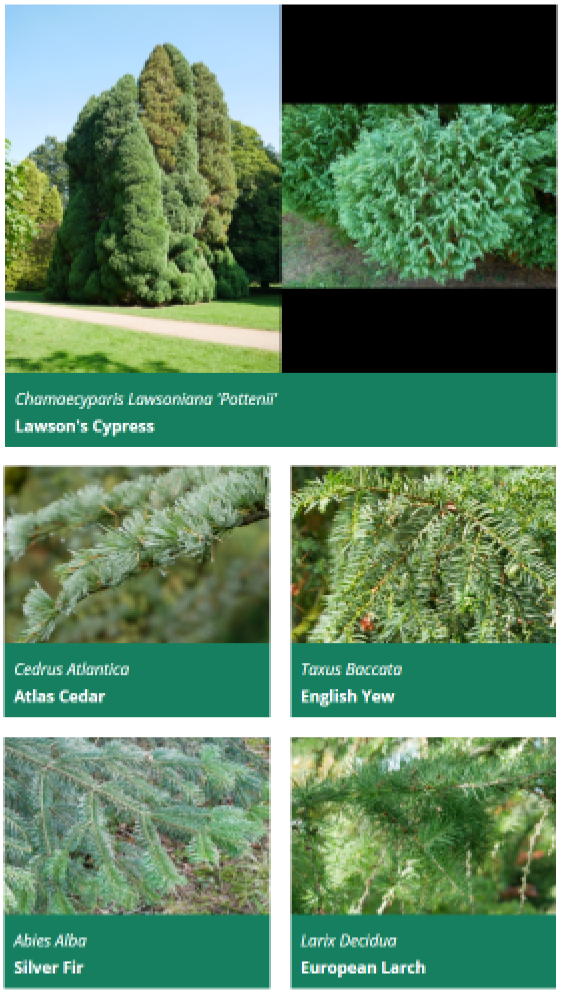
Fraxinus Excelsior
European Ash

Corylus Avellana
Hazel

(The image for Lawson's Cypress is one single PNG like the others, there is nothing special to do here to fit in two images side by side.)

The small-leaved lime (Tilia Cordata) featured here is believed to be over 2000 years old, making it one of the oldest trees in Britain.

From 400px to 600px, the page should show all trees except the "featured" ones in two columns:



And below 400px width, all trees should be shown in a single column layout, whether featured or not.

Each tree is defined in the HTML file with an element like this:

```
<div class="card">
    <img class="card-image" src="images/cedrus_atlantica.jpg" alt="Cedrus Atlantica"/>
    <span class="latin-name">Cedrus Atlantica</span>
    <span class="common-name">Atlas Cedar</span>
</div>
```

The two "featured" trees at the start also have class `featured` on their card element.

Exercise

General non-grid-related CSS that you can just paste in:

```
body {
    font-family: "Open Sans", sans-serif;
    margin: 0;
}

header {
    background-color: #04443c;
    color: white;
    margin: 0;
    padding: 10px;
    padding-left: 50px;
}

.card {
    background-color: #167f5f;
    color: white;
}

.card-image {
    max-width: 100%;
    height: auto;
}

.card span.latin-name {
    display: block;
    font-style: italic;
    padding: 10px;
    padding-bottom: 0;
}

.card span.common-name {
    display: block;
    padding: 10px;
    padding-top: 5px;
    font-weight: bold;
}
```

- The grid should be horizontally centered if the screen is wider than the grid, by setting `margin: 0 auto` on the container.
- The grid should be applied to the container div, with a margin of 20px on top and bottom (this has to come after the horizontal centering), and a padding of 10px on left and right (this has to be padding, not margin, to avoid undoing the centering again).
- Set a grid gap of 20px.
- If the screen is at least 600px wide, then the container should have a maximum width of 960px and four equally wide columns. Featured trees should take up a 2x2 space on the grid, all other trees a 1x1 space.
- Between 400px and 600px, the grid should only be two columns wide, trees still take up 2x2 space if featured and 1x1 otherwise.
- Below 400px, make a grid with only one column, and all trees take up 1x1 space. Note that you do not need to do anything for this as this is the default - just make sure that your other grid rules such as making featured trees 2x2 only apply on larger screens.

Test this out by making your browser window wider and narrower. You cannot normally make a chrome/edge window narrower than 400px, but there are two workarounds. With the F12 developer tools open, you can position them to the right of the window and then drag the divider to make the actual page width as narrow as you want, or you can switch to "mobile" mode (second icon in top left of developer tools, usually) and manually set a page size or select one from a dropdown list of common phone models.

Advanced note

In practice, one would use a layout like this to make the individual grid elements clickable (such as news stories, or in this case links to a more detailed page about the tree).

One way to do this would be to use `<a class="card" href="...">` elements instead of `<div>` ones, and set them manually to `display:block`. This has the advantage that the entire card, including the image, becomes clickable. An `<a>` tag can contain arbitrary content, not just text.

JavaScript

JavaScript is a programming language for creating dynamic web pages.

Videos

Video	Length	Slides
JavaScript: Basics	24 minutes	Slides
JS Objects (incl. JSON)	5.21 minutes	Slides
Object Oriented Programming	9.40 minutes	Slides
Asynchronous JavaScript	12 minutes	Slides

Some example code can be found in the `code/javascript` folder of the repository.

Readings

Essential

- MDN: Working with JSON
- MDN: Manipulating documents
- MDN: Asynchronous JavaScript
- MDN: Fetching data from the server (skip the section on `XMLHttpRequest`)

Further

- MDN: Arrow functions
- MDN: Document Object Model (DOM) (see guides on the left)
- Wat, a famous lightning talk by Gary Bernhardt (2012).

Advanced

The following references go significantly beyond the curriculum.

- MDN: Inheritance and the prototype chain. This is a guide to the protoyping-style object-orientation of JavaScript.

- MDN: The event loop. This explains the concurrency model of JavaScript, as well as the way asynchronous code and promises arehandled.
- Structure and Interpretation of Computer Programs, JS edition (2022). This is a classic programming book that has been recently rewritten in JavaScript.
- Boris Cherny, "Programming TypeScript" (2019). This is a book-length introduction to TypeScript, a typed extension to JavaScript which is also supported by React.

Exercises

A single-page application

This activity is about developing a very simple, single-page web application in HTML, CSS, and JavaScript. The point is to do this with basic JavaScript, i.e. using just the Fetch API and manipulating the DOM.

The purpose of the application is:

- to use the Bristol City Council API to fetch population data
- at the press of a button corresponding to each Bristol ward, present that data nicely in a table

Setup

Because you will need a browser to run this application, it is best if you run it on your actual machine. After all, the point of web applications is that they are (sort of) *portable*, i.e. they are able to run and present virtually identical behaviour on all sorts of machines. (This is not entirely true however, because different JavaScript engines may fail to support various language features.)

A basic version of the app can be found at `code/javascript/bristol-app` in the repository. It comprises three files:

```
app.html
style.css
script.js
```

We will go through each one in detail. But before we do so, point your browser at `app.html` and try the application.

Static structure

The static structure of our application is given by the `app.html` file, as styled by `style.css`.

HTML

The main html file `app.html` is surprisingly unassuming: it contains almost nothing of substance! In fact, we can quote the complete file here:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>The City of Bristol</title>
    <link href="https://fonts.googleapis.com/css2?
family=Libre+Baskerville:ital,wght@0,400;0,700;1,400&display=swap"
rel="stylesheet">
    <link rel="stylesheet" href="style.css" />
    <script src="script.js"></script>
  </head>
  <body>
    <header><h1>The City of Bristol</h1></header>
    <nav id="nav">
      <div class="initial">Initialising...</div>
    </nav>
    <main id="dataPane">
      <div class="initial">Please select one of the wards on the left to view
more data about it.</div>
    </main>
    <footer>created by G. A. Kavvos</footer>
  </body>
</html>
```

This is a very standard HTML file, which readily passes the W3 validator. It consists of a `<head>` and a `<body>`.

- The `<head>` sets the title, loads some nice fonts from the web, and declares that `style.css` is the stylesheet. Finally, the `<script>` tag tells the browser that there is JavaScript code to be run in the file `script.js`, which will be retrieved from the same location as the page that is being loaded.

Advanced note

If you look at introductory content on JavaScript on the internet, you will find many examples where JavaScript is included in a so-called *inline* style. For example, you might see code of the form

```
<script>alert("Hello World!");</script>
```

which embeds a fragment of JavaScript directly in the HTML file, instead of putting it in a separate `.js` file, and including it using an `src=` attribute like we did above.

Do not do this. This is very bad style. Always keep HTML and JS in separate files.

- The `<body>` describes a document with four sections:
  - There is a `<header>` with a big heading reading "The City of Bristol".
  - There is a `<nav>` navigation section, with a single `<div>` (i.e. a generic block container) with placeholder text saying "Initialising...".
  - Then then there is a `<main>` section, which contains a single `<div>` with some instructions for using the app.
  - Finally, there is a `<footer>` with the author's name in it.

The two `<div>`'s are instances of the `.initial` class. Both `<nav>` and `<main>` have unique id attributes, so they can be referred to uniquely.

Advanced note

It is generally agreed that id attributes must be unique in an HTML document. However, there is no enforcing mechanism for this: the browser will *not* check that we have not mistakenly re-used an id for some other tag. Again, this is in the interest in providing a smooth user experience without errors. Do not abuse this!

This sparse document structure will be all that we need to present our single-page application.

CSS

Go ahead and open up `app.html` in your browser of choice. You will see that the structure described by the tags above appear in very particular places.

Indeed, the location and appearance of the four sections (header, nav, main, footer) are set in the `style.css` file, which is explicitly loaded in `app.html`. The main things to note are the first few lines

```
html {
  font-size: 12pt;
  font-family: 'Libre Baskerville', serif;
}

body {
  display: grid;
  grid-template-columns: 200px 1fr;
}
```

which tell the browser to

- use the Libre Baskerville font (loaded in the `<head>`) throughout the entire document, and
- lay out the `<body>` of the document using CSS grids.

We will not dwell on the grid structure here, but note that, as the `<nav>` comes first, it becomes the first `200px` column, and the `<main>` section becomes the `1fr` column.

The `style.css` file also sets many other details, such as margins, colours, etc.

Dynamic structure, part 1

When you point your browser at `app.html`, the "Initialising..." string appears in the left of the screen (the `<nav>`) only for a fraction of a second. This is the time it takes to load and execute the code in `script.js`, which provides the dynamic behaviour of the app.

Making some buttons

Indeed, looking at `script.js` we notice that it goes straight into business:

```
window.onload = function () {
  let wards =
fetch('https://opendata.bristol.gov.uk/api/v2/catalog/datasets/wards/records?
limit=50&select=name,ward_id')
    .then(response => response.json())
    .then(populateWards)
    .catch(err => console.log(err));
}
```

This fragment of code creates a function, and assigns it to `window.onload`. This means that this function, which takes no arguments, will be called when the browser window has finished loading the page.

The body of the function makes an HTTP request to the Bristol City Council API. This returns a promise; when it is resolved, it is parsed as JSON, which itself creates another promise. Finally, when *that* promise is resolved, the `populateWards` function is called with the fetched JSON as argument. If any of these steps cause an error, the last line catches it, and prints it on the console (only visible if you press F12). These four lines are a modern JavaScript idiom that uses all the latest technology: the fetch API; promises; and higher-order, anonymous functions.

One might wonder: what will the input passed to `populateWards` in this call look like? To answer this we can peek at the above URL and see what it returns: on any Linux machine we may run

```
curl -X 'GET'
'https://opendata.bristol.gov.uk/api/v2/catalog/datasets/wards/records?
select=name,ward_id' | json_pp | less
```

which will make a `GET` request at this URL, parse the result as JSON (`json_pp`), and display it in scrollable format (`less`). The result looks a lot like this:

```
{
   "links" : ...,
   "records" : [
      {
         "links" : ...,
         "record" : {
            "fields" : {
               "name" : "Eastville",
               "ward_id" : "E05010897"
            },
            "id" : "996b607b4c31e6aca6a7614bd02ea18a4c14c525",
            "size" : 40500,
            "timestamp" : "2020-09-03T10:02:58.597Z"
         }
      },
      {
         "links" : ...,
         "record" : {
            "fields" : {
               "name" : "Southville",
               "ward_id" : "E05010914"
            },
            "id" : "bc2f85ccc34ca606a2fe6473491b5fc50fd4e0d1",
            "size" : 25544,
            "timestamp" : "2020-09-03T10:02:58.597Z"
         }
      },
      ...
   ],
   "total_count" : 34
}
```

I have abbreviated the `links` fields, which add a lot of noise to the output. We can thus see that this HTTP request returns a JSON object which contains the names and ward IDs of all the wards of the city of Bristol! The final field returns the total record count, which is 34.

Unlike SQL databases, which come with a strongly-typed data schema, the data here is semi-structured at best. We may discern its structure by looking at the above output. The top-level object has a `records` field, which contains an array of records. Each of these records is a JSON object itself. Its `record` field contains a field called `fields`, which contains the `name` and `ward_id` of each ward.

Advanced note

One might ask: how did I figure out the correct URL to obtain all the wards?

Unlike relational databases, where a predetermined schema tells the developer exactly where to look, the situation with APIs is more of a trial-and-error affair. Many APIs you will have to use in your life are poorly documented, and using them invovles some guesswork.

In this particular instance, I went on the Open Data Bristol website, and looked through the available datasets. There I found a dataset called 'Wards'. The description seemed to

> match what I wanted, which was confirmed by clicking on the 'Table' tab, and seeing some sample data.
>
> Using the 'API' tab on the same page is misleading, as it presents an interface for querying the old version (v1) of their API. Following the link to the 'full API console' reveals that there is a modern, REST-type API (v2), described in a format known as OpenAPI. This is the current industry standard for describing REST APIs.
>
> Looking through the documentation, it was evident that the endpoint of interest is
>
> ```
> /catalog/datasets/{dataset_id}/records
> ```
>
> I replaced `{dataset_id}` with the id of the ward dataset (`wards`). I also passed in two query parameters:
>
> - `limit=50` which limits the response to contain at most 50 data points in the JSON object (which is way more than the Bristol wards)
> - `select=name,ward_id` which limits the fields in the response to those in which we are interested
>
> To test this I used variations of the above `curl` command, and looked at the output.

Now that we understand the structure of the data returned from that endpoint, we can go ahead and write the `populateWards` function:

```
function populateWards(wards) {
  let buttons = new DocumentFragment();

  wards.records.forEach(w => {
      const [id, name] = [w.record.fields.ward_id, w.record.fields.name];
      const b = document.createElement("button");
      b.textContent = name;
      b.onclick = displayData(id, name);
      buttons.appendChild(b);
  });

  let nav = document.getElementById("nav");
  nav.textContent = '';
  nav.append(buttons);
}
```

This function is a callback that will receive the JSON object containing ward names.

The first line creates a `DocumentFragment` called `buttons` . Loosely speaking, this is an object that can be used to collect a bunch of stuff that will be added to a page. When creating many new elements on a page it is prudent to add them to a `DocumentFragment` first, and only then add it to the page. That way they will all appear on the page at roughly the same time, and the user will not see new elements appear on the page one after the other.

The next part of the function iterates through every ward. First, it extracts the `id` and `name` fields. Then, it creates a new `<button>`, and sets its text to be the name of the ward. When each of these buttons is clicked, the function returned by the call to `displayData` will be run. Each new button is then added to the `DocumentFragment` using the `appendChild` function.

Finally, the `<nav>` is obtained by using its ID. Its contents are deleted, and replaced by the `DocumentFragment` containing all these new buttons, by using the append function.

In short, this is the bit of code that creates the buttons on the navigation section on the left!

Advanced note

Of course, someone could argue that the wards of Bristol do not change very often, and that as a consequence it is slow and wasteful to perform an HTTP request to obtain a list of wards; it would be much better to simply hard-code these buttons and their names into the app. This developer could well be right.

Exercises.

1. Modify the code above so that each button is on a separate line.
2. Modify the code above so that buttons appear in alphabetical order.
3. The line

   ```
   b.onclick = displayData(id, name);
   ```

   sets a button's `onclick` property to be a newly-created function. Sometimes this is undesirable. For example, if another script had already added some code for what should happen when a button is clicked, this would completely replace it. Change this line so that it instead adds an `EventListener` to `b`, so that it simply adds functionality, without interfering with other code.

[*Hint.* Exercises 1 and 2 can be completed by adding one line of code for each. For the second one, use `Array.sort`].

Dynamic structure, part 2

The rest of `script.js` is concerned with displaying data about a given ward in the `<main>` section of the page. In fact, it consists of just one function, called `displayData`. Let us look at its structure, whilst eliding some of its implementation details:

```
function displayData(id, name) {

  function buildPopulation(records) {
    ...
  }

  return function () {
    ...
  }
}
```

When called with two arguments `id` and `name`, the function `displayData` does two things:

1. It defines a nested function called `buildPopulation`. This is a function whose definition is *local* to `displayData`: it can only be used within its body, but not in the rest of the program. However, this function is allowed to refer to the arguments of its enclosing function (i.e. `id` and `name`). We sometimes call this an *auxiliary function*, or a *helper function*.
2. It returns an anonymous arrow function. Thus, `displayData` will return... a newly defined, anonymous function! Recall that in the previous part of the script, `displayData` was used only in one line, namely

   ```
   b.onclick = displayData(id, name);
   ```

   `b` is a button, and its `onclick` property is a function that will be run when it is clicked. Thus, `displayData` should return a function. To construct such a function we need to know the `id` and `name` of the ward whose data is to be displayed, so `displayData` needs to take them as parameters.

Both of these constitute examples of functional programming in action.

Let us now look in more detail in the function that is returned:

```
function displayData(id, name) {

  function buildPopulation(records) {
    ...
  }

  return function () {
    let wards =
fetch(`https://opendata.bristol.gov.uk/api/v2/catalog/datasets/population-
estimates-time-series-ward/records?
limit=20&select=mid_year,population_estimate&refine=ward_2016_code:${id}`)
      .then(response => response.json())
      .then(data => {
        let heading = document.createElement('h1');
        heading.innerText = name;

        let population = buildPopulation(data.records);

        let dataPane = document.getElementById("dataPane");
        dataPane.textContent = '';
        dataPane.append(heading, population);
      })
      .catch(err => console.log(err));
  }
}
```

The pattern here is much the same as before: an HTTP request is made, its response is parsed as JSON, and then processed.

The endpoint is determined as before, but now asks for data from the `population-estimates-time-series-ward` dataset, which contains estimates of the population that lives in each Bristol ward. We extract only the `mid_year` and `population_estimate` fields, which contain the year in the middle of which the population is estimated, and the actual estimate itself. We also use a new feature of the API, namely the `refine` query parameter. This allows us to extract only records one of whose *facets* (i.e. fields of data) has a particular value. At this point in our code, we know the variable `id` holds the ID of the ward whose population we want to display, say `E05010914` for Southville. By peeking in some sample data, we can work out that if we pass the query parameter

```
refine=ward_2016_code:E05010914
```

we will only retrieve the population estimates for Southville.

The value `E050109141` is presumably stored in the `id` argument. However, `id` is a variable in our JavaScript program, so it needs to be turned into a string, which can then be spliced into the URL at the right position. Fortunately, modern versions of JavaScript provide a neat way of doing this, namely template literals. Instead of enclosing the URL string in single quotes as before, we now enclose it in backticks. Then, we can use any JavaScript expression `e` that evaluates to a string by writing `${e}` directly in the URL string. Thus,

```
`https://opendata.bristol.gov.uk/api/v2/catalog/datasets/population-estimates-
time-series-ward/records?
limit=20&select=mid_year,population_estimate&refine=ward_2016_code:${id}`
```

evaluates to the correct string at runtime.

The rest of the function is unremarkable. It creates an `<h1>` heading containing the name of the ward. Then, it calls the local function `buildPopulation`, passing the records in the data returned by the HTTP request. The `<main>` tag is retrieved through its unique ID, its contents are erased, and replaced by the heading and whatever was returned by `buildPopulation`.

A version of this function is run every time someone clicks a ward button.

Generating tables

The rest of the application is contained in the auxiliary `buildPopulation` function:

```
  function buildPopulation(records) {

    // Make heading
    let heading = document.createElement('h2');
    heading.innerText = 'Population';

    // Make table
    let table = document.createElement('table');
    table.setAttribute('id','populationTable');

    // Make table header
    let header = document.createElement('tr');
    header.innerHTML = '<th>Year</th><th>Population</th></tr>';
    table.appendChild(header);

    // Populate table
    records.sort((x1, x2) => x1.record.fields.mid_year <
x2.record.fields.mid_year ? -1 : 1)
      .forEach(r => {
        let year = document.createElement('td');
        year.innerText = r.record.fields.mid_year;
        let population = document.createElement('td');
        population.innerText = r.record.fields.population_estimate;

        let row = document.createElement('tr');
        row.append(year, population);
        table.appendChild(row);
    });

    let population = new DocumentFragment();
    population.append(heading, table);

    return population;
  }
```

Upon receiving an array `records`, this function sets out to build a new `<table>` with two columns, Year and Population. After building its header, it sorts the records by year. This is achieved through the build-in `sort` function, which takes a higher-order argument that decides which of two records comes first. It has a funny interface: negative numbers signify that the first argument comes before the second, and positive arguments the opposite.

The program then iterates through the records, and makes a row `<tr>` for each data point. Finally, it assembles this data into a table, prepends an `<h2>` heading, and returns a `DocumentFragment` consisting of the heading and the table.

Thus, this part of the application is responsible for retrieving and presenting the requested data.

Exercises

Exercise (easy). Make the app present only population estimates after the year 2015.

This may be achieved in two ways:

- by changing the REST API call; or
- by processing the data after it has been fetched.

The second one is significantly easier.

[Hint: Moreover, the second one can be achieved with one line of code if you use Array.filter.]

Exercise. Expand this application so it also presents an estimate of life expectancy for each ward. I recommend that you define an auxiliary function

```
function buildLifeExpectancy(records) {
  ...
}
```

which, given the data, builds the necessary HTML elements - as above.

You will also face another problem, which is that you will need to use data from two HTTP requests. The obvious way to do this is to nest two fetch calls:

```
fetch('first-URL')
  .then(response => response.json())
  .then(data1 =>
    fetch('second-URL')
      .then(response => response.json())
      .then(data2 => {
        // ... here you can use both data1 and data2 ...
      });
  );
```

However, recent versions of JavaScript provide a neater way of doing this, namely `Promise.all`. This function accepts an array of promises, and returns a single 'composite' promise. When all the premises in the array are resolved, it passes an array of the returned values to its `then()` clause. In this particular instance, you could do something like

```
Promise.all([fetch('first-URL'), fetch('second-URL')])
  .then([response1, response2] => Promise.all([response1.json(),
response2.json()]))
  .then([data1, data2] => {
    // ... here you can use both data1 and data2 ...
  });
```

or, in a more succinct style:

```
Promise.all([fetch('first-URL'), fetch('second-URL')])
  .then(responses => Promise.all(responses.map(r => r.json())))
  .then([data1, data2] => {
    // ... here you can use both data1 and data2 ...
  });
```

Simplifying your code

If you have changed the app so that it displays life expectancy as well, you will notice that you have amassed a formidable amount of code. For example, my solution runs to 116 lines - all to achieve something that is essentially simple.

For example, to create the population data table we wrote the following code:

```
let year = document.createElement('td');
year.textContent = r.record.fields.mid_year;

let population = document.createElement('td');
population.textContent = r.record.fields.population_estimate;

let row = document.createElement('tr');
row.append(year, population);
table.appendChild(row);
```

Staring at this code, certain patterns become evident: every time we want to create a new HTML tag, we must:

1. create the HTML element,
2. set its `textContent` (or `innerHTML` elsewhere), and
3. assemble things hierarchically (in rows, tables, etc.)

This pattern occupies the majority of our code. It is an artifact of the way the DOM API is designed. For example, it would be preferable if we had a constructor `document.createElementWithText(tag, text)` which allowed to construct a new element of the DOM, and immediately initialize it with some text.

Luckily, JavaScript is a particularly expressive language, and such an interface can be defined:

```
function createElementText(tag, text) {
  let elem = document.createElement(tag);
  elem.textContent = text;
  return elem;
}
```

This small function performs steps 1 and 2, and then returns the new element.

To deal with 3, i.e. the hierarchical assembly of elements, we can define a function

```
function createElementWith(tag, xs) {
  let elem = document.createElement(tag);
  for (const x of xs) {
    elem.appendChild(x);
  }
  return elem;
}
```

This function takes the name of a tag, and an array `xs` of elements. It then creates a new element with this tag, and appends everything in `xs` as children of this new element. Finally, it returns the newly constructed element.

These functions allow us to create new elements of the DOM in a more functional style. For example, an array containing the rows of the population table can be created using the following code fragment:

```
  records.filter(d => d.record.fields.mid_year >= 2015)
    .sort((x1, x2) => x1.record.fields.mid_year < x2.record.fields.mid_year ?
-1 : 1)
    .map(r =>
      createElementWith('tr', [
        createElementText('td', r.record.fields.mid_year),
        createElementText('td', r.record.fields.population_estimate)
      ])
```

The expressions in the body correspond to the actual tree structure of the row. As a result, they are much more readable.

Exercise. Rewrite the application using the functions defined above (and similar ones). Try to make it as succinct and readable as possible.

For example, my version of the complete application (with life expectancy) runs to 114 lines, and that is without making much of an effort to be succint. This is only 2 lines shorter than my previous version, but one must remember that about 20 of these lines are the new functions, and we could factor these out in a separate JavaScript module. This amounts to a 19% reduction in the amount of code - which can be rather significant in larger projects.

Of course, larger projects are likely to use frameworks like React to build such applications, whose use of JSX and function components deal with exactly this sort of problem.

Web Scraping

Videos

Video	Length	Slides
Crawling	16 minutes	slides
BeautifulSoup	10 minutes	slides

Exercises

- Crawling
- BeautifulSoup

Crawling the Web

So far in this unit we've had you use `wget` only for one of its simplest use-cases: when you want to download a single file from the web. Consistent with the Unix philosophy that tools should 'do one thing well', `wget` is capable of a lot more than this.

To demonstrate this, we're going to have you first run a server to deploy a website locally, and then test out various `wget` options by connecting to that server via localhost. This of course doesn't mean that `wget` can only do these things via localhost -- it's designed to work with real websites -- but we decided that getting ~200 students to test out web-crawling on a particular live website was probably not a great idea, so we're having each of you run your own.

Set up the server

As with the HTTP exercises, it would be best to either carry out these steps directly on a lab machine or on your own machine (optionally from the Debian Linux VM with ports forwarded). If you are using a lab machine via SSH instead then you'll need to open a second SSH session *to the same lab machine* in another terminal, to act as a client for the next parts of the exercise. However, we'll periodically invite you to check things in your browser, which is a lot simpler if you're not trying to use a remote machine.

First, download the webpages to be served by the webserver. If you like you can even do this using `wget`:

```
wget https://cs-uob.github.io/COMSM0085/exercises/part2/resources/cattax.tar.gz
```

Then extract the contents of the tarball using `tar -xzf cattax.tar.gz` in the folder you downloaded it to. This will create a folder `cattax` which contains some webpages and resources.

Next, use the `darkhttpd` server from a previous week's exercises to serve the content of the `cattax` folder on `localhost:8080`. *(Refer to the HTTP week's exercise instructions if you have forgotten how to do this)*.

You can check that this is working in a browser (unless you are connecting via SSH) by navigating to `localhost:8080/index.html` -- you should see a webpage talking about **Felidae**. You'll need to leave this server running -- the simplest way forward would be to open another terminal for the steps below. *(Alternatively: use your shell expertise to figure out how to run the server in the background without its output interfering with the rest of what you're going to be doing in this terminal).*

Single-page download

To keep your filesystem tidy, we're going to work within a 'client' folder. We'll be repeatedly downloading files and sometimes deleting them, and you'll neither want lots of duplicated webpages littering your filesystem nor want to run `rm *` in a directory that possibly contains files you don't want deleted.

Make sure you are in the parent directory that contains `cattax` (i.e., you can type `ls` and see the directory `cattax` in the output) and *not inside* `cattax` itself. Then create a directory and move into it:

```
mkdir client
cd client
```

Now we'll start with the simple use of `wget` you have already become familiar with:

```
wget localhost:8080/index.html
```

This downloads the same `index.html` as is being served from `cattax` by `darkhttpd`. However, if you open this downloaded file in your browser, you'll see that there's a sense in which something missing -- `wget` has only downloaded the specific HTML file you requested, and not any of the resources that the page itself references, like the CSS file -- so the version you open in your browser from your `client` directory won't look the same as the version being served via localhost. This can be desirable default behaviour (we only asked it to get that page, after all), but if we wanted to download a copy of a webpage and later read that webpage's copy with the styles and images it was originally created to contain, we'd need to get `wget` to also download these resources.

One way to do this would be to manually identify each of the required resources and download them one-by-one. But this is tedious, repetitive work -- highly suited to automation -- and moreover, `wget` can save us the effort. Try the following, and **read the output**.

```
wget -p localhost:8080/index.html
```

Notice that this time `wget` downloaded multiple files. It also created a directory named `localhost:8080` to store all the files in. This is helpful organisation if you're ever using `wget` to download pages from multiple different websites -- it stores them under directories named after the domain you requested them from.

If you read the output carefully you'll notice that as well as the `index.html` we requested directly, `wget` has also downloaded the `catstyle.css` file referenced in that page, and another file called `robots.txt` that you didn't ask for and which isn't mentioned in `index.html`. This 'robots' file is part of a standard for responsible web crawling, and tells crawling tools which parts of a website they are or aren't allowed to visit. When you use `wget` to crawl a webpage or website it will check the site's `robots.txt` to understand which resources it may not be allowed to download. You can read more about how these files are written here.

Open the `index.html` file from the new `localhost:8080` folder that was created, and you should see that it looks just like the version you got in your browser by navigating to the URI `localhost:8080/index.html`. *(There are some cases where this wouldn't be true for a webpage -- wget can sometimes not be permitted access to some resources required to display a page the same way as it is shown in your browser).*

Crawling a site

The version of the webpage you downloaded using the previous command still has one major flaw: the links on the page don't work. Or, rather, the links to the 'Felinae' and 'Pantherinae' pages are broken, because those links are made relative to the webpage, and the corresponding files don't exist in the client's folder. The link to Wikipedia in the page footer *does* still work, because the `href` attribute of that link is set to a full URI.

What do we do if we want to download more than one webpage from a site? Wget supports something called 'recursive downloading'. Simply put, when used in this manner it will follow all links internal to a site and download the resources displayed at those links, storing a copy locally and creating a directory structure if necessary. One version of this recursion is to use the `-r` (or `--recursive`) option, which downloads all linked pages up to a certain maximum depth. Try this out:

```
wget -r -l 1 localhost:8080/index.html
```

This downloads recursively with the 'level' (maximum depth of recursion) set to 1 level of recursion. You should see that both the requested `index.html` and the two pages linked from that resource have been downloaded, along with `robots.txt`. Notice as well that the Wikipedia page has not been downloaded -- it's not hosted at `localhost:8080`, so `wget` ignores it, and the link will work from the downloaded page anyway. Our two newly-downloaded pages, however, will contain dead links to other pages, because we limited the depth of recursion to just one hop. If we increase this:

```
wget -r -l 2 localhost:8080/index.html
```

You'll see that a *lot* more files get downloaded. These are only very small, simple web-pages, without many links (contrast, for example, any given Wikipedia page). Very short recursion depths can capture an awful lot of a domain, and if you ever tell `wget` to crawl links without caring about which domain they belong to, this becomes explosively worse (`-l 2` in such a case for our `index.html` would involve downloading everything linked from the Wikipedia page referenced in the footer -- several hundred resources). In the case of our cattax website, however, there are still a few pages that are more than 2 steps away from the index page. Let's start afresh:

```
rm -r localhost:8080
wget -m localhost:8080/index.html
```

The `-m` flag is designed to provide some sensible defaults for 'mirroring' an entire website (something you might do if you wanted to keep a copy of it for offline browsing, or for providing a public backup of a valuable resource). It sets the recursion level to infinite and checks timestamps before downloading files, as well as setting a few more configuration settings. For many cases where you might want to download an entire website, this is the flag you would use -- perhaps also with a polite `-w 1`, which sets a 1-second delay between requests, to avoid over-burdening the server if the website is large.

Further Exercises

1. Read `man wget` to understand what the `-i` `--force-html` and `--spider` options do. Download a copy of this webpage (the one you are currently reading) and use `wget` to test all the links on the page. Are there any broken links?
2. Tell `wget` to use a different user agent string in a request to your server running on localhost. Check what the request looks like to your server.
3. How would `wget -r -l 1 http://example.com` differ from `wget -p http://example.com`? *(Hint: think about external resources)*.
4. Look for 'Recursive Accept/Reject options' in the `wget` manpage. How would you get `wget` to crawl pages from multiple different domains?
5. Look up what `-nc` does. What is clobbering, and why would or wouldn't you want to do it?

BeautifulSoup

Previously in this unit we have taught you how to construct webpages, including how to construct webpages so that they present structured information (e.g., the contents of a database). The presumption has generally been that you are designing webpages for (usually visual) consumption by a human using a standard web browser, or at most writing Javascript functions that will help a human use a webpage in particular way.

However, there are situations where you may need to write code that interacts with webpages in an automated fashion -- perhaps to extract some data from webpages that isn't available in a more machine-friendly format. This *scraping* of webpages is a valuable (if sometimes maliciously applied) skill that can be useful to many kinds of software engineers and data scientists. This exercise talks you through some applications of one of the most common frameworks, a Python library called BeautifulSoup.

Setup

First, you want to install the BeautifulSoup library, which you can achieve with `sudo apt install python3-bs4`. Python does have its own package manager, `pip`, but it would get in the way of our system-wide package management to use it here.

You can test that Python is working by just typing `python3` at the command line -- it should launch the interactive interpreter, which will prompt you for Python code with `>>>`. Type `from bs4 import BeautifulSoup`. If this completes then you've successfully imported the library.

Interacting with Soup

Next, enter something like the following:

```
file = "cattax/index.html"
soup = BeautifulSoup(open(file, 'r'))
```

(You may need to change the 'file' line to reflect where the `cattax/index.html` file really is relative to where you launched `python3`. Also, if you close the interpreter at any point remember you'll need to re-run the `import` line above to get the library).

You now have a 'soup' object, which is a Python object that has various methods for interacting with HTML (and XML) documents and accessing their contents. To start with, just type `soup` in your interpreter. Python will print out the basic textual representation of the object, which is just the source of the `index.html` page. Next, let's attempt one of the commonest use-cases for scraping.

```
soup.get_text()
```

You should see a string containing the textual content of the webpage. If you call `print` on the result, you should see that the text has been laid out in a fair approximation of how the visible text would appear on a webpage.

```
text = soup.get_text()
print(text)
```

Getting the visible text out of a page is a common requirement if, e.g., you were going to use the webpage as input to an NLP system. You can also access non-visible portions of the page -- `soup.title` will give you the title element of the webpage, and `soup.title.text` will get you the textual content within the title element. Note the difference: `soup.title` is a BeautifulSoup element (of type `Tag`), and has methods associated with being a tag; `soup.title.text` is just a string, and only has methods that apply to strings.

As HTML documents are structured, you can interact with them in a structured manner. `soup.head` will get you a soup object reflecting the 'head' part of the HTML structure, and `soup.head.findChildren()` will return a list containing all of the 'children' elements inside the head. By knowing the structure of the document you can thus navigate to certain elements programmatically. This doesn't just relate to the tags, either: you can also access the value of attributes. `soup.head.meta['charset']` would access the `charset` attribute of the meta tag in the head of the document.

Exercises

1. Take a look at some other examples from the BeautifulSoup documentation, in particular regarding the use of the `.find_all()` method.
2. Use your interpreter to access a list of all the `<strong>` elements in the webpage, and figure out how to print out just the text contained within them.
3. How would you use `.find_all()` to find all `<div>` elements with a particular class value (e.g., 'container')? Would your method work if the div had multiple classes?

A Scraping Script

Download this python script to the directory that contains your `cattax` folder (*not* into `cattax` itself). On the command line, you should be able to run this script with `python3 scrape.py`. You'll see that it prints out a series of lines related to all the files in `cattax`.

Open `scrape.py` in an editor and inspect what it is doing. The script imports two libraries, one is BeautifulSoup and the other is `os`, which allows the script to use certain operating system functions. `os.listdir` is then used to list the contents of our `cattax` directory and iterate over them. We filter the filenames by checking to see which of them end with the string 'html', and if they do then we open the file and parse it into a BeautifulSoup object. Then, we print out text from certain elements, separated by a ':'. Understand how this works, asking a TA or lecturer for help if you aren't sure.

Exercises

1. Modify `scrape.py` so that it *also* prints out the contents of the 'info' paragraph in each page (this can be a second print statement). Run the script again to test that it works.
2. Currently the script prints something for *every* page. Modify it so it would only print something out for the leaf nodes -- those pages that don't have a 'container' element of their own.
3. Printing things out can be useful, but often we want to store values we scrape for later programmatic work. Rather than printing out information, create and update a Python dict for all leaf nodes where the dictionary keys are the titles of a page and the values are the corresponding content of the 'info' box. *Run your script with `python3 -i scrape.py` and it will execute your script and then place you in an interactive session immediately following your script's execution. You can then check that the dictionary's contents are what you expect by interacting with the dict object in your interpreter.*

When to scrape

As we discuss in this week's lectures, scraping webpages can come with some risks, depending upon how you go about it and what you do with the results. Guidelines you should follow include:

- Always respect the robots.txt presented by a site. Resources denied to crawlers may be protected for a legal or ethical reason.
- Always look for an API first. If a site will provide you with a JSON endpoint to query for the structured data you want, it is both easier and more polite to use that API instead of scraping the same data from webpages designed for human consumption.
- Be very careful about any scraping behind an authenticated log-in, such as from a logged-in social media account. Think about the privacy expectations people have about the content you are collecting, both in legal and ethical terms. If you post publicly something that was intended only for a limited audience, you could be betraying a confidence, and might even face legal repercussions.
- Generally beware that being able to access and read information from the web does not mean you are permitted to republish it, even in a modified form.

Further reading

You may have noticed that we downloaded our webpages with `wget` in the previous exercise, and then dealt with their content using Python in this one. If you were writing a Python tool that was meant to do something specific with the content of a webpage, then you would often want to avoid using a second tool, and make web requests directly from your program. The Python library that is most often recommended for requesting web resources is the Requests library, and getting familiar with it would be useful both if you wanted to access web pages programmatically and if you need to interact with a website's API.

We've also been dealing with the content of *static* websites, without the dynamic content like that you learned about in the Javascript exercises. Scraping dynamic sites gets trickier, as your scraper needs to have a browser-like context to execute Javascript within, and often might need to pretend to interact with the page. Systems like Selenium are designed for cases like this.