CS 4220

- Current Trends in Web Development -

Application Design and Development with Node.js

Prof. Cydney Auman

AGENDA

01 Asynchronous

02 JavaScript Timers

03 Callbacks

04 Promises and Async/Await

Review

We discussed First Class functions and how functions are that are treated like any other type in JavaScript.

We looked at Higher Order functions pair with various Array methods like: `forEach()`, `map()`, `filter()` and `sort()`.

We also discussed ES6 Classes using the syntactic sugar. And we compared classes with Module Pattern focusing on private variables and functions.

Synchronous vs Asynchronous

In a **synchronous** programming model, things happen one at a time. When you call a function that performs a long-running action, it returns *only* when the action has finished and it can return the result. This stops your program for the time the action takes.

An **asynchronous** model allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result.

JavaScript code ***runs single threaded***. There is just one thing happening at a time. However, some JavaScript platforms - the browsers and **Node.js rely on the Event Loop** for asynchronous programming for anything that takes awhile to run.

JavaScript Timers

Timer methods in JavaScript are **higher-order** functions that can be used to delay or repeat the execution of some other function.

`setTimeout(fn, number)`

Calls a function to execute code after specified delay.

`setInterval(fn, number)`

Calls a function to execute code repeatedly, with a fixed time delay between each call to that function. Returns an Interval ID

`clearInterval(intervalID)`

Cancels repeated action which was set up using `setInterval()`.

**mocking delays that we will see next week from making HTTP reqs or File Reads*

Callbacks

A **callback function** is a function that is passed into another function as an argument - this callback is then invoked inside that function to perform some kind of action.

One approach to asynchronous programming is to make functions that perform slow actions use a **callback** function. The process is started, and when it finishes, the callback function is then invoked/called passing in the error and result as an argument.

TL;DR - A **callback is a function** to be executed after another function is done executing.

Callback Syntax

Generally, in callbacks are structured with a set style and format and the most popular style is called **"error-first" callbacks**.

The **first argument** of the callback is reserved for an **error** object. If an error occurs, it will be returned in the first argument.

When using the error argument, the **second argument** of the callback is used for any **successful** response. And contains the data for that success. If no error occurred, then the error response will be set to null.

```
const downloadImage = (callback) => {
   // some long running download process
   callback(null, image);
};
const start = () => {
   downloadImage((error, success) => {
       // handle error and success
   });
};
```

Callback Hell or Pyramid of Doom

Because Callbacks are functions which are nested in another function - doing this repeatedly creates an an-pattern referred to as “callback hell”.

This anti-pattern starters to shape our code structure so that is resembles a pyramid thus many also referred to this as the “pyramid of the doom”.

This is an anti-pattern because it makes our code very difficult to understand and maintain.

To avoid this we can either modularize our functions to keep them shallow. Or Promises inherently resolve the issue and makes our code more readable.

Promises

Promises were added in ES6. A **Promise** is an object which takes a **callback**. A promise specifies some code to be executed later (as with callbacks) and also explicitly indicates whether the code succeeded or failed at its job. Promises can be chained together based on success or failure.

A Promise is in one of these states:

- *pending*: initial state, not fulfilled or rejected.
- *resolved*:   meaning that the operation completed successfully.
- *rejected*:   meaning that the operation failed.

Promise Syntax (Writing)

A **Promise** is written using the using the new keyword. A Promise accepts a function as an argument. That function takes to arguments resolve and a reject which are also functions.

The **resolve** callback functions is used to when the operation is completed successfully.

The **reject** callback functions is used when the operation has failed.

```
const downloadImage = (url) => {
   return new Promise((resolve, reject) => {
       // some long running process to download image

       if (!error) {
           resolve(image);
       } else {
           reject(error);
       }
   });
};
```

Sytnax Promises (Using)

To use a Promise that has been created. We use chaining syntax.

If Promise has been resolved successfully - `.then()` is invoked. It accepts a function as an argument and receives the results from the resolved promise.

If a Promise has been rejected with an error - `.catch()` - is invoked. It accepts a function as an argument and receives the error from the rejected promise.

```
downloadImage(url)
    .then((image) => {
        // code to handle success conditions
    })
    .catch((error) => {
        // code to handle error conditions
    });
```

Async/Await

The ES7 release added **async/await** which builds on top of Promises. This type of building is also known as "syntactic sugar".

async/await cannot be used with callbacks. Because async/await is built on top of Promises, they are also asynchronous.

The syntax used for async/await gives the look that asynchronous code behaves like synchronous code. It is important to remember that this is not the case. Just because it looks like synchronous code it is not behaving like synchronous code.

Async / Await Syntax

Using the keyword **async** before the function declaration defines it as an asynchronous function.

The **await** keyword can only be used within an async function code block. Meaning you cannot use **await** by itself. Doing so will throw an error.

The **await** is used for calling a Promised based function and then waiting for it to be resolved or rejected . The await blocks the execution of the code ***only*** inside the async function. When using async/await - we also need to use **try/catch for error handling**.

```
// async with arrow function expression
const asynchronousFunction = async () => {
   try {
       const image = await downloadImage(url);

       // code to handle success conditions

   } catch (error) {
       // code to handle error conditions
   }
};
```

Async/Await Error Handling

The most common way to handle errors when using the async/await syntax is by using JavaScripts **try...catch.**

The **try** statement consists of a try block, which contains one or more statements that will be executed.   If no exception is thrown then the results will be inside the try block.

If any part of that code execution fails then a **catch** block contains statements that specify what to do if an exception is thrown.

Promises vs. Callbacks

Promises provide a clearer way of handling asynchronous operations. Promises are really just a different syntax for achieving the exact same effect as Callbacks.

The **advantages to Promises** are increased readability and providing a way of chaining functions. With the release of ES7 the ability to apply syntactic sugar **using async/await**.

It is important to note that Promises ***do not*** remove the use of callbacks. In fact, Promises utilize callbacks in a smarter way - the *resolve* and *reject* functions are callbacks.

Resources

ES6 Features

https://github.com/lukehoban/es6features

Modern JavaScript Tutorial

https://javascript.info/callbacks
https://javascript.info/promise-basics
https://javascript.info/async-await

Mozilla Introduction to JavaScript

https://developer.mozilla.org/en-US/docs/Glossary/Asynchronous
https://developer.mozilla.org/en-US/docs/Glossary/Callback_function
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Review and Prep

Review

- Review Slides
- Review and Run Code Examples

Quiz

- Open Thursday at 2/23 at noon
- Closes at Sunday 2/26 at 11:59pm PST
- Timed 12 minutes from start of Quiz.
- 10 Questions based from Weeks 3 to Week 5

Next Class

- Read Eloquent Javascript - Chapters 20 (HTTP & File)
- Node.js Guide - Getting Started

```
console.log(‘Week 05’);
console.log(‘Code Examples’);
```