Overview

This page contains the exercises for Part 2 (the last 5 weeks) of COMSM0085 Software Tools.

As with Part 1, each week contains two 'activities' -- sets of exercises which are supported by videos to watch (linked from the main page for each activity). The exercises within each activity will require you to do some work within your Alpine Linux VM in Vagrant, which we'll be supporting in the labs.

Note that the 'Complete Application' material delivered in Week 23 is not examinable, though it may be practically very useful to you in future projects.

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
```

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 320 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. On alpine, you need to do the following if you have not done so already:

- Install the packages `openjdk17` and `maven`.
- Add the line `export PATH="$PATH:/usr/lib/jvm/java-17-openjdk/bin/"` to your `~/.profile` file.
- Run `source ~/.profile`.

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.

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.
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:

```css
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:

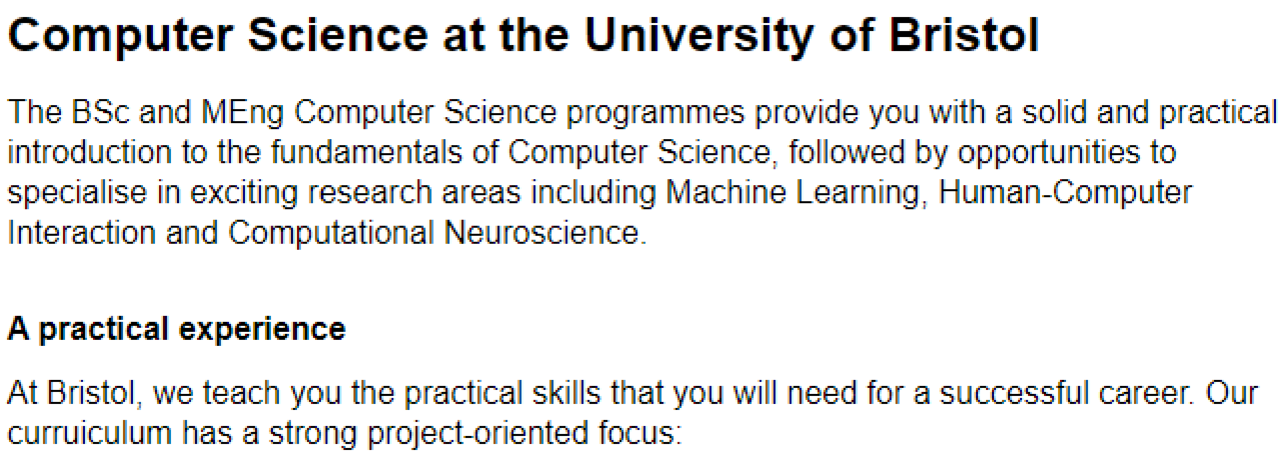
```
document.body.style.fontSize="24px";
```

(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.

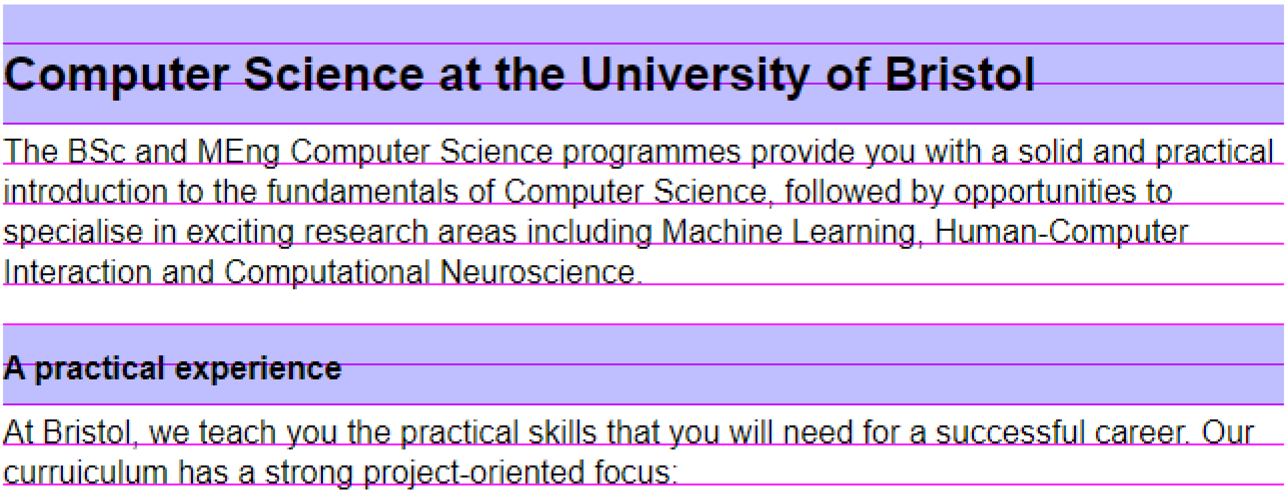
A practical experience

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:

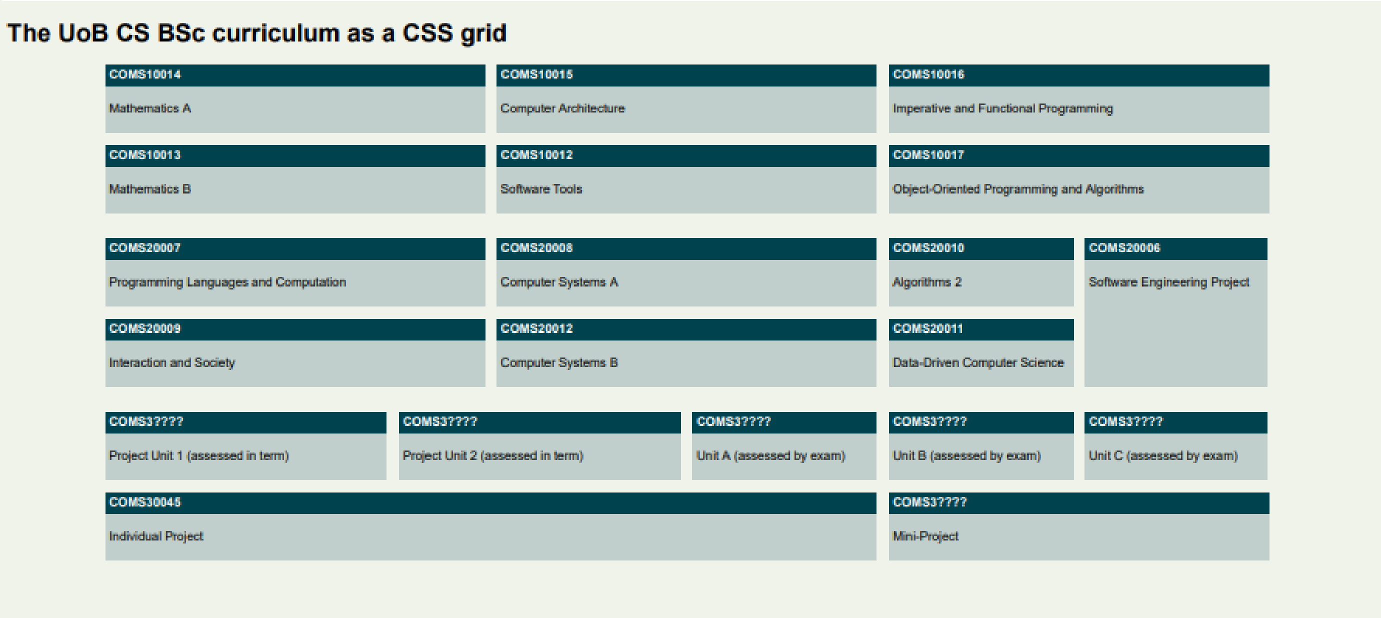
- Grids on MDN
- CSS-tricks complete guide to grids

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.

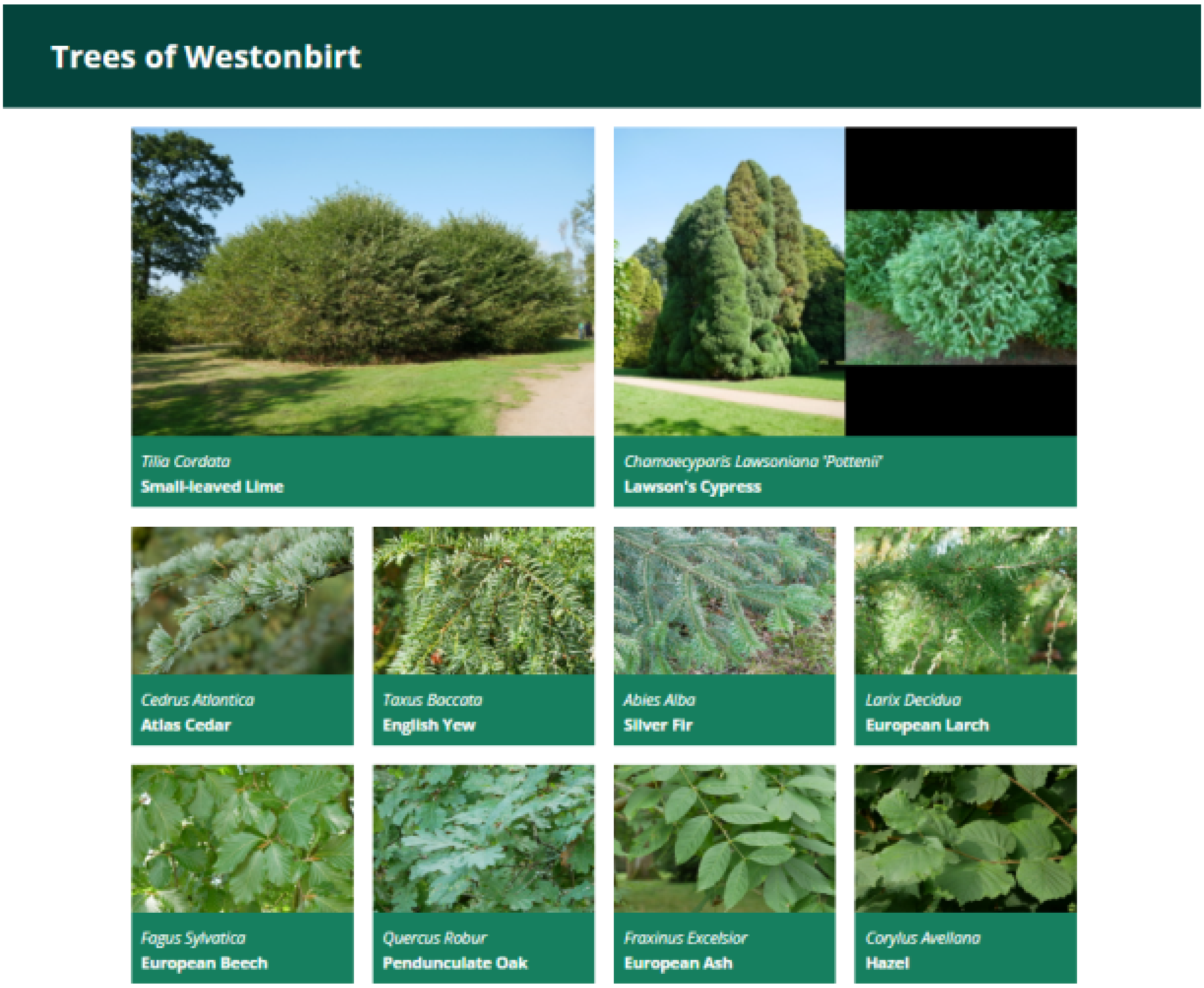
Responsive layout: trees exercise

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.

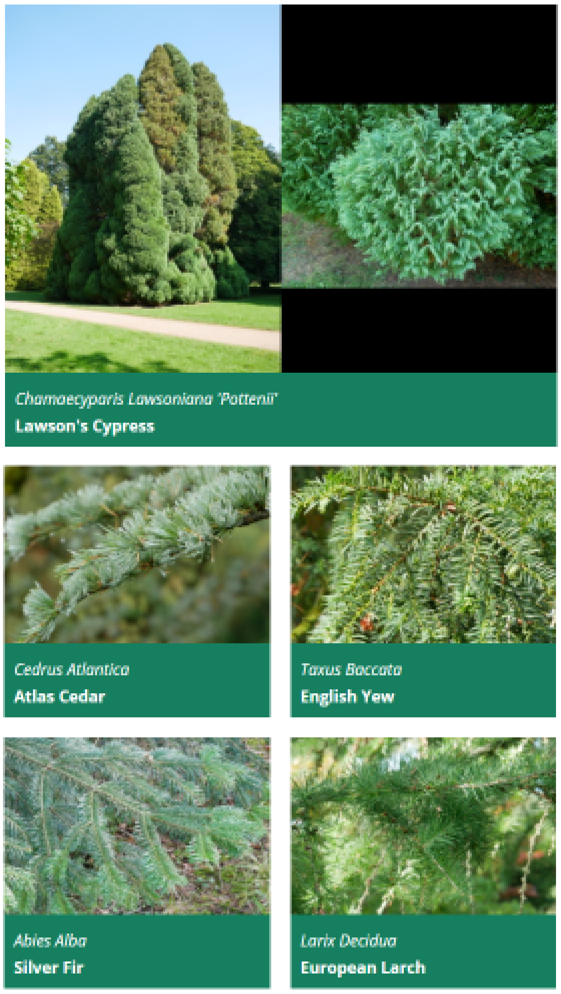
On media that is 600px or wider, the page should look like this:



(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 app
- Static structure
- Dynamic structure, part 1
- Dynamic structure, part 2
- Further 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 Alpine 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 -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'll need to install `python3` and `py3-pip` if you didn't already install those packages as part of the Build Tools exercises in Part 1.

Next, you want to install the BeautifulSoup library, which you can achieve with `sudo pip install bs4` (or just `pip install bs4` to use the user installation mode).

You can test that python is working by just typing `python` 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 `python`. 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 `python 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' paragrah 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 `python -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.

React

React is a JavaScript web development framework built by Facebook.

Videos

Video	Length	Slides
React	14 minutes	slides
React Part II	9 minutes	slides
React Part III	9 minutes	slides

Readings

The main reading is the official React guide.

Exercises

- Installing React
- Hello World in React
- Character Design

Installing React

Nodejs and NPM

JavaScript's original selling point was that it runs in the browser. NodeJS is a JavaScript runtime without a browser, so you can for example write a server / back-end in JavaScript too. Using a library called Electron, you can even write desktop apps in JavaScript/NodeJS (Electron is essentially google's Chrome browser without the branding, so you can use it to build your own user interface.) We will need NodeJS to run React, which compiles JSX code into JavaScript and HTML.

NodeJS comes with its own package manager, NPM (NodeJS Package Manager). The shell command to run it is `npm`, but there is also `npx` which downloads the latest version of a package and immediately runs it, if it contains a command-line tool.

- Install NodeJS for your operating system. On alpine, use `sudo apk add nodejs npm`; for other operating systems you can download it from nodejs.org or use your system's package manager, if available.

Your first React app

- Create a new react app with `npx create-react-app FOLDERNAME`. You cannot call the folder `react` itself, so please choose a different name. Downloading and preparing everything for the first time might take a while.
- By default, react uses port 3000 for its server. If you want another port, like 8000, edit `package.json` and edit the line

```
"start": "react-scripts start",`
```

to read as follows

```
"start": "PORT=8000 react-scripts start",
```

- Start the react app with `npm start` inside the folder with the `package.json` and open your browser on `localhost:8000` or whichever port you are using (the default is 3000). You should see a message "Edit `src/App.js` and save to reload." in your browser.

Explore the Code

Have a look at the structure of the project files:

- `package.json` is the configuration file (like the `pom.xml` for Java/maven).
- The page being served is `public/index.html` although the server replaces `%PUBLIC_URL%` itself (in development this just becomes the empty string). Note the contents of the page are essentially `<div id="root"></div>`, but if you look at the source in the browser (Control+U) you will see that react has added a line at the end of the body pulling in the JavaScript files. If you look with the developer tools (F12) you will see that the content of the app has been added as a child of the root div.
- The code lives in `src/`, mainly in `src/App.js`. The default code uses a function style component rather than a class style one, but the elements you see in the JSX code here (the HTML-embedded-in-JavaScript) are the ones that you saw in your browser inside the root div.

Hello World in React

In this exercise we will build a simple React app that greets the user, and so demonstrates managing state and passing it between components.

Create a react app folder with `npx create-react-app FOLDERNAME`. Edit the port if necessary, as described on the previous page.

You can start the development server (`npm start`) and it will automatically reload when it detects that you have edited a source file. This feature can be quite convenient to develop with.

We will create an app with three components:

- The app component manages some state, namely a person's name.
- The input component lets you enter and change your name.
- The output compoment displays "Hello, NAME".

A basic app

First, strip out the sample code so your `App.js` file looks like this:

```
import './App.css';
import React from 'react';

class App extends React.Component {
  render() {
    return (
      <p>App</p>
    )
  }
}

export default App;
```

Save the file - if the `npm run` server is running in another window, it will reload - and check that you get a page with just the text `App`.

Add some state

In the app class, we can add some state such as a name field:

```
import './App.css';
import React from 'react';

class App extends React.Component {
  state = {
    name: 'David'
  }

  render() {
    return (
      <p>Hello, {this.state.name}!</p>
    )
  }
}

export default App;
```

Check that this now displays `Hello, David!`.

Pass state between Components

Let's make a separate component to display the name - add this class before the `export default` line:

```
class NameGreeter extends React.Component {
  render() {
    if (this.props.name === "") {
      return (
        <p>Hello!</p>
      )
    } else {
      return (
        <p>Hello, {this.props.name}!</p>
      )
    }
  }
}
```

This component expects one property (the react version of a "parameter") called `name` and displays either a generic greeting or a greeting with a name.

Change the render method for the App class to this:

```
render() {
  return (
    <NameGreeter name={this.state.name} />
  )
}
```

This tells react that when it's time to render the App, it should create a `NameGreeter` component and pass it one prop called `name` which has the value of the name in the current state.

Note how we refer to `this.state.name` in the component that owns the state, but `this.props.name` in the child component. Generally, `this.state` is a component's own state and `this.props` is state (or other things) that a component gets from its parent.

You could try making more than one name greeter, but a JSX block must always have a single root tag so this won't work:

```
// don't do this
return (
  <NameGreeter name={this.state.name} />
  <NameGreeter name={this.state.name} />
)
```

But this is fine:

```
return (
  <div>
    <NameGreeter name={this.state.name} />
    <NameGreeter name={this.state.name} />
  </div>
)
```

Change State

We are going to make a `NameEditor` component with an input field to change the name. However, the name *belongs* to the app component: this is the *state lifting* pattern in React, where if more than one component wants to 'share' some state, then you make a common parent component to hold the state. Any changing of the state has to happen on the parent component that 'owns' it, so we have to create functions to pass it around correctly.

First, create a new component like this:

```
class NameEditor extends React.Component {
  render() {
    return (
      <p>
        <label for="name">Name: </label>
        <input type="text" id="name" value={this.props.name} />
      </p>
    )
  }
}
```

And change the render method of the `App` component to create a name editor:

```
return (
  <div>
    <NameEditor name={this.state.name} />
    <NameGreeter name={this.state.name} />
  </div>
)
```

When you run this, it will show you an input box with the current name, but if you try and change it then React will change it straight back again.

Input and output in React works with *data binding*: react maintains the correspondence between items on the screen (such as the input field) and its own state, but by default this only works in one way: React makes sure that the components on the screen reflect React's own state.

To make the edit box work, we have to first create a function on the `App` component that allows the state to be updated:

```
class App extends React.Component {
  constructor() {
    super();
    this.onNameChange = this.onNameChange.bind(this);
  }

  onNameChange(newName) {
        this.setState({name: newName})
  }

  // state and render method here, same as before
}
```

The real work is happening in `this.setState`, we can't just do `this.state.name = newName` as that would not trigger React's updates. The `setState` function which is part of React, apart from setting the new state, also re-renders any component that needs to update itself.

The constructor is just boilerplate code to correctly re-bind the `this` variable for `onNameChange`, you have to do this for every method you write in a React class. You also need to call the `super()` constructor explicitly at the start of the constructor, as this calls the `React.Component` constructor to set the component up correctly.

We also have to let the name editor know which function to call to process changes. Change the JSX line that creates it to

```
<NameEditor name={this.state.name} onNameChange={this.onNameChange} />
```

We did not put brackets after the function name, as we don't want to call the function when the component is created or rendered, we just want to tell it "here's a function that you can call later when you need it".

Next, in the `NameEditor` class, we create a method to handle updates:

```
constructor(props) {
  super(props);
  this.onNameChange = this.onNameChange.bind(this);
}

onNameChange(e) {
  this.props.onNameChange(e.target.value);
}
```

If our component takes props, then we should pass them to the super constructor too.

The parameter `e` is an element of type event, which is triggered by the input component when the user changes its value. The event class has a property `target` which refers to the element on the page that triggered it, and if that was an input box like it is here then we can read its `value`. This is what we pass, via `props`, back to the `App` component.

Finally, change the line that creates the input box to

```
<input type="text" id="name" value={this.props.name} onChange=
{this.onNameChange} />
```

The `onChange` attribute of an `<input>` is defined in the HTML standard and takes a JavaScript function that should be called whenever the input value changes.

With all this in place, you should be able to run the app and edit the text box, and watch the greeting change as you do this.

Sequence of Events

Make sure you understand what happens when you type in the input box. You can observe this yourself if you set breakpoints in the F12 debug tools:

1. You type in the input box.
2. The browser reacts by calling the input box's `onChange` function with one parameter, an `Event`. This function is `NameEditor.onNameChange`.
3. Your `NameEditor.onNameChange` calls `App.onNameChange`.
4. `App.onNameChange` calls React's `setState`.
5. In response to this, React calls `render()` on the `App`.
6. This triggers further `render()` calls in both `NameGreeter` and `NameEditor`.

Build the app

When you have finished developing an app, run `npm run build`. This will create your app in `build/`, compiling the JSX to JavaScript and HTML files. You can open `build/index.html` in a web browser without a server running.

Read all about it

We have essentially, with a different example, done step 10 of the official React tutorial. After the workshop, you may want to read this page again.

Character design exercise

The final React exercise today is to write a "character design" React app. Functionality is your first priority, before design: it's ok to lay out everything in simple paragraphs one above the other.

The constraints are:

- There are four stats, called *Charisma, Prowess, Agility* and *Strength*.
- You have 4 stat points to spend in total, but you may not spend more than 2 points on the same stat.
- You should make a reusable "stat editor" component that displays the points allocated to a stat and has +/- buttons to change these, and displays the name of the stat.
- The buttons must be enabled when you are allowed to increase/decrease the stat in question, and disabled otherwise.
- You should also make a separate component that displays the number of stat points remaining to spend (this was not in the example in the lectures).

You can ignore the character's name for now.

The design question here is where to put the functions that access or manipulate state. For example you will probably want a function somewhere that returns the number of stat points remaining to spend.

You will also need to think about how you can make a *reusable* stat editor component, so that you can include it four times in the app's render method. The design question here is which props you pass to the component - each of the four copies of the stat editor will need a different value for at least one of the props. The solution shown in the lectures is just one of many possibilities.

The Cloud

For this session, there are no videos. Instead, you should read the following web resources, the content of which is examinable:

- What is Cloud Computing?
- What is Cloud Economics?
- As A Service

There are no exercises for the cloud material. Instead, use this time to complete the React exercises, recap exercises from earlier in this part of the unit, and ask questions about this or other material from across the unit. If there are exercises that you could not solve or want to ask questions about, now would be a good time to revisit them and ask for help.

(The original intention was to include a non-examinable practical component here using AWS services, but there was trouble getting this set up for the class in time. While not the same as a guided exercise, some of you may want to look at AWS' own onboarding material here.)

A complete application, part 1

In this workshop we will set up and inspect a complete application including database, Java back-end/API and React front-end.

Just one short video to introduce the topic:

Video	Length	Slides
Application	18 minutes	slides

Please try the steps on the "setting things up" page before you come to the workshop - ideally you will then be able to start the walkthrough in the workshop itself immediately, but if you get stuck in the set-up, you will be able to ask for help at the very start.

Setting Things Up

The application lives in the `code/application` folder of the unit repository.

The Database

For the application to work, you will need a MariaDB database that serves the `census` database.

If you want to run the database on a vagrant VM but the Java application on your own machine (this is the recommended way), make sure the VM is serving the database correctly:

- MariaDB must be installed on the VM and the census database installed as described earlier in the unit.
- Port 3306 must be exported in the `Vagrantfile`.
- The `/etc/my.cnf.d/mariadb-server.cnf` must have the `skip-networking` line commented out or removed.

If you want to run the database and the Java application on your vagrant VM, make sure that the following are set up:

- MariaDB must be installed on the VM and the census database installed as described earlier in the unit.
- You must export port 8000 in the `Vagrantfile`. You do not need to export port 3306 to the host, as both the database and the application using it will be running on the VM.
- Either the `/etc/my.cnf.d/mariadb-server.cnf` must have the `skip-networking` line commented out or removed, or in `src/main/resources/application.properties` you must add `?localSocket=/var/run/mysqld/mysqld.sock` to the end of the data source URL to use a socket instead of a network (TCP) connection.

If you want to run everything on your own machine, then you need to install and run MariaDB and make sure the server is running on port 3306 (this proably means uncommenting `skip-networking` in a configuration file). By default, the Java application expects the database to have a username of `vagrant` and an empty password, but you can edit this in `src/main/resources/application.properties`.

The React Client

The client lives in `code/application/client` in the unit repository.

There are two ways to run the client. First (and recommended), install nodejs and npm if you have not done so already - this can be on a VM or on your local machine - open a terminal in the `client` folder and run the following:

- `npm install` to download and install the dependencies. These will end up in a folder `node_modules`, which is (sensibly) excluded from the repository via a `.gitignore` file in the same folder. You will get some warnings which you can ignore for now.
- `npm run-script build` will build the application and put the results in the `build` folder (which is also sensibly excluded from the repository).
- Copy the contents of the `build/` folder to the folder `../src/main/resources/static/` (the `../` is relative to the `client/` folder) for the Java application to serve.

If you are actively developing the react application, then instead of building it you can use `npm start` which will run a development server on `localhost:3000`. The advantage of this method is that whenever you change a source file in the client folder, it will recompile and reload the react client automatically.

The Java Application

In the folder `code/application` in the repository, there is a maven `pom.xml` file.

- Open this folder in a terminal and run `mvn spring-boot:run`.

This will automatically download any dependencies.

You can now point your browser at `localhost:8000` and you should see an overview of the regions of England. Clicking an item such as a region navigates to it, and the *Details* link loads the employment statistics for the current unit in a second panel on the right-hand side.

Developer Tools

For the following walkthrough of the application it is helpful to have everything running and to have the browser's F12 developer tools open in the tab running the application.

Since this is a React app, you may also want to install the React Developer Tools extension if you are using a Chrome-based browser (chrome, chromium, edge etc.). This provides extra React debugging and development integration, for example a new *Components* tab showing the structure of the React components on the page with their state and props.

Walkthrough

Here is a quick overview of what happens when the application loads and you click first *South West* then *Details*.

You should look at the mentioned code snippets as you follow along and make sure you understand what they do.

The first screen

1. When you open `localhost:8000` in your browser, a HTTP request for `/` goes to the Java server. If there were a controller configured for this URL pattern, it would now fire, but there isn't so spring falls back to serving `src/main/resources/static/index.html` if it exists as this is by convention the default file name to show. This file does exist, so it gets sent to the browser.
2. This file came from `client/build/index.html` which is the compiled version of `client/public/index.html`. There is not much to see here except a `<div id="root">` which is where the React application will be loaded (the necessary script tags get added when React compiles the file). If you watch the network tab when you open `localhost:8000` you will see requests for several css and js files, followed by a request of type `fetch` for `E92000001` which is the React app loading the country data to show.
3. When the React scripts on the page load, it runs the compiled version of the code in `client/src/App.js`. The `App.render()` method creates the navigation bar at the top (this is a standard Bootstrap component) and two custom elements called `OverView` and `UnitView` (in their own files), passing along some state. The initial state of the app is set in its constructor.
4. Once the component is loaded, the `OverView.componentDidMount()` function fires. This just delegates to `_fetchData()`. Here we fetch the country data: `App.state.overview` starts out at `country` and `App.state.overviewCode` starts out at `E92000001`. These are passed as props to the `OverView`, with names of `type` and `code`, which are referenced in the `fetch()` call in `componentDidMount`. The effect is to send off a request for `http://localhost:8000/api/country/E92000001`.
5. You can (and should) paste the above URL in a new browser tab, and you will see some JSON as a result. On the server side, the request is handled by the `CountryController.getCountryById` function (in `src/main/java/uk/ac/bristol/cs/application/controller`) as the path is declared in its `@GetMapping` annotation.
6. The `CountryController` first calls `getWithChildren` on the `repository.CountryRepository`. This is just an interface - the implementation is provided by Hibernate, automatically triggered by Spring - but the important part here is the `@Query` annotation that explains the extent of the data to fetch. This is HQL,

Hibernate's query language, and we tell it to fetch not only the country but also all its regions.

7. This returns an instance of `model.Country` which is more or less a simple Java class with properties `name`, `code` and `Regions`. Hibernate is in charge of creating the instances and the JPA annotations such as `@OneToMany` explain how it should process JOINs to fetch data from other classes (such as the regions).
8. Back in the controller, the next thing we do is render the country as JSON. We will discuss later why we are doing it this particular way, but the code is in `model.ModelClass.renderJSON` so it can be reused for different classes.
9. The JSON object now gets sent to the client. If you queried the URL yourself in a new tab, you just see the JSON at this point. If React queried it with `fetch()`, it can now check - back in `OverView._fetchData()` - whether the call was successful, and if so run `this.setState({loaded: "yes", item: result})` to store the object in its state (the previous call to `r.json()` decoded the JSON to a JS object, which is now in `result`).
10. The `setState` causes react to call `render()` which, now that `this.state.loaded` is `yes`, renders a the country information including a list of regions, pulled from `this.state.item` which is where the decoded JSON object is stored.

Navigating to a Region

Click on one of the region entries. The following happens:

1. The link (actually a button styled as a link, for accessibility reasons) has a click handler `onClick={() => this.navigate(i.code, false)}` (created in `OverView.children()`, where `i` is the item to navigate to, e.g. the region).
2. `OverView.navigate` calls `App.navigate` through `this.props` (the `App` passes a reference to its own `navigate` method when it creates the `OverView`).
3. `App.navigate` contains a state machine that decides, based on the current state (are we viewing a country, region, county or ward) and the `isParent` flag (are we navigating to a child or returning to the parent) what type of object we are navigating to. The only option when we are coming from the country view is a region, so this is set up correctly with `setState` and the code of the new item is set in the app state.
4. This state change causes `App.render` to run, which changes the props on the overview component.
5. The props change causes `OverView.componentDidUpdate` to be called with the old props for comparison (the new ones are already in `this.props`). If there has been a code change, then we (a.) immediately set the state back to "not loaded", which triggers a `render()` call and (b.) call `_fetchData()`, which asynchronously loads the new data and eventually calls `setState` again, triggering another `render()` this time with the new data.

Exercise: Return to the countries view using the *Back to parent* link, then shut down the Java server (so that futher API calls will fail). Then click on a region. You should immediately see a "Loading ..." message, followed around a second or so later by "Network error". Explain, with the help of the `OverView` code, how this happens.

Details View

Exercise: based on what you have learnt above, sketch the steps that happen when you click on *Details* to display the occupation data for the selected geographic unit. Look at both the client and the server code. The server-side methods start in `StatisticsController`.

React Lifecycles

React components have a lifecycle. The first important rule is that react can call `render()` any time it likes, and your implementation must always do something (other than crash) even if the component is not in a useful state. For example, if no data has been loaded yet then you can just display "loading ..." as we do in this example application, using the `state.loaded` flag to keep track of whether there is data to display or not.

React will call certain lifecycle methods on your component:

- `componentDidMount` gets called after the component is created and rendered for the first time. If you need to load data asynchronously (as we do here) then this is the method to do it in (in our case we delegate to `_fetchData` because there's another lifecylce method that wants to use this code too).
- `componentDidUpdate` gets called after a component's props or state have changed. This is another place where you can make network requests, for example.
- `componentWillUnmount` gets called before react removes a component. We do not need this in our application, but it can be used to e.g. save some data before the component "closes".

In `componentDidUpdate`, you should always check if this is a "real" update with an if-condition guard. Otherwise, you create an infinite loop: if `componentDidUpdate` calls `setState`, then the state change triggers another `componentDidUpdate` call. So this code:

```
componentDidUpdate(oldProps) {
    if (this.props.code !== oldProps.code) {
        this.setState({loaded: "no"});
        this._fetchData()
    }
}
```

only does anything if the props have just changed, namely because the parent component has just re-rendered us with a new code and/or type of geographical unit to display. In this application, that can only happen in a `navigate()` call.

This application also shows an example of *separation of concerns*: the `App` itself is in charge of holding the information on what item is currently being displayed, and knowing that e.g. the child elements of a country are called regions. The app class is also responsible for knowing how to navigate: the `OverView` can ask the app to navigate, but the app could in principle also refuse this. The app class does not however deal with loading, storing or displaying data for the individual countries, regions or other geographical units.

The `OverView` component's responsibility is to display the information for the object it is currently given via its props. It does not know how to navigate (beyond passing requests to the app) but it does know, given a code and a type, how to fetch, store and display that object.

The general pattern here is that the `OverView` updates its *state* based on its *props* - remember that a component's state is the "stuff it manages itself" whereas props are "stuff the parent is in charge of managing". This is a common pattern in react programming, but it has some complexities and the recommended solution has changed over time. In this application we follow the recommendation in the component documentation to deal with this in `componentDidUpdate`, but a 2018 react blog entry suggests using `getDerivedStateFromProps` instead.

The extent problem

On the server side, imagine a request comes in for a `Region` object.

It looks like spring, hibernate and jackson (the library that produces JSON) should be able to handle this automatically, hibernate converting a database row in to a Java object and jackson converting that to a string containing JSON, with Spring making sure the correct classes are loaded in the first place.

Unfortunately, the automatic approach assumes that there is only one way to deal with each domain class. If you can answer the question "if you load a region, do you want the counties or not?" once and for all in your application, then the automatic approach will work. However,

- When we are viewing a region, we do want to know the counties, as we want to show a clickable list of counties to the user.
- When we are viewing a country, we want to know the regions, but we do not care about the counties in the regions, and even less about the wards in the counties - otherwise a request for England would load the entire contents of the country, region, county and ward tables into memory all at once!

The solution in this application is to use explict *extents*, that is for each object we declare when we fetch it what properties we do or don't want to see.

The first step is to tell Hibernate not to do any joins unless told to, for example in the Region class with two `FetchType.LAZY` entries:

```
// Region.java

@ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name="parent")
@JsonView(Region.class)
private Country parent;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "parent")
@JsonView(Region.class)
private List<County> counties;
```

When we're fetching a region, the controller uses our own `findByIdFull` method:

```
// RegionController.java

@GetMapping(path="/api/region/{id}", produces="application/json")
String getRegionById(@PathVariable String id) {
    Region r = repository.findByIdFull(id);
    return ModelClass.renderJSON(r, Region.class, id);
}
```

which has an annotation declaring its extent:

```
// RegionRepository.java

@Query("SELECT r FROM Region r JOIN FETCH r.parent LEFT JOIN FETCH r.counties
WHERE r.id = ?1")
Region findByIdFull(String id);
```

Here, we use HQL (Hibernate Query Language) `JOIN FETCH` to declare that when we're loading a region with this method, we want both the parent (country) and all the children (counties), but not for example the children of the children (wards) as they're not in the query. The `?1` at the end is HQL's way of doing a bound parameter for a prepared statement.

The result is now that when we're trying to display a region, we get exactly the extent of information that we need to create the links, no more and no less. In contrast, when we're fetching a region through another query (e.g. to display a country) then the extent of that query will not load the region's children, as we don't need them.

Extents and JSON

The problem is even more pronounced when converting to JSON, as when we are displaying a region we do not want this to happen:

```
{   "code":"E12000009",
    "name":"South West",
    "counties": [
        {"code":"E06000022",
         "name":"Bath and North East Somerset",
         "parent":???}
... ]}
```

A region countains counties (which we want to see to create the list), and those counties have parents, and their parents are regions. So what do we put for the parent of this county? It can't be a copy of the region otherwise we have created infinite recursion. JSON doesn't have a nice way to make a "pointer" back to the copy of the region we already have (which is how the problem is avoided for Java objects in memory).

We can't say either that all objects should ignore their parents, as when we're displaying a county, we need the code of the parent to make the "Back to parent" link work. But when we're displaying a region, we don't care about the parent objects of the containing counties: not only don't we need to create links for them, but we already know the code - it's the code of the region we're currently displaying.

Jackson, the JSON generation library, by default displays all properties of an object - that is, it looks for all getter methods on a class and calls all of them in turn. There are several ways this can not end up with what we want.

First, if regions have counties as children and counties have regions as parents (a so-called *bidirectional association*) then the "call all getters recursively" strategy ends up in an infinite loop (in practice, it crashes with a stack overflow).

Secondly, if we call a getter on something that Hibernate hasn't fetched yet with a JOIN, then what the getter actually returns is a Hibernate proxy object, that in turn responds to calls to its getters by firing off a new SQL query to fetch the requested data.

Hibernate's proxy objects are a good solution to simple cases of the extent problem: when fetching a region, don't load the counties by default, but load them in a separate query if someone actually tries to access them later on. Unfortunately, especially combined with a "fetch all getters" JSON renderer, this gets you the N+1 problem as we already discussed earlier. Assuming we fetch just children, not parents to avoid the infinite loop, then even if Hibernate is clever enough to use a single query to get all the counties in a region, when the JSON renderer loops over each county to get the wards (which we don't want in the first place when displaying a region!) that's one query per county, so the N+1 problem in action.

My recommended solution here is the following line in `src/main/resources/application.properties`:

```
spring.jpa.open-in-view=false
```

This turns off the proxy objects' ability to load data later on: a call to load data that was not JOINed to start with now causes an exception. This is a good thing: you will immediately notice any extent problems (because they crash your application) and you can go and adjust your HQL queries to fix the problem. You can even write unit tests to make sure all your methods run correctly, whereas there is no easy way to write a unit test for "did this method only use 1-2 SQL queries to load its data".

To really solve the problem, we have to tell jackson which properties we want in our JSON, and again the answer is "it depends": when we're displaying a region, we do want the region's counties; when we're displaying a country, we want the regions but not the counties.

In the `ModelClass` base class for our domain classes, we customise exporting to JSON:

```java
// ModelClass.java

public static <T> String renderJSON(T element, Class<T> cls, String id) {
    if (element == null) {
        throw new NoSuchElementException(cls, id);
    }
    try {
        return new ObjectMapper()
            .writerWithView(cls)
            .writeValueAsString(element);
    } catch (JsonProcessingException e) {
        throw new RuntimeException(e);
    }
}
```

The key call here is `writerWithView(cls)` that creates a JSON writer for a particular *view*, allowing you to have more than one view per class. In this case we choose to reuse the domain classes as the views, namely, when we are viewing the contents of e.g. a region, then the view class is simply `Region`.

We can now declare attributes to have a `@JsonView`:

```java
// Region.java

@ManyToOne(fetch = FetchType.LAZY) @JoinColumn(name="parent")
@JsonView(Region.class)
private Country parent;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "parent")
@JsonView(Region.class)
private List<County> counties;
```

Jackson will only export an attribute if the view class is equal to, or a subclass of, the class requested in the code above so we now have the following situation:

- When we are viewing a region, the view class is `Region` so the region's parent and children (counties) will be included in the JSON.
- When we are viewing a country, the view class is `Country` so the parents and children of the country's regions will not be included in the JSON.

The mystery of `getParentCode()`

There is one more trick in the region class:

```
// Region.java

public Country getParent() { return parent; }
public String getParentCode() { return parent.getCode(); }
```

Jackson calls all the getters except ones it's been told not to, so when we're exporting a country to JSON, it won't call `getParent` on the contained regions because the `Region.parent` field is marked as `@JsonView(Region.class)`.

However, Jackson will call `getParentCode` on the regions, so the actual JSON returned for the country view (http://localhost:8000/api/country/E92000001) looks like this (with some pretty-printing):

```
{ "code":"E92000001", "name":"England",
  "regions": [
      {"code":"E12000001","name":"North East","parentCode":"E92000001"},
      {"code":"E12000002","name":"North West","parentCode":"E92000001"},
      {"code":"E12000003","name":"Yorkshire and The
Humber","parentCode":"E92000001"},
      {"code":"E12000004","name":"East Midlands","parentCode":"E92000001"},
      {"code":"E12000005","name":"West Midlands","parentCode":"E92000001"},
      {"code":"E12000006","name":"East of England","parentCode":"E92000001"},
      {"code":"E12000007","name":"London","parentCode":"E92000001"},
      {"code":"E12000008","name":"South East","parentCode":"E92000001"},
      {"code":"E12000009","name":"South West","parentCode":"E92000001"}
  ]
}
```

The regions don't show their parent, but they do show the `parentCode`. We don't need this for the application, this feature is more for demonstration purposes.

How does this work? Doesn't the following code require the parent to be loaded?

```
public String getParentCode() { return parent.getCode(); }
```

The answer is no, for two reasons. First, the reason that these regions were fetched here at all is that they were children of the current country. Hibernate knows that the `country.regions` and `region.parent` match up (we told it with JPA annotations, after all) so it could figure out the parent in this case anyway.

For the second reason, try and manually visit the page localhost:8000/api/regions which is not used by the react client. This is handled by `RegionController.getAllRegions`:

```
@GetMapping("/api/regions")
List<Region> getAllRegions() {
    return repository.findAll();
}
```

This in turn calls the `RegionRepository.findAll` method which is implemented inside JPA/Hibernate itself (you won't see it in the `RegionRepository` class itself, it's declared on the parent class `JpaRepository`). You can see the SQL query it ends up producing in the log output:

```
select region0_.code as code1_2_,
       region0_.name as name2_2_,
       region0_.parent as parent3_2_
from Region region0_
```

There is no JOIN with the parent (country) class going on here. In the JSON output, which does not use the `ModelClass` method of selecting an extent explicitly, it just returns a `List<Region>` which spring takes as a cue to pass through jackson to create JSON (shown here pretty-printed and abbreviated):

```
[ { "code":"E12000001",
    "name":"North East",
    "parent":null,
    "counties":null,
    "parentCode":"E92000001"
  },
  { "code":"E12000002",
    "name":"North West",
    "parent":null,
    "counties":null,
    "parentCode":"E92000001"
  },
  /* and the others ... */
]
```

The parent and counties attributes are `null` because we haven't explicitly excluded them, yet jackson also can't fetch them because they were not JOINed in the original query and we have turned off "open session in view" which would allow them to be fetched when jackson asks for them. Jackson and hibernate work well enough together not to cause a crash here: jackson notices that the objects are hibernate proxies and first asks behind the scene "are these attributes available?" before trying to call e.g. `parent.getName()`.

So why does the following succeed? Think about it for a few moments.

```
// Region.java

public String getParentCode() { return parent.getCode(); }
```

The answer is that the parent's code, but not the other attributes, is stored as a foreign key in the Region table, so the SQL query above actually does fetch the parent code. When hibernate creates a Region object and the parent / counties are not included in a JOIN, it will replace the parent and counties with hibernate proxy classes that roughly do the following:

- If "open session in view" (configuration line `spring.jpa.open-in-view` in `application.properties`) is turned on (this is the default) then the proxies run another SQL query to get the required data whenever someone asks for it, e.g. calls `getName()` on the parent proxy.
- If "open session in view" is turned off, and someone asks for data that is not loaded yet, throw an exception.
- Alternatively, libraries like jackson who notice the proxy objects can also ask "is this data available?" which is why the JSON returns "null" for the parent / counties in this case.

But since the parent code, unlike the rest of the parent data, was loaded in the original query, hibernate actually creates a proxy object here that stores this code and returns it if someone calls `getCode()` on the parent proxy. Only on the other methods like `getName()` where the data is not loaded yet, do you get an exception.

Conclusion

The learning point here is that when you have a one-to-many relationship and you're working with an object on the side with the foreign key, then it's no extra effort to keep the foreign key around. There are two reasons you might want to do this.

The first is to avoid infinite recursion (or finite but unnecessary bloat) when transferring a collection of objects via JSON or some other mechanism that does not have a concept of "pointers". For example, when sending a country and its regions via JSON, we can't have the regions contain a copy of the country in their parent field (as that would cause infinite recursion), and JSON itself does not support a "pointer" data type, but we can emulate this by having the regions contain the country's code.

If the data is being put back into proper objects on the client side (e.g. we had JavaScript classes for countries and regions) then the foreign keys could be used to set up "pointers" (technically: object references) correctly again.

Secondly, foreign keys can be used for navigation. A common pattern is to include the foreign key in a link (in our case "back to parent") as an option for the user. In other words, to provide a "navigate to parent" feature, you do not have to fetch/join the parent object and then read its primary key: you can just use the foreign key that you already have, saving a JOIN. Clicking a navigation link that was created using a foreign key ends up fetching the object (in this case a country) with the matching primary key.

Exercises

To see if you have understood the request lifecycle, here are a couple of small exercises where you can make changes to the application.

Client-side change

A change request comes in from the users: when viewing a unit (country, ward etc.), instead of just "Contains:", the heading before the list of children should read e.g. "Contains 5 wards:".

There are two parts to this change: first, you need to count the relevant children; secondly you need to display the correct noun e.g. if you're viewing a region, it should say "Contains ... counties" whereas when viewing a country, it should say "Contains ... regions".

The easiest way to make the change is as follows:

1. Find the file where the "Contains:" string is created (which component is this?).
2. From the point of view of this component, how do you access the new data you need?
3. Make the change, then run the react development server (`npm start` in the `client/` folder) then view the page on `localhost:3000` and check that the text is correct (the database and java server must be running, of course).
4. `npm run-scripts build` then copy the `build/` files to the `static/` folder again as you did in the set-up. The new page now should show under `localhost:8000` when you restart the java server.

Note that the name for the children is already stored in a variable for you. You do not need to remove the 's' if there is only one child, for this exercise.

You might ask what happens if you are viewing a ward, as wards do not have children. Convince yourself, the way the react component is written, that this case is handled safely already.

Server-side change

A bug report comes in from a code reviewer:

> The data returned from the API call `/api/ward/ID` does not contain the parent, even though this is in the spec. For example, `/api/ward/E05000132` gives

```
{"code":"E05000132", "name":"Fortune Green", "parent":null,
"parentCode":"E09000007"}
```

> but according to the spec it should be
>
> ```
> {"code":"E05000132", "name":"Fortune Green", "parent":{"code":"E09000007",
> "name":"Camden", "parentCode":"E12000007"}, "parentCode":"E09000007"}
> ```
>
> I also note that the server-side method for handling this call does not use the same pattern as the corresponding country, region and county methods. This may be related to the bug.

Your task is to rewrite the ward method along the same lines as the other ones to get the extent correct for this API call, and to make any other changes that this implies. You do not have to change the ward class itself, but you will need to make changes to two source code files. For each line that you change, write down separately for your own learning, or discuss with your group of students:

1. What is the effect of this line / change?
2. Why is it required?

You do not need to change any client code, as the client only consumes the JSON and it doesn't mind the extra fields. You will need to call the API method yourself to check that your change fixes the bug. But it's good practice to check that the whole application still works after your change!

(It would be even better practice to have unit tests for all of this, and you should write tests in your second-year software project, but for now we're teaching one thing at a time.)

A complete application, part 2

There are no new videos or readings for this workshop - you should instead review anything you need to prepare for the following exercise.

You will be able to continue working on the exercise in the following week, as there are no new lab exercises.

Exercise

Create an application similar to the one from the previous workshop, but for the elections database. The layout is up to you, but it should contain the following features:

- In one place, the application should show a list of wards in Bristol (from the elections database). There is no need for this list ever to reload later on, as it does not change, but you should create it in react and load it with a `fetch()`, instead of just creating the HTML on the server.
- Clicking a ward should, in another place, show the election results for this ward. The data for this should be fetched in the background with `fetch()`.
- Implement the list of wards and the election results as separate components, to demonstrate proper state handling in react.

Some hints to get you started:

- On start.spring.io you can create a ZIP with a template application and POM file for a spring application. You will need "MariaDB Driver", "Spring Web", and "Spring Data JPA" as dependencies.
- Create domain classes for wards, candidates and parties based on the schema.
- Write a simple controller that does something like listing all wards, or fetching a single ward by id - without including any candidates or parties yet (leave them off the ward class for now). Test this in your browser and check you get the JSON that you expect.
- Create a sample react application in a separate folder, and write a component that fetches and displays a list of wards (just a list of names is fine to start with, they do not need to be links yet).

From this "MVP" onwards you can decide which data you need to pass to the client for which features, write controllers based on that, and then implement components that consume and display the data. It is probably easiest to work with two servers running while you are developing, the Java one on `localhost:8000` and the React one on `localhost:3000`.

The data set is small enough that you do not have to worry about extents too much - you can just return domain objects directly from your controllers and Jackson's automatic JSON conversion should do what you need, as long as you do not produce an infinite loop. The way to avoid an infinite loop is to make the associations in only one direction, for example a ward can have a list of candidates but a candidate object does not need a pointer back to the ward as you'll never be working with a candidate in a place where it's not clear what ward they stood in.

For some computations, like if you want to display the total number of votes cast in a ward, it is your choice whether to do them on the client or on the server. If you do this on the server, you can for example add a `getTotalVotes()` method to the ward class, then the JSON export will contain a `totalVotes` property.