An example of asynchronous loading

Relevant links

- Introduction to promises[1]
- MDN documentation on promises[2]
- async and await[3]

Notes

In this section we will a small example to illustrate the different asynchronous loading techniques.

We will use a small Node.js application that runs a tiny local server and keeps a running list of tasks. Here is the code for it, stored in a file called app.js:

```
// To run, need:
// npm install express express-handlebars
// node app.js

const express = require('express')
const exphbs  = require('express-handlebars');

const app = express()
const port = 3000

const tasks = [];

// Templates
app.engine('handlebars', exphbs({ defaultLayout: false }));
app.set('view engine', 'handlebars');

app.get('/json', (req, res) => res.json({ tasks: tasks }));

app.get('/', (req, res) => res.render('tasks', { tasks: tasks }));
app.post('/', (req, res) => {
    tasks.push(req.body.task);
    res.redirect('/');
});

app.listen(port, () => console.log(`Example app listening on port ${port}!`))
```

This uses a view file called views/tasks.handlebars to render the list of tasks. Here is the code for that:

```
<!DOCTYPE html>
<html>
    <head>
```

[1] http://exploringjs.com/es6/ch_promises.html
[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

```
        <meta charset="utf-8">
        <title>Example App</title>
    </head>
    <body>
        <div>
            <ol id="tasks">
                {{#each tasks}}
                <li>{{this}}</li>
                {{/each}}
            </ol>
        </div>
        <form action="/" method="post">
            <input type="input" name="task" placeholder="add a task here!" />
            <input type="submit" value="Add it!" />
        </form>
        <script type="module">
        // Add javascript code here
        </script>

</body>
</html>
```

We can start the application by running `node app.js` from the terminal. Then we can open up our browsers to `localhost:3000` and interact with the application (in fact, multiple people can interact at the same time, all adding to the same list of tasks).

Currently this works non-dynamically:

1. When you hit the submit button, a POST request is sent to the server.
2. The server processes this request, and serves back a **new** webpage, which just so happens to be the one we started at.
3. One of the consequences is that when someone else adds a new task to the list, we will not see it until we either refresh the current page or submit a new task of our own.

Basic XHR solution

What we want instead is a system whereby the page periodically checks for changes in the background. To that end, we have the `/json` endpoint (try `localhost:3000/json`), which provides the list of tasks in a more compact and easy to process way. Let's use the `XMLHttpRequest` object to process this:

```
function requestUpdates() {
  let xhr = new XMLHttpRequest();
  xhr.onload = function(ev) {
      console.log("Event", ev);
      console.log("xhr object has info", xhr);
  };
  xhr.open("get", "/json", true);
  xhr.send();
}

setInterval(requestUpdates, 1000);
```

So we create a function that sets up the whole xhr object, an onload handler etc. Then we set that function to be called on a 1 second interval. So every second this function will ask for updated data from the server, then it will console.log some information for us.

Of course instead of it logging some information, we want to change what the webpage shows us. In general, what we do in the onload function can be pretty complex, we'll want to find a way to separate what we do from the loading process.

To start with, let's extract the information we want instead of console.logging those massive objects.

```
  xhr.onload = function(ev) {
      let json = JSON.parse(xhr.responseText);
      let tasks = json.tasks;
      console.log(json);
  };
```

So now we have this json object, which contains the updated list of tasks. What we want to do is add any new tasks to the webpage. We will do this in a simple way here, though using jQuery or something similar would probably be best long term, then some nice visual effects can be added.

```
let tasks = json.tasks;
let existingCount = document.getElementsByTagName("li").length;
let list = document.getElementById("tasks");
for (let i = existingCount; i < tasks.length; i ++) {
  let el = document.createElement("li");
  el.innerHTML = tasks[i];
  list.appendChild(el);
}
```

Now try it out! Open another browser instance, and add a task to that instance. Watch as the first instance updates within a second.

Decoupling things

So now that we have something basic working, let's consider how to improve the code. Right now we put everything we want to have happen inside the onload function. This kind of gets buried in the code that tries to manage the connection. What we would like to be able to do is say "why don't you mess with all that connection stuff, and then give me the resulting list of items, then I'll do something with it". So we would *like* to be able to write the following:

```
// Won't work
let updates = useXHRandGetResults();
... do something with updates ...
```

There are two reasons why this won't quite work out. The first is that the process of remotely getting data might fail (wrong connection, timeouts etc). Our code above does not appear to handle this in any way. The other is that getting that data happens asynchronously; it will take time and we don't want our javascript code to just freeze there waiting for that to happen.

One solution to this is the so-called callbacks solution. Here's how that might look like:

```
function asyncLoad(link, onSuccess, onError) {
  let xhr = new XMLHttpRequest();
  xhr.onload = function(ev) {
      if (xhr.status == 200) {
         onSuccess(JSON.parse(xhr.responseText));
      } else {
         onError(xhr.status);
      }
  };
  xhr.open("get", link, true);
  xhr.send();
}

function updatePageWithTasks(json) {
   let tasks = json.tasks;
   let existingCount = document.getElementsByTagName("li").length;
   let list = document.getElementById("tasks");
   for (let i = existingCount; i < tasks.length; i ++) {
     let el = document.createElement("li");
     el.innerHTML = tasks[i];
     list.appendChild(el);
   }
}

function requestUpdates() {
   asyncLoad('/json',
             updatePageWithTasks,
             status => console.log("Error status: " + status));
}

setInterval(requestUpdates, 1000);
```

So we have separated the loading logic from the processing logic. We have this asyncLoad method which takes as input the webpage to load, and two functions: One function to handle a successful load, and one to handle a failed request. This is a nice separation of concerns, using the key idea of **callbacks**.

Introducing promises

This callback approach does have its limitations however. One of these limitations is that we must provide those callbacks along with calling the function, we can't decide what they will be later on. And if we chain such asynchronous requests over after the other, we can get a very nested sequence of callsbacks that becomes hard to read.

Promises are here to help with this idea. A **promise** is an object whose value hasn't been resolved yet. But it is a fully formed object and it can be passed around as any other object, until at some point later it will get resolved. We can then add handlers to that object on what should happen if and when it gets resolved.

As an example, here is a simple promise object that will get resolved in 3 seconds:

```
let o = new Promise((resolve, reject) => {
  setTimeout(() => resolve("yes!"), 3000);
});
```

By itself this does not do much. but now we can *add* handlers to the object, as follows:

```
o.then(value => console.log(value));
```

So the then part tells the promise that whenever it figures out its value it should pass that value to the console.log function.

The nice thing about it is that you can do this multiple times:

```
o.then(value => console.log(value));
o.then(value => console.log(value));
```

and now *both* callbacks will execute.

Or you can even do this *after* the object has been resolved. So for example we can do:

```
// After the 3 seconds
o.then(value => console.log(value));
```

And it will execute right away, as that object is resolved already.

Let's also look at an example with an error thrown:

```
let o = new Promise((resolve, reject) => {
  setTimeout(() => reject("something went wrong?"), 3000);
});
o.then(v => console.log("all's well."))
  .catch(err => console.log("well now:", err));
```

Notice that we *chained* the calls. In fact when we write o.then(f) this is actually again a promise. This allows you to chain a series of operations, and have a single catch at the end to account for all the things that can go wrong.

A Promise object can be in one of three states:

- **pending**, meaning it is an ongoing computation.
- **fulfilled**, meaning it has completed in a positive way.
- **rejected**, meaning an error happened.

A promise starts its life in the "pending" state, and it will move exactly once, to one of the other (final) states. We say that the promise is then **settled**.

Here's another example: We can implement a simple delay via a promise that does setTimeout, like so:

```
function delay(ms) {
    return new Promise(
      (resolve, reject) => setTimeout(resolve, ms)
    );
}

// Using delay():
delay(5000).then( () => console.log('after 5 seconds!') );
```

Promises with XHR

Now we will use promises to do the remote task reading described earlier. Here's how that might look like:

```
function asyncLoad(link) {
  return new Promise(
    (resolve, reject) => {
      const xhr = new XMLHttpRequest();
      xhr.onload = function(ev) {
          if (xhr.status == 200) {
             resolve(JSON.parse(xhr.responseText));
          } else {
             reject(new Error(xhr.status));
          }
      };
      xhr.open("get", link, true);
      xhr.send();
  });
}

function updatePageWithTasks(json) {
   ... same
}

function requestUpdates() {
   asyncLoad('/json')
    .then(updatePageWithTasks)
    .catch(status => console.log("Error status: " + status));
}

setInterval(requestUpdates, 1000);
```

Before we move away from promises, here are some useful methods that the Promise object offers:

Promise.all Given an array of promises, it returns a promise. If any of the promises in the array rejects, then the returned promise also rejects. If all promises in the array are resolved, then the returned promise also is resolved, with value the array of values from the provided promises.

Promise.race Given an array of promises, it returns a promise that settles the moment any one of the passed promises settles, and in the same way.

Async and await

Async functions are another way to work with promise objects. Basically:

1. An async function basically immediately returns a promise object then starts executing.
2. Along the way, it can use await "calls" to *wait* for another asynchronous function or promise to be resolved. It does so without blocking (i.e. other functions get to execute while it waits).

3. When the function reaches a return call, then the promise object that it returned gets resolved to that value.
4. If the function throws an exception or some other error along the way, then the promise object it returned is rejected.

As an example of this, our requestUpdates function from earlier could be written as follows:

```
// old version
function requestUpdates() {
   asyncLoad('/json')
    .then(updatePageWithTasks)
    .catch(status => console.log("Error status: " + status));
}

// new version
async function requestUpdates() {
  try {
   let json = await asyncLoad('/json');
   updatePageWithTasks(json);
  } catch(e) {
    console.log("Error: " + e)
  }
}
```

The big advantage of this is that it is written almost like a synchronous function would be written, with only difference the fact that it actually isn't: The await asyncLoad part will wait (but not block, other functions will get to run in the meantime) until the asyncLoad function has resolved its promise (i.e. the data has been read). In other words, the await part takes in a promise and waits for it to be resolved, then returns the corresponding value (or an error is raised if the promise was rejected).

As another example, with the delay function we discussed earlier we could write an asynchronous function like so:

```
// returns a promise right away.
async function f() {
  await delay(3000);  // waits 3 seconds
  console.log("done!");
  return 5;           // the promise gets resolved with value 5
}

let o = f(5); // runs right away
o.then( (v) => console.log("function done with value: " + v) ); // runs right away
// other stuff happens here
// the console.logs will happen after 3 seconds, with "done!" happening first
```