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 `props` and the component state (`useState` hook). The GUI will not be updated if you change `inventory` in lab 2 and 3). In which 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 `App()`, add an empty inventory to the state:

```
const [inventory, setInventory] = useState({});
```

Now the app should work again, at least if your components can handle an empty inventory object. For example in `ViewOrder` or in `ComposeSalad`, you might have something like:

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

It will throw an `TypeError` when inventory is an empty object. The above code is not a problem now since 'Tomat' can not be selected in the first place, but later in the lab you will load a salad from local storage befor inventory is populated.

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://beta.reactjs.org/learn/synchronizing-with-effects`. The effect hook is only available in function components. Class based components should instead override `componentDidMount()`, see `https://reactjs.org/docs/react-component.html`.

   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 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`. First fetch the list of foundations, the 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. Important! `fetch()` throws an error in some situations, but not all. If the server response code is 404, `fetch()` will treat this as a valid response. To catch tis 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();
  });
}
```

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). Also, we do not want to show the user a subset of the options, either none or all foundations options are shown. It is however ok to show the foundation options while still waiting for the protein options. 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. You should use four calls to `Promise.all()`, one for each for foundations, proteins, extras, and dressings.

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 chain in the outer promise chain, making the code easier write and read.

1. Write an `async` function for fetching a single ingredient. Example:
   `fetchIngredient('extras', 'Bacon')` should return {`price: 10, extra: true`},
   or
   {`Bacon: {price: 10, extra: true}`}.
2. Write the code to fetch the list of foundations. This will give you an array of ingredient names.
3. Foreach ingredient, use `fetchIngredient()` to get the ingredient properties from the server. *Hint:* `Array.prototype.map()`.
4. Wait for all ingredient properties to be fetched. Use `Promise.all`.
5. Update the component state: `setInventory (oldInventory => merge(oldInventory, fetchedIngredients))`.

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. 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 an order confirmation to the user.
Note: if you want to use a toast, you need to initialise it: `https://stackoverflow.com/questions/67946016/how-to-show-bootstrap-5-toast-in-react-js`.
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 that `new Salad(text)` only handles a singel salad object, but localstore contains the order, an array of salad objects. Place the code for parsing the array of salads in the Salad class instead of the React component.

   After reading salads stored in localstore, new salads created in the `ComposeSalad` component might not get a unique `id`. This can happen since the static instance 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:
Alfred Åkesson
Anton Risberg Alaküla
Mattias Nordal
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`