JavaScript Callbacks & Asynchronicity

Goals

- Examine callbacks in context of asynchronicity
- Understand role of promises
- Understand role of *async* / *await*
- See common patterns and how to solve with *async* / *await*

Callbacks

A callback is a function passed to another function, for it to call.

They are used for many things in JS.

Functional Programming Patterns

```
let paidInvoices = invoices.filter(
  inv => inv.owed <= 0
);
```

Useful for separating processing patterns from business logic.

Event-Driven Programming

```
$("#myForm").on("submit", sendDataToServer);
```

Register a function to be called when an event happens.

Asynchronous Code

```
setTimeout(callMeInASec, 1000);

ajaxLibrary.get("/page", callMeWhenResponseArrives);
```

Call those callbacks when asynchronous operation completes.

> NOTE **Some AJAX Libraries Use Callbacks**
>
> While axios doesn't allow callbacks as a control-flow possibility for AJAX, other AJAX libraries, like jQuery, do.

Callbacks Will Always Be Useful

First two cases will always be done with callbacks:

- Functional programming patterns
- Event-driven programming

Let's talk about third case, for handling asynchronicity

Callbacks for Asynchronicity

Why Not This?

```
// pause for a second
stopHere(1000);
// now this code runs

// get via AJAX
var response = ajaxLibrary.get("/page");
// now we have response
```

- JS is *single-threaded*, only one bit of code can run at once
- If JS *actually* stopped there, it couldn't do other things
  - Respond to events (clicks in browser, etc)
  - Repaint DOM changes in browser

So, by having a callback function for "once-async-thing-is-done", JS can finish running your code as quickly as possible.

This way it can get to those other waiting tasks ASAP.

> NOTE **Is JavaScript Single Threaded?**
>
> Largely, yes. JavaScript has "worker threads" that can do some background tasks, but these are limited in their scope. Unlike many other programming languages, JavaScript cannot run multiple top-level threads.

Callback Patterns

Sequential callbacks can lead to hard-to-understand code:

```
ajaxLib.get("/step-1", function f2(resp) {
  ajaxLib.get("/step-2", {resp.body}, function f3(resp) {
    ajaxLib.get("/step-3", {resp.body}, function done(resp) {
      console.log("got final answer", resp.body);
    });
  });
});
```

This is often called "callback hell"

You can flatten that by using non-anonymous functions:

```
ajaxLib.get("/step-1", doStep2);

function doStep2(resp) {
  ajaxLib.get("/step-2", {resp.body}, doStep3);
}

function doStep3(resp) {
  ajaxLib.get("/step-3", {resp.body}, giveAnswer);
}

function giveAnswer(resp) {
  console.log("got final answer", resp.body);
}
```

Each function needs to know what to do next; this makes writing independent functions hard.

It can be particularly hard to handle errors well for things like this.

Promises

Promises provide an alternate way to think about asynchronicity.

A promise is **one-time guarantee of future value**.

demo/pokemon.js

```
const url = `${BASE_URL}/1`;
const p = axios({ url });
console.log("first", p);   // Promise {<pending>}
```

- Promises in JavaScript are objects
- They are native to the language as of ES2015
- A promise can be in one of three states:
  - *Pending* - It doesn't yet have a value
  - *Resolved* - It has successfully obtained a value
  - *Rejected* - It failed to obtain a value for some reason
- The only way to access the resolved or rejected value is to chain a method on the end of the promise (or await it)

.then and *.catch*

- Promises provide a *.then* and a *.catch*, which both accept callbacks.
- The callback to *.then* will run if the promise is resolved, and has access to the promise's resolved value.
- The callback to *.catch* will run if the promise is rejected, and typically has access to some reason behind the rejection.

NOTE **Thenables**

When reading about promises, you'll often see a related term, called a **thenable.** A thenable is simply any object or function that has a *then* method defined on it.

By this definition, all promises are thenables, but not all thenables are promises! There are many more specifications that a promise needs to satisfy.

Here's a simple example of a thenable that isn't a promise:

```
let notAPromise = {
  fruit: "apple",
  veggie: "carrot",
  then: () => {
    console.log("I'm just a random object with a then method.");
  }
};

notAPromise.then();
// "I'm just a random object with a then method."
```

demo/pokemon.js

```
const validUrl = `${BASE_URL}/1`;
const futureResolvedPromise = axios({ url: validUrl });

futureResolvedPromise
    .then(console.log)
    .catch(console.warn);
// keeps going ...
// ...
// ...

// NFW to get that answer

// promise to be rejected

const invalidUrl = `http://nope.nope`;
const futureRejectedPromise = axios.get(invalidUrl);

futureRejectedPromise
    .then(console.log)
    .catch(console.warn);


// promise chaining
axios({ url: `${BASE_URL}/1` })
    .then(function f1(r1) {
      console.log(`#1: ${r1.data.name}`);
      return axios({ url: `${BASE_URL}/2` });
    })
    .then(function f2(r2) {
      console.log(`#2: ${r2.data.name}`);
      return axios({ url: `${BASE_URL}/3` });
    })
    .then(function f3(r3) {
      console.log(`#3: ${r3.data.name}`);
    })
    .catch(function (err) {
      console.error(err);
```

demo/pokemon.js

```
const invalidUrl = `http://nope.nope`;
const futureRejectedPromise = axios.get(invalidUrl);

futureRejectedPromise
    .then(console.log)
    .catch(console.warn);


// promise chaining
axios({ url: `${BASE_URL}/1` })
    .then(function f1(r1) {
      console.log(`#1: ${r1.data.name}`);
      return axios({ url: `${BASE_URL}/2` });
    })
    .then(function f2(r2) {
      console.log(`#2: ${r2.data.name}`);
      return axios({ url: `${BASE_URL}/3` });
    })
    .then(function f3(r3) {
      console.log(`#3: ${r3.data.name}`);
    })
    .catch(function (err) {
      console.error(err);
```

Promise Chaining

- When calling *.then* on a promise, can return *new* promise in callback!
  - Can chain asynchronous operations together with *.then* calls
- Only need *one .catch* at the end—don't have to catch every promise

demo/pokemon.js

```
axios({ url: `${BASE_URL}/1` })
    .then(function f1(r1) {
      console.log(`#1: ${r1.data.name}`);
      return axios({ url: `${BASE_URL}/2` });
    })
    .then(function f2(r2) {
      console.log(`#2: ${r2.data.name}`);
      return axios({ url: `${BASE_URL}/3` });
    })
    .then(function f3(r3) {
      console.log(`#3: ${r3.data.name}`);
    })
    .catch(function (err) {
      console.error(err);
    });
```

Benefits of Promises Over Callbacks

- Easier to write good functions
  - Each step doesn't have be tied directly to next step
  - With promises, *.then* method can just return value for next without having to itself know what comes next

- Error handling is much easier

Async / Await

async / *await* are language keywords for working with promises.

async

- You can declare any function in JavaScript as *async*
- *async* functions **always** return promises!
- In *async* function, you write code that looks synchronous
  - But it doesn't block JavaScript

await

- Inside an *async* function, we can use *await*
- *await* pauses execution
- Can *await* any promise (eg other *async* functions!)
- *await* waits for promise to resolve & evaluates to its resolved value
- It then resumes execution
- Think of the *await* keyword like a pause button

Example

demo/pokemon.js

```
async function getPokemonAwait() {
  const r1 = await axios({ url: `${BASE_URL}/1/` });
  // ...
  console.log(`#1: ${r1.data.name}`);

  const r2 = await axios({ url: `${BASE_URL}/2/` });
  console.log(`#2: ${r2.data.name}`);

  const r3 = await axios({ url: `${BASE_URL}/3/` });
  console.log(`#3: ${r3.data.name}`);
}
```

Error Handling

demo/pokemon.js

```
async function getPokemonAwaitCatch() {
  try {
    const r1 = await axios({ url: `${BASE_URL}/1/` });
    console.log(`#1: ${r1.data.name}`);

    const r2 = await axios({ url: `${BASE_URL}/2/` });
    console.log(`#2: ${r2.data.name}`);

    const r3 = await axios({ url: `${BASE_URL}/3/` });
    console.log(`#3: ${r3.data.name}`);
  } catch (err) {
    console.warn("Try again later!");
  }
}
```

Comparing *.then*/*.catch* and *async*/*await*

- Under the hood, they do the same thing
- *async*/*await* are the modern improvement
  - Code can be written more naturally
- There are a few cases where it's easy to deal with promises directly

Asynchronous Patterns

Many Calls, Do Thing On Return & Don't Block

Need to make several AJAX calls and do things *as they return*:

Promises

```
let results = [];

axios({ url: "/1" })
  .then(function f1(r1) {
    doThing(r1);
  });

axios({ url: "/2" })
  .then(function f2(r2) {
    doThing(r2);
  });

axios({ url: "/3" })
  .then(function f3(r3) {
    doThing(r3);
  });

// rest of code runs while that's happening
```

Async/Await

```
async function getAndDo1() {
  let r = await axios({ url: "/1" });
  doThing(r);
}

async function getAndDo2() {
  let r = await axios({ url: "/2" });
  doThing(r);
}

async function getAndDo3() {
  let r = await axios({ url: "/3" });
  doThing(r);
}

// don't await the calling of these
getAndDo1();
getAndDo2();
getAndDo3();

// rest of code runs while that's happening
```

Many Calls, in Sequence

Need to make AJAX calls one-at-a-time, in order:

Promises

```
let results = [];

axios({ url: "/1" })
  .then(function f1 (r1) {
    results.push(r1.data);
    return axios({ url: "/2" });
  })
  .then(function f2 (r2) {
    results.push(r2.data);
    return axios({ url: "/3" });
  })
  .then(function (r3) f3 {
    results.push(r3.data);
    // here is final list of results
  });
```

Async/Await

```
let r1 = await axios({ url: '/1' });
let r2 = await axios({ url: '/2' });
let r3 = await axios({ url: '/3' });

let results = [r1, r2, r3];

console.log(results);
```

- *Promise.all* accepts an array of promises and returns a *new* promise
- New promise will resolve when every promise in array resolves, and will be rejected if any promise in array is rejected
- *Promise.allSettled* accepts an array of promises and returns a *new* promise
- The promise resolves after all of the given promises have either fulfilled or rejected, with an array of objects that each describes the outcome of each promise.

Promise.allSettled

```
const GITHUB_BASE_URL = 'https://api.github.com';

let elieP = axios({url:`${GITHUB_BASE_URL}/users/elie`});

let joelP = axios({url:`${GITHUB_BASE_URL}/users/joelburton`});

let doesNotExistP = axios({url:`${GITHUB_BASE_URL}/user/dsdsdsds`});

let results = await Promise.allSettled([elieP, joelP, doesNotExistP]);

console.log(results);

/*
  {status: 'fulfilled', value: {…}}
  {status: 'fulfilled', value: {…}}
  {status: 'rejected', reason: Request failed with status code 404...}
*/
```

> **TIP** **Promise.all - many calls, in any order, fail fast**
>
> Make several calls, but they don't depend on each other. If one fails, the whole thing fails
>
> ```
> let p1 = axios({ url: '/1' });
> let p2 = axios({ url: '/2' });
> let p3 = axios({ url: '/3' });
>
> let resultsPromise = await Promise.all(
>     [p1, p2, p3]);
> ```

Many Calls, First One Wins

Can get answer from any call; stop after any responds:

Promises

```
let p1 = axios({ url: '/1' });
let p2 = axios({ url: '/2' });
let p3 = axios({ url: '/3' });

let answerPromise = Promise.race(
    [p1, p2, p3]);

// can `await` or `.then` this promise
```

Async/Await

```
// tricky to do with async / await
```

- *Promise.race* accepts an array of promises and returns a *new* promise
- This new promise will resolve or reject as soon as *one* promise in the array resolves or rejects

Advanced: Building Own Promises

- You can use *Promise* with the *new* keyword to make your own promises

- Unfortunately, the syntax here takes some getting used to
- *Promise* accepts a single function (call it *fn*) as an argument
  - *fn* accepts two functions as arguments, *resolve* and *reject*
  - Pass *resolve* a value for the promise to resolve to that value
  - Pass *reject* a value for the promise to reject to that value

Example: setTimeout as a Promise

Asnyc callbacks can be rewritten as promises:

demo/promiseTimeout.js

```
function wait(msec) {
  let p = new Promise(
      function (resolve, reject) {
        setTimeout(function () {
          // cause promise to resolve: any await-er will resume
          resolve();

          // we have no error conditions -- if we did, we
          // would call reject() to reject this
        }, msec);
      }
  );
  return p;
}

async function demo() {
  console.log("hi");
  await wait(1000);
  console.log("there");
}
```

Wrap Up

- Callbacks will always be useful!
  - But they're not the best way to handle asynchronicity
- Promises improve things
  - But they're much easier with *async* / *await* for most things
- Prefer using *async* / *await* over promises
  - Watch out for stuff like:

    ```
    var resp = await axios({ url: "/1" })
        .then(resp => console.log());
    ```

  - What is this awaiting?! What is resp?!

resp will be the result of *await* for the entire line–the axios call is resolved by the arrow function, which does console log the error, but returns a promise of undefined — which is then awaited and becomes the resolution of *undefined* which is assigned to *resp*.

Phew.

In other words: don't mix *await* and *.then*. It almost certainly isn't useful. Just *await* things.