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 but to use it, you need to download it and run it on your own computer. It is based on the `express` package and some other packages. Let `npm` download the packages for you.

   1. download lab4_server.zip from canvas.
   2. unpack the zip
   3. install the npm packages and start the server:

   ```
   > cd lab4_server
   > 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 -i http://localhost:8080/foundations/
   > curl -i http://localhost:8080/proteins/
   > curl -i http://localhost:8080/extras/
   > curl -i http://localhost:8080/dressings/
   > curl -i http://localhost:8080/dressings/Dillmayo
   ```

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`. The GUI will not be updated if you change `inventory` in lab 2 and 3). In wich 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 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. In the constructor of `App`, add an empty inventory to the state:

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

Update the rest of the code in `App` 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 `ComposeSalad`, you might have something like:

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

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

3. Next we will update `this.state.inventory` 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 lifecycle of a React component 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/Tomat')`. Fetching the properties for all ingredients are independent actions and must be carried out in parallel. Important! `fetch()` throws an error in some situations, but not all. If the server response code is 404, `fetch()` will ignore this. For proper error handling you need handle this:

```
fetch(url)
.then(response => {
  if(!response.ok) {
    throw new Error(`${url} returned status ${response.status}`);
  }
  return response.json();
}).then (...)
.catch(error =>
  console.error(error);
});
```

   Every time you call `setState()` the render functions are potentially called and the DOM is updated. This is a costly operation and we do not want to do this for each inventory (it can make the UI slow). We want to wait until all, or a sufficient subset for example all foundations, 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.

Having nested calls to `fetch()` becomes complicated if you try to write a singel chain of `.then()`. Divide this task into small steps:

1. Write a function for fetching a single ingredient. It should return a `Promise`.
2. Write the code to fetch the list of foundations.
3. Foreach ingredient, use `fetchIngredient()` to get the ingredient properties from the server.
4. Use `Promise.all` to wait for all promises from 1 to settle.
5. If needed, merge the data to a singel object
6. Use `setState` with this object to update the component state.

Hint: the task might be simpler if you leverage on closure. In the example bellow, `inventory`, is visible in all `.then` blocks:

```
const inventory = {};
fetch(...)
.then(x => {...})
.then(y => {...})
.then(z => {...})
.catch(error => {
  console.error(error);
});
```

Or, you wrap all data needed down the chain in an object:

```
fetch(...)
.then(x => ({x, step1: foo(x)}))
.then(y => (...)) // x is not in the scope, but y contains both x and foo(x)
.catch(error => {
  console.error(error);
});
```

Note: For security reasons, JavaScript code is 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 output in the terminal from the `curl` commands in assignment 1. In the headers 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 functionality involving a REST call missing in your app, placing an order. Add an order button in the `ViewOrder` component. To place an order, you need to send a `POST` request containing the details. The REST api for this can be tested using:

```
curl -isX POST -H "Content-Type: application/json"
   --data '[["Sallad", "Norsk fjordlax", "Tomat", "Gurka", "Dillmayo"]]'
   http://localhost:8080/orders/
```

The body of the request contains an array of salads. Each salad is an array with the selection for the salad (array of strings). *Hint:* `Object.keys(mySalad.ingredients)`. The response is an order confirmation:

```
{"status":"confirmed",
"timestamp":"2022-02-06T13:36:50.506Z",
"uuid":"478a217b-19d4-4958-ad27-11a694ea8574",
"price":55,
"order":[["Sallad","Norsk fjordlax","Tomat","Gurka","Dillmayo"]]}
```

View the order confirmation to the user, for example using a toast: `https://getbootstrap.com/docs/5.1/components/toasts/`.
Optional assignment: Store the order confirmations in `App` and view them in a new component.

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 stored in localestore as 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 `getPrice()` method et.c. are missing. There ar several solutions to this:

   - `const salad = new Salad(text)` if your salad class support it.
   - Use `Object.setPrototypeOf()` so set up the right prototype chain for the objects returned from `JSON.parse()`
   - Do not use `Salad` objects for storing the data. Instead create static functions that takes `Object` instances as a parameter. The functions assumes that the argument have the same structure as a `Salad` class instance. For example: `Salad.getPrice(mySalad)` instead of `mySalad.getPrice()`.
   - Set `this` when calling `getPrice()`: `Salad.prototype.getPrice.call(mySalad)`, instead of `mySalad.getPrice()`.

   Reflection question: Which alternative do you prefer? Due to the problem with `JSON.parse` mentioned above, the convenience of object literals, and the need to copy objects for `setState`, the third option is common i React apps.

   After reading salads stored in localstore and creating new ones you should note that the uid of the salads are not unique. It would be better if `Salad` used Universally Unique IDentifier (UUID). RFC4122 details how these can be created. You do not need to implement this since it has already been done and is available at npm, `https://www.npmjs.com/package/uuid`. In the terminal (`ctrl-c` if the development server is running):

   ```
   npm i uuid
   ```

   And in `Salad.js`:

   ```
   import { v4 as uuidv4 } from 'uuid';
   constructor(){
     ...
     this.uuid = uuidv4();
   }
   ```

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`