Async code

best practices

Lesson Plan

- problem 1 - dealing with errors in async code
- problem 2 - callback hell
- Promise
- async await
- async.js

Lesson summary:

https://www.nerdeez.com/articles/node/common-async-problems

Error on async

- async callbacks will usually get an error instance if something went wrong
  - example fs
- We can deal with the error in the callback function
- what if we want to deal with the error outside the callback function?
- can we wrap the async code in a try and catch blocks?

callback hell

- lets do the following task
  - create a setTimeout which runs after a few seconds
  - in the timer callback read a file content which contains a URL
  - after you get the URL query a server with this url and print the result to the console
- multi level nested code is a sign in node for bad pattern and its referred to as **callback hell**
- additionally we would like to deal with the errors on the callback.
- We will categorize those errors to the following
  - critical error - if happened we should stop the execution of the other async code
  - non critical - we can still continue executing the other async code
- We can run the async code in a few ways:
  - parallel - all the async code runs together and we would like a callback when everyone is finished
  - waterfall - each async code is run after the previous one (and the previous one optionally pass the data)
  - combination - some async and some waterfall

Async problems

- to learn how to deal with async problems we will need additional tools for dealing with async code

We will cover those tools and patterns in the next section

Promise

- promise is a class which wraps our async code
- the class constructor gets the async function and runs it synchronously
- In the async function you call resolve or reject when the async code returns
- a promise can be in one of the following states
    - pending
    - resolved
    - rejected
- a promise can resolve or reject once
- you can attach a listener to the promise with the method **then**
- the listeners will always be async and can be called on promise rejection or resolve
- promise is relatively new in node (before we used external libraries) but now that it’s officially in node, the built-in modules are starting to support promise based API’s

Promise chaining

- **then** method will return a promise
- in the **then** method you can return a value and that value is what the next promise will contain
- if returning a promise, the promise will contain the value in that promise
- this allows us to chain **then's** together, something referred to promise chaining
- if one of the promise in the chain throws an error then the nearest reject method will be called, which also returns a promise.
- this allows us to jump our code to the end of the chain if there is an error (critical error) or continue running the async code (non critical error)
- there is a shot method on promise for reject function called **catch**

Parallel

- chaining promises together will run them as waterfall
- what if we want to run our async code in parallel
- **Promise.all** will get array of promises, run them in parallel return a single promise which resolves with an array of the data from all the promises
- if one promise will reject then the entire **Promise.all** will reject

EX.

- create a module which exports **setTimeoutPromise**
    - **module.exports = function(miliseconds) { … } // returns Promise<void>**
- Create a module which exports **readFilePromise**
    - **module.exports = function(pathToFile) { … } // returns Promise<string>**
- create a module which exports **httpGetPromise**
    - **module.exports = function(url) { … } // returns Promise<string>**
- try to run them in waterfall and parallel
    - for waterfall: first the timer, then read file, then the request
- when running in waterfall try to make the read file error be critical and non critical

async await

- async function is a function that returns a promise
- async function is a function that can exit and return in mid execution
- we can exit the function by using the **await somePromise** and the function will exit and reenter after promise will be resolved.
- This allows to create a more sync syntax to async promises
- in the async function you can place try and catch blocks for async code
- to return a rejected promise simply throw an error
- lets try to do the setTimeout, readfile, send request, with an async function syntax

EventEmitter - error report

- another method we can solve our async problem error reporting is by returning an event emitter instance
- another common practice is to create a class which extends event emitter.
- instead of rejecting promise we can simply throw an error and it will react the error event.

async.js

- async.js is a utility module that provides us functions for working with asynchronous javascript
- those functions are divided to 3 categories
  - collections
  - control flow
  - utils
- to understand async.js we have to first be familiar with some concepts

Node style async function

- this name is referred to a function which gets any number of arguments, and the last argument is a callback function
- the callback function first argument is an error that gets null if there is no error
- we saw this kind of function before when we worked with **fs** module

install async.js

- first lets start by install async.js using npm
    - **npm install async --save**
- There are 70 methods for dealing with popular async problems, we will not go over all of them rather we will go over the most used ones in each category

Collections

- the collections methods usually get as first argument an array.
- The second argument is a node style async function where the first argument is an item from the collection and the last argument is a callback
- we call the callback with error and arguments
- as a result we get a callback function with error as first argument

collections - filter

- we have multiple async functions, that we want to run in parallel
- we want only some async result that pass a certain condition
- we get only the ones that call the callback with true
- lets try and get an array of filenames and return only the filenames that exist

collection - map

- mapping collection from one data to another
- lets try and get a collection of filenames and convert it to the file content

Control flow

- given a collection of async functions will run them in a certain way.
- will return a certain aggregation of those functions

waterfall

- gets a collection of async functions
- run the async function one at a time according to the order in the array
- will call each function with the arguments returned from the previous one
- if error in one of the functions then it will jump to the last
- otherwise the callback fn has the result of the last function in the collection
- lets try to do an example with a collection of functions with timers

retry

- retry gets the number of retries and a node async function and will run that function until success or until retry amount is reached
- lets try to use this method to query the todo server until the request is reached.

Utils

- utils functions that usually decorate a node async function

timeout

- decorates a node async function
- will add a timeout of millisecond to a function and if callback is not called will throw a timeout error.

Summary

- now that we have a variety of tools to deal with common async problems, and we know the signs of bad patterns, we can avoid the pitfalls of async programming and write a better async code with less bugs.