305CDE Worksheet 5

Colin Stephen

October 2014

About

This week's lab tasks extend what you were working on in last week's tasks, namely:

- Responding to asynchronous events in your program.

The difference is that we will use a programming pattern that differs from *nested callbacks* due to limitations of the latter.

We will employ a pattern that uses future-facing objects called "Promises", designed to deal with things that happen at any time in the future. See the HTML5Rocks blog post in the resources for further discussion of the type of code you will be working with in this lab.

The code here will take a while to fully understand. However, perseverance will pay off as promises are a very powerful pattern to invoke in event-based programming with JavaScript.

- They make your code more readable
- They make your code asynchronous

Resources

Tutorials

- HTML5Rocks JavaScript Promises Overview
- Using native JSON
- Getting Started with AJAX

Downloads

- ES6 Promise Polyfill code - use for reverse compatibility in non ES6-compliant browsers.
- RSVP.js Asynchronous Library - contains ES6-promises and more!

Task List

NOTE: Remember to commit your changes to a fork of the 305CDE git repository (see the Week 3 worksheet) so that you can continue later. The techniques and knowledge developed this week will be very useful later.

1. Use promises to get a series of URLs and display their content.
2. Chain promises (that get URLs) together, then post-process their data in bulk.
3. Chain promises (that get URLs) together, while processing their data in real-time.

These three stages mirror the development of code in the HTML5Rocks blog post on JS promises so you can read that too, to aid understanding.

Step-by-Step

1. Using promises to get a series of URLs

Here we begin to move from synchronous programming to asynchronous programming by introducing `Promise` objects.

- Begin with a quick review of the Week 4 Worksheet which contains analysis of `ajax_sync.html`. That file contains code which synchronously gets URLs and populates the DOM with the contents parsed from the returned JSON data.
  - To show that the AJAX get request "blocks" other operations, add the following line before the final `</script>` closing tag in the `week4/code/ajax_sync.html` file: `addTextToPage("Hi from the end of the code");`
  - Tick the "Fake network delay" box and notice that the newly added text only appears after a few seconds, once the AJAX requests have completed.

The code you will look at this week gets and processes AJAX data *outside of the function execution chain* - in other words "in the background" or asynchronously. That means your code does not get "blocked" from doing useful things while the data is still downloading.

- Open `ajax_async.html` and `js/utils.js` in Brackets and preview the functionality with 'live preview'.
- To simulate a more realistic scenario, tick the "Fake network delay" box and watch the page reload
  - Note that each chapter text arrives after some delay
- Hit F12 and load the "Network" tab to view the files being fetched using AJAX calls
- Refresh the page again
  - In the development tools, note the large gaps between the files being received (simulated network traffic).

  - This is what causes the slow chapter-by-chapter loading of the text on to the screen.

The loading of the text and its presentation in the DOM happens "chapter by chapter" here. This is similar to the sync version you looked at last week. *However, this time there is no **blocking** of other code in your program.* To see that your AJAX requests no longer block other code:

- As before, add the following line before the final `</script>` closing tag in the `ajax_async.html` file: `addTextToPage("Hi from the end of the code");`
- Refresh the page
  - Note that the newly added text appears immediately! The AJAX calls are left to execute in the background.

Understanding how promises work

Promises essentially define actions to perform once other actions complete, whether successfully or not.

- Read through the JS code in `ajax_async.html`
- Note that there is only one big chain of code:
  - begins with `getJson(url)`
  - chains various `.then()` methods as well as a `.catch()` method

The chain of `then()` and `catch()` methods indicates that the `getJson()` call returns an object with these methods available. The object returned is called a "JavaScript promise object", and the methods set out what to do *when the action associated with the promise completes successfully or fails to complete.*

- To confirm that `getJson()` returns one of these "promise" objects, look in `js/utils.js` at the definition of the function `getJson()`
- Note that it is a wrapper for a call to `get()` which also has a `then()` method.
- This indicates that `get()` returns a "promise" object. To verify this, look at the definition of `get()` and you will see that it does indeed:
  - *construct* a `new Promise()` object
  - returns a promise (actually one composed of two actions)

Test your understanding

- Add an additional `then()` call at an appropriate place in `ajax_async.html` (you will need to be careful with nesting) which pops up a confirmation dialog after each new chapter is displayed.
- The confirmation should ask whether the user wishes to continue downloading the next chapter

  - if so, progress as normal
  - if not, throw an exception, for example:

```
var err = {message: "User aborted download"};
throw(err);
```

- Throwing an `err` object like this will pass it on to the `.catch()` method down the chain, which will show the message on the screen.

2. Async Fetch Data, and Post-Process

The next file you will look at changes the order of activity a little bit. Above, the sequence was:

1. Make an AJAX call for story data
2. For each chapter in the story data:
   - Make an AJAX call for chapter data
   - Display the chapter data in DOM

However, this means each of the chapters is being fetched one-by-one. When chapter 1 has been downloaded, it is displayed, then the next call to `getJson()` is passed the URL for chapter 2.

So, while the code to fetch all the story and chapter data is asynchronous *with respect to the rest of the program*, each of the AJAX calls for chapter data is still forced to wait for the previous chapter to complete.

This is wasting resources! The browser is perfectly capable of downloading ALL of the separate chapter files *at the same time*. So, we want to adjust the promises to exploit this.

- Open `ajax_async_all.html` and `js/utils.js` in Brackets and live preview them in the browser
- Click on “Fake network delay”, hit F12 for the Network tab, and note the pattern of calls to the `chapter-n.json` files
  - This time they take place *simultaneously*!
  - Therefore the total completion time is shorter than when each was downloaded before the next.

There is one stupendously powerful block of code that lets this happen:

```
.then(function(story) {
  addHtmlToPage(story.heading);
  var chapterUrls = story.chapterUrls;
  chapterUrls = chapterUrls.map(function(url){ return "data/"+url; });
  return Promise.all(
    chapterUrls.map(getJson)
  );
})
```

- Note the final block of code executed here:
  - `return Promise.all(chapterUrls.map(getJson));`
  - This returns a promise which completes when `.all()` of the promises in its argument complete.
  - The argument maps each chapter URL to a "promise to supply some JSON" (see lecture slides later)

Basically an array of promises (each of which gets some JSON for a particular chapter) must complete before the next `then()` or `catch()` block in the chain will be executed. When the array completes, an array of the separate results is passed to the next block *in the same order*. So further blocks can process the `chapters` data returned by the various `getJson()`'s.

Test your understanding

You will have done well to understand this code by the end of the lab!

- Insert a new `.then()` block directly after the call to `getJson('data/story.json')`.
- Your new `.then()` block should create a `Promise` object that:
  - Prompts the user to break the promise or keep the promise.
  - `resolve()`s itself if the user clicked "OK", and passes the `story` on to the next `then()` block.
  - `reject()`s itself if the user clicked "Cancel", and passes an appropriate error object with a `message` key containing appropriate information.
- The `then()` block should `return` the new `Promise` object that you created.

3. Async Fetch Data, and Real-Time Process

While getting all of the chapters JSON simultaneously over the network reduces the total elapsed time to display the entire story, there is a problem. There is a (potentially) long wait until the final chapter arrives before the display in the DOM can begin.

To avoid this, it would be nice to put as much information on to the screen as possible, in the correct order, as the chapters arrive over the network.

For example, say the AJAX calls to `getJson()` complete in the following order:

$$chap_1 \to chap_4 \to chap_2 \to chap_5 \to chap_3$$

Then there is no need to wait until the end of the queue to display chapter 1 in the DOM. This sequence of AJAX responses should ideally result in the following update of the DOM:

$$chap_1 \rightarrow chap_2 \rightarrow (chap_3 + chap_4 + chap_5)$$

In other words, the user sees the first chapter as soon as it has arrived, the second chapter as soon as the *first two* chapters have arrived, the third chapter as soon as the *first three* chapters have arrived, etc. etc..

This "real time" update is what you will achieve in the final code file.

- Open `async_ajax_best.html` in Brackets and preview it with the fake network delay
- Refresh while looking at the Network tab in the browser development tools
- Note that the DOM is updating with *as much of the story as possible* when the various parts arrive.
- Even though chapter 3 may arrive last (as in the above example), the DOM would still show chapters 1 and 2 for the user to read while it was being downloaded.

How is this achieved?

- Read through the code in `async_ajax_best.html`
- Note the complex `return` statement within the first `then()` block
  - This is where all the action happens.
- The key is to use a combination of the functional programming constructs `map` and `reduce` which both operate on arrays.
- However, here *the contents of the arrays are JavaScript promises*!
  - The `map()` returns an array of promise objects corresponding to getting each of the chapters
  - The `reduce()` accumulates a chain of `.then().then()` calls attached to an initially "pre-resolved" promise object
    * This is where the use of `Promise.resolve()` comes in
    * Two `then()`s are added to the chain for each chapter in the story
  - So when the array of `getJson()` promises is fully reduced, the result is a chain of the following form:

```
Promise.resolve()
.then(/* resolve getting chapter 1 */)
.then(/* display chapter 1 */)
.then(/* resolve getting chapter 2 */)
.then(/* display chapter 2 */)
    // repeat!
.then(/* resolve getting chapter 5 */)
.then(/* display chapter 5 */);
```

Now, the special thing about this chain is that the prior call to `story.chapterUrls.map(getJson)` has *already started to asynchronously resolve all of the chapter fetches in the `.then()` blocks here.*

Complicated, but powerful.

Test Your Understanding

Try to define a map-reduce chain of promises corresponding to button click events.

- Add five buttons to an HTML page
- Define five promise objects in JS that resolve when the corresponding button is clicked (the promises will probably contain listeners - ideally "one shot" listeners that remove themselves afterwards)
  - NB: you will need to use a closure here to keep your code readable
- Create an array of these promise objects
- Apply map-reduce to the array such that the DOM is updated to display the longest sequence of buttons clicked so far (similar to the AJAX calls completing above). For example:
  - User clicks `2, 1, 3, 5, 4`
  - DOM updates `none, (1,2), (1,2,3), none, (1,2,3,4,5)`
  - User clicks `5, 4, 3, 2, 1`
  - DOM updates `none, none, none, none, (1,2,3,4,5)`
  - etc.

Don't look at the solutions on GitLab until you have tried this!