Lab 3

CSS Media Queries

First, let's start with a little bit of CSS to get it out of the way and move onto the exciting stuff.
One relatively new feature that helps make responsive (mobile-friendly) websites is called Media Queries (https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries). It allows to modify the CSS of the website depending on screen characteristics (resolution, width, …), device type (screen, print), or even user preferences (dark mode, reduced motion, …). This is all done using the `@media` special (https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule) rule.

A media query has 3 components:

- **Media type** (https://developer.mozilla.org/en-US/docs/Web/CSS/@media#media_types): the general category of the device. Can be `screen`, `print` (for paged material and documents, such as when printing the page), `speech` (for screen readers), or `all` (the default)
- **Media feature** (https://developer.mozilla.org/en-US/docs/Web/CSS/@media#media_features): the characterestic of the user agent or environment, such as `prefers-reduced-motion` (https://developer.mozilla.org/en-US/docs/Web/CSS/@media/prefers-reduced-motion) (indicates that the user doesn't like animations), `width` (https://developer.mozilla.org/en-US/docs/Web/CSS/@media/width) (used to test for the width of the viewport), or `hover` (https://developer.mozilla.org/en-US/docs/Web/CSS/@media/hover) (used to test whether the user's main input method supports hovering; think of a mouse compared to touch screens).
- **Logical operators** (https://developer.mozilla.org/en-US/docs/Web/CSS/@media#logical_operators): Used to combine the different features using `and`, `not`, and `only`.

Examples:

```
@media print {
  body { font-size: 10pt; }
}

@media (min-width: 35rem) {
  div {
    background: yellow;
  }
}

@media (max-width: 50rem) {
  div {
    border: 2px solid blue;
  }
}
```

We usually use media queries to implement so-called breakpoints:

```
/* Extra small devices (phones, 600px and down) */
@media only screen and (max-width: 600px) {...}
/* Small devices (portrait tablets and large phones, 600px and up) */
@media only screen and (min-width: 600px) {...}
/* Medium devices (landscape tablets, 768px and up) */
@media only screen and (min-width: 768px) {...}
/* Large devices (laptops/desktops, 992px and up) */
@media only screen and (min-width: 992px) {...}
/* Extra large devices (large laptops and desktops, 1200px and up) */
@media only screen and (min-width: 1200px) {...}
```

Node and NPM

Node.js® (https://nodejs.org/en/) is a JavaScript runtime that runs outside the browser, allowing for writing server-side and CLI apps. It comes with a package manager called **npm** (https://www.npmjs.com/) that helps with managing the project's dependencies.
To create a Node project, go to a new directory and run `npm init`, which will prompt you for some pieces of information and then create a new file called `package.json`. The existence of this file is what identifies a Node package.

We can then start installing dependencies using `npm install <pkg>`. For example, let's install the popular lodash (https://lodash.com/) library: `npm install lodash-es` (The ES Modules version).
Now, how do we make use of it? Let's try to import it directly:

```
<script type="module">
  import clamp from './node_modules/lodash-es/clamp.js';
  console.log(clamp(1, 10, 12));
</script>
```

However, the `node_modules` folder should not be shipped to the user since it is huge and contains tons of unnecessary code. This is why we use bundlers.

Bundlers

Bundlers are the equivalent of compilers in the JavaScript world. They combine multiple JavaScript files into a single file that is production-ready. “Production-ready” means that it is minified (shortens the code to save on the precious bytes), tree-shaken (eliminates dead code), and sometimes poly-filled (made compatible with older standards).

One such bundler that we will use today is Rollup (https://rollupjs.org/guide/en/). After initializing our Node project, we add Rollup to the development dependencies using `npm install --save-dev rollup`. Then, to use it, we need to add the following to `package.json`:

```
"scripts": {
    "build": "rollup --config rollup.config.js"
}
```

Now we can run it using `npm run build`. This is because scripts defined in `package.json` can resolve binary (executable) files in the `node_modules` dependencies, and rollup defines one such file. Note that the command `rollup` is otherwise not available from your command line directly (unless you installed it wit the `-g` flag).

There are a few changes needed now. First, we should add the configuration file: **`rollup.config.js`**:

```
export default {
    input: 'src/main.js',
    output: { file: 'bundle.js' },
};
```

Since we are using ES Module syntax here, we also need to add `"type": "module"` to `package.json` to tell Node.js not to use the default CommonJS syntax.

`index.html`:

```
<body>
    <script src="./bundle.js"></script>
</body>
```

src/module.js:

```
export const PI = 3.14;
export const inc = (n) => n + 1;
```

src/main.js:

```
import { PI, inc } from './module';

console.log(PI);
```

Now, after running `npm run build`, we can see that it outputs the following file:

bundle.js

```
const PI = 3.14;

console.log(PI);
```

We can see that even though we defined, exported, and imported the `inc` function, it doesn't make its way to the final bundle since we never called it or used it in any way. That's tree-shaking.

Now let's try the same import as before from `lodash` and see it copy over the function, removing the need for shipping `node_modules` to production:

src/main.js

```
import clamp from 'lodash-es/clamp';

console.log(clamp(1, 10, 12));
```

Notice how the bundler removes the need to specify relative path to `node_modules` and to explicitly write the `.js` extension?

The output now looks like this:

```
import clamp from 'lodash-es/clamp';

console.log(clamp(1, 10, 12));
```

hmm, something doesn't seem right…

Plugins

The problem is that, by default, the Rollup bundler can only find (resolve) files inside our project; it does not know anything about the `node_modules`, which is why it left the `import` statement intact instead of replacing it with the content. To import Node modules, we need a **plugin**.

Node modules resolver

Plugins, simply speaking, allow to customize Rollup's behavior. They do that by defining some functions (hooks) that operate at important points during the bundling process. For example, to add the Node resolution algorithm (https://nodejs.org/api/modules.html#modules_all_together) (which allows to import from `node_modules`) to Rollup, we use the @rollup/plugin-node-resolve (https://www.npmjs.com/package/@rollup/plugin-node-resolve) plugin. We install it as a dev dependency and use it as such:

```
import { nodeResolve } from '@rollup/plugin-node-resolve';

export default {
    input: 'src/main.js',
    output: { file: 'bundle.js' },
    plugins: [
        nodeResolve(),
    ],
};
```

Now we can see the generated bundle containing the relevant code (and nothing else from the `lodash` library) without any imports. It is ready to be included directly in the HTML.

Terser

Another plugin that is used to minify your code is called `@rollup/plugin-terser` (https://www.npmjs.com/package/@rollup/plugin-terser). We use it in pretty much the same way: `npm install @rollup/plugin-terser --save-dev` followed by

```
import { nodeResolve } from '@rollup/plugin-node-resolve';
import terser from '@rollup/plugin-terser';

export default {
    input: 'src/main.js',
    output: {
        file: 'bundle.js',
    },
    plugins: [
        nodeResolve({ browser: true }),
        terser(),
    ],
};
```

It mangles the variable names so they are shorter and generally compresses your code. For example, for the code above using `inc` (before the lodash example), it outputs just the following:

```
console.log(1+1);
```

We can observe that it even inlined the function call (which is fine since it's a pure function).

Task

Today, we're going to create a simple chat room application to practice using bundlers. The code was mostly taken from socket.io's chat-example (https://github.com/socketio/chat-example/blob/master/index.html)

Server

Since this is a frontend course, we will not bother much with backend. Just copy the following code:

`server/package.json`:

```
{
  "name": "server",
  "private": true,
  "main": "main.js",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.18.1",
    "socket.io": "^4.5.1"
  }
}
```

`server/main.js` :

```
const app = require('express')();
const cors = require('cors');
const http = require('http').Server(app);
const io = require('socket.io')(http, {
    cors: { origin: '*' }
});
const port = process.env.PORT || 3000;

app.use(cors());

io.on('connection', (socket) => {
  socket.on('chat message', msg => {
    io.emit('chat message', msg);
  });
});

http.listen(port, () => {
  console.log(`Socket.IO server running at http://localhost:${port}/`);
});
```

Don't forget to run `npm install` first inside the `server` directory, and then `node main.js`. If you see a message like `Socket.IO server running at http://localhost:3000/` in the console, everything is good so far 👍.

Client

Let's start by installing the required client library: `npm install socket.io-client` (note that it is a normal dependency this time, not a dev one).

Now, we want an interface that displays other people's messages, as well as an input to send a message of our own:

```
<ul id="messages"></ul>
<form id="form">
  <input id="input" autocomplete="off" />
  <button type="submit">Send</button>
</form>
```

Note that having the `<input>` and `<button>` both inside a `<form>` element causes the browser to emit the `submit` event whenever you either click on the button or press `Enter` inside the input.
I'll leave the CSS as a practice for you :)

Note that having the input inside a `form` element allows us to use either the send button (whose `type` attribute now defaults to `submit`) or the `Enter` key to send a message.

First, let's receive messages sent to us by the server:

```
import { io } from 'socket.io-client';

const socket = io('http://localhost:3000');

const messages = document.getElementById('messages');

socket.on('chat message', function (msg) {
  var item = document.createElement('li');
  item.textContent = msg;
  messages.appendChild(item);
  window.scrollTo(0, document.body.scrollHeight);
});
```

Great! Now all we need to do is allow sending messages to the server via the WebSocket:

```
const form = document.getElementById('form');
const input = document.getElementById('input');

form.addEventListener('submit', function (e) {
  e.preventDefault();
  if (input.value) {
    socket.emit('chat message', input.value);
    input.value = '';
  }
});
```

Amazing! Now we can chat with other people on the internet (assuming our server is not hosted at `localhost`)!
Speaking of the server URL, it would be great if we can have it come from outside instead of it being hard-coded in our code, so that we can reuse the code in different environments. Well, that's the job for yet another plugin!

Install the `dotenv` (https://www.npmjs.com/package/dotenv) package and `@rollup/plugin-replace` (https://www.npmjs.com/package/@rollup/plugin-replace) plugin: `npm i --save-dev dotenv @rollup/plugin-replace`. `dotenv` allows us to populate environment variables (the `process.env` object in Node) from a `.env` file rather than having to add them to our system's environment variables, and `@rollup/plugin-replace` acts as a macro (`#define` in C/C++) that replaces parts of our with whatever else we define.

Now, we just need to add a `.env` file with the following content:

```
SERVER_URL=http://localhost:3000
```

It should actually point to where your server is deployed. It is also worth mentioning that `.env` is usually kept under `.gitignore` since it can contain secrets or configurations specific to your environment.

Then we add this towards the top of `rollup.config.js`:

```
import replace from '@rollup/plugin-replace';

import dotenv from 'dotenv';
dotenv.config();
```

and add it to the plugins as such:

```
plugins: [
    nodeResolve({ browser: true }),
    replace({
        values: {
            'process.env.SERVER_URL': JSON.stringify(process.env.SERVER_URL),
        },
        preventAssignment: true,
    }),
    terser(),
],
```

Lastly, we replace the hard-coded string in the main script file as such:

```
const socket = io(process.env.SERVER_URL);
```

And that’s it. Now, we can use the same code but point to different servers when using different environments. For example, one GitHub Action may build the code to point to a staging server (when testing), while a different one may instruct it to use the production one (when releasing to the users).

Homework

Nothing for this week!