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
   ```

   In powershell, use `Invoke-WebRequest -Uri http://localhost:8080/proteins/`.

2. We will use react router data API for loading inventory. Each route can have a loader function which will be executed befor the associated component is rendered. Here is my rout for compose-salad:

   ```
   {
     path: "compose-salad",
     loader: inventoryLoader,
     Component: ComposeSalad
   }
   ```

   You also need to implement the `inventoryLaoder` function. Start with this:

```
async function inventoryLoader() {
  const inventory = { Sallad: { price: 10, foundation: true, vegan: true } };
  await new Promise(resolve => setTimeout(resolve, 500));
  return inventory;
}
```

There is a delay of 500 ms befor the component is rendered. Feel free to vary this during the lab. Most of the time you want it to be 0, but increase it when working on the spinner. You can get the data returned by the loader inside a component using a hook:

```
import { useLoaderData } from 'react-router-dom';
function ComposeSalad({ onOrder: handleOrder }) {
  const inventory = useLoaderData();
  \\ more code
}
```

Load your app in a browser and test so the router and loader function works. Of curse there is only one ingredient to choose now.

3. Next we will fetch data from the REAS-server. Use the `fetch(url, [options])` function to send a http request to the inventory server. 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 of `fetch()`. It returns a `Promise` that resolves to a `Response` object. To get the http body you can use `Response.json()`, which returns yet another `Promise`. First fetch the list of foundations, then for each foundation fetch its properties from the server, e.g. `fetch('http://localhost:8080/extras/Tomat')`. Fetching the properties of the ingredients are independent actions and to reduce loading time, you must fetch them concurrently. *Hint:* `fetch()` returns directly, so it is easy to start several concurrent requests. Use `Promise.all()` when you need to wait for several promises. Important! The `fetch()` promise will only reject with a TypeError if a network error occurs. If the server response with an http code, for example 404, `fetch()` will treat this as a valid response. To catch http errors you need to check the response.ok flag:

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

Mixing sequential and parallell actions in a singe `Promise.all()` chain can be hard to write and confusing to read. Use functions to organise and structure your code. Place sequence of actions you want to run in parallel in a function. Then the parallel actions becomes a singe link in the outer promise chain, making the code easier write and read.

1. Write the code to fetch the list of foundations. This will give you an array of ingredient names.
2. Write an `async` function for fetching a single ingredient. Example: `fetchIngredient('extras', 'Bacon')` should return `{price: 10, extra: true}`, `{Bacon: {price: 10, extra: true}}`, or similar.
3. For each ingredient in the list returned in step 1, use `fetchIngredient()` to get the ingredient properties from the server. Now you have both the name and property of

each ingredient. Add them to the inventory.

4. Wait until all foundations have been added to the inventory. Use `Promise.all`.
5. You need to repeat the step above for proteins, extras, and dressing. This is easier if step 3 and 4 is in a function.
6. `inventoryLoader()` should return a promise containing the inventory that will settle when it is complete.

Hint: Look at the slides from lecture 6, "chaining", for examples on how to pass data down the promise chain.

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 JavaScript code from 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. Fetching the inventory might take some time and the user will get annoyed if there is no direct response when navigating to the `view-order` page. To solve this we will display a spinner while the inventory is loading. The `ComposeSalad` is not rendered until after the inventory is fetched, so we can not place the spinner there. Instead it must be in a component that is already rendered when the navigation starts, either in `App` or a new `Spinner` component. Change the render code in `App/Spinner` to either show a spinner, or its child in the router tree (`<Outlet />`). This will display a spinner when any child loads data. Use the navigation hook to detect when data is loading. It will return an object containing information of any ongoing navigation. Its `state` property will be one of `idle`, `loading`, or `submitting`. You can also look at the `location` property. It will be `undefined` when there is no ongoing navigation, see `https://reactrouter.com/en/main/hooks/use-navigation`. Feel free to use a bootstrap spinner `https://getbootstrap.com/docs/5.3/components/spinners/`.

```
function BootstrapSpinner() {
  return <div className="d-flex justify-content-center">
    <div className="spinner-border" role="status">
      <span className="visually-hidden">Loading...</span>
    </div>
  </div>
}
```

Reflection question: What is the difference between `<BootstrapSpinner />` and `{BootstrapSpinner()}` when using the bootstrap spinner component in a JSX expression?

5. 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 with the ordered salads. Each salad is an array with the selected ingredients (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 an order confirmation to the user, for example using a toaster `https://getbootstrap.com/docs/5.3/components/toasts/`. You need to strore the order confirmation in the component state.
Optional assignment: Store the order confirmations in `App` and view them in a new component.

6. I have one more assignment for you. Store the shopping basket in the browsers local storage, 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 `Salad.parse()` from lab 1.

   After reading salads stored in localstorege, new salads created in the `ComposeSalad` component might not get a unique `id` when using a static instance counter. This can happen since the counter is reset to 0 when the app is reloaded. The solution is to use the `uuid` property.

Editor: Per Andersson

Contributors in alphabetical order:
Ahmad Ghaemi
Alfred Åkesson
Anton Risberg Alaküla
Mattias Nordal
Oskar Damkjaer
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`