Lab 4

This is the final laboratory in the web programming course, *objectives*:

1. Get experience using a REST api for fetching data.
2. Get experience with chaining `Promises`
3. Get experience using persistent storage in the web browser.

Background

The assignments bellow assumes you have a working solution for lab 3, i.e. a working react app with three components: `App`, `ComposeSalad`, and `ViewOrder`.

Assignments

1. We are going to remove the inventory from our compiled code and instead fetch the data from a REST server. The server is already implemented and you need to download it and run it on your own computer. It is based on the `express` package and some other packages, so you need to download them as well. Lets use `npm` for this:

   ```
   > mkdir server
   > cd server
   > curl -o server.js http://fileadmin.cs.lth.se/cs/Education/EDAF90/labs/lab4/server.js
   > curl -o package.json
            http://fileadmin.cs.lth.se/cs/Education/EDAF90/labs/lab4/package.json
   > npm install
   > npm start
   ```

   The server should now be running and waiting for requests. Test it using `curl` in the terminal, or write the urls in a browser:

   ```
   > curl -is http://localhost:8080/foundations/
   > curl -is http://localhost:8080/proteins/
   > curl -is http://localhost:8080/extras/
   > curl -is http://localhost:8080/dressings/
   > curl -is http://localhost:8080/dressings/Dillmayo
   ```

   Option: You can also access the salad bar REST service on codesandbox: `https://wkqy6rpw25.sse.codesandbox.io/foundations` IMPORTANT: you must load the page in a web browser before you start your application. Codesandbox first send a html page containing the compilation process before it jumps to the `server.js` response. This will most likely give problems for your application.

2. We need to store the fetched data somewhere in your application. Since it will affect the rendering of your application it must be stores in a component state (remember, `render()` must be a pure function of `this.state` and `this.props`). In witch component should you store the fetched data? The data will be used in several components and of course should only be fetched once. To share data between components, we must pass the data from a parent to its children (`props`). Therefore shared data must be stored in a common ancestor of all components using the data. In our case this is

App. Please read more about dynamically fetching data in react applications here https://www.robinwieruch.de/react-fetching-data/.

To minimise the changes to the application we will recreate the inventory object from inventory.ES6.js. You have already written the code for passing the data from App to the children, so now we only need to recreate the inventory object and add it to this.state. First, open App.js and remove the import line:

```
// import inventory from './inventory.ES6';
```

This leads to some compile errors since inventory is removed from the global name space. A good practice is to always have a valid data structure for dynamically fetched data in your app. Initially the inventory is empty, in the constructor of App:

```
this.state = {orders: [], inventory: {}};
```

Update the rest of your code so it works with this change, e.g. replace inventory with this.state.inventory. *Note:* make sure your components can handle an empty inventory object, for example in ViewOrder or in Salad, you probably have something like:

```
let selected = 'tomato';
const price = this.state[selected].price;
```

It will throw an TypeError when inventory is an empty object.

3. Next we will recreate the inventory object by fetch data from the salad bar REST server. When should the app fetch the inventory data? You could fetch the data when needed, but in this case i suggests you fetch all data when the app launches. Check out the react lifecycle hooks to get some suggestions on where to place the fetch code, see https://reactjs.org/docs/state-and-lifecycle.html.

   Use the fetch(url, [options]) function to send a request. It might be easiest to build the url using string concatenation, but you can also check out the URL(url, [base]) class. Browse the documentation for fetch(). It returns a Promise that resolves to a Response object. To get the body you can use Response.json(), which returns yet another Promise. When you have the ingredient name, the next step is to fetch its properties from the server, e.g. fetch(‘http://localhost:8080/extras/Tomato’). Fetching the properties for all ingredients are independent actions and should be carried out in parallel.

   Every time you call setState() the DOM is potentially updated and the view is updated. This is a costly operation and we do not want to do this for each inventory (it will make the UI slow). We want to wait until all, or a sufficient subset, of the fetch operations finish. This is a perfect use case for Promise.all(). It takes an array of Promise objects, which will run in parallel, and returns a Promise that settles when all inner promises are settled. Use this to avoid to frequent updates of the state.

   Note: For security reasons, JavaScript code will only allow to send http requests to the server it was downloaded from, its origin. The reason is to protect the user from cross site scripting attacks, which will be covered in the last lecture. The origin is both the IP-address and the port. The salad bar REST server is running on a different port than the react development server, so the servers have different origins and, by default, the browser prevents your app from communicating with the salad bar REST server. Luckily there is a way to relax this constraint. A server can allow communication with scripts from other origins using the Access-Control-Allow-Origin header. If you look at the headers returned by the salad bar REST server, see the curl commands above, you see that the server allows access from *, meaning any server. The browser still do not trust this communication, and

hides most http headers. Do not bother looking for the header information in your app. In the lab you may assume that the body contains json data, however do check the status code to make sure your request was successful.

4. There is one more REST functionality missing in your app, placing an order. Implement this using a `POST` request containing the salad order. The REST api for this can be tested using:

```
curl -isX POST -H "Content-Type: application/json"
     --data '["salad1", "salad2"]' http://localhost:8080/orders/
```

5. I have one more assignment for you. Store the order in local store, and load it when the app starts. This is done using the `window.localStorage` There are two functions, `setItem(key, value)` and `getItem(key)`. All values are text, so use `JSON.stringify()`, and `JSON.parse()`. Note, `parse` gives you a JavaScript object with the right structure, but it is not an instance of the `Salad` class. The `price()` method et.c. are missing. This can be solved in two ways:
   - Use `Object.setPrototypeOf()` so set up the right prototype chain for the objects returned from `JSON.parse()`
   - Make a static function that takes the salad as a parameter, `Salad.price(mySalad)`.

Editor: Per Andersson

Contributors in alphabetical order:
Alfred Åkesson
Oscar Ammkjaer
Per Andersson

Home: `https://cs.lth.se/edaf90`

Repo: `https://github.com/lunduniversity/webprog`

This compendium is on-going work.
Contributions are welcome!
Contact: `per.andersson@cs.lth.se`