Lecture 2: Modules, Applications, and Errors

CS-546 – WEB PROGRAMMING

Structure of a Node App

What is a Node app?

Throughout the term, you will be building a number of applications in Node.

Ultimately, you can view a node app as a series of scripts that are run together.

A web application, for example, would have the following scripts

- A script that, when run, listens on port 3000 for HTTP requests to retrieve data
- A second script that runs as a background process and researches information to add to the databases
- A third script that analyzes newly researched information and creates statistics on the new data

What makes an app?

Generally, your application will have the following structure in this course;

Project folder

- package.json; describes the application, and its dependencies
- node_modules/; stores all the dependencies
- app.js; initializes and runs a server, or whatnot
- views/
- static/

package.json

The package.json is a very important file that stores information about your project, such as:

- Name
- Repository
- License Type
- List of dependencies
- List of developer dependencies
- Author
- Scripts

When starting a project, navigate to your folder and use the *npm init* command to interactively create the start of that file.

When submitting assignments in this course, you must submit the package.json file and include the author field with your name.

Dependencies

It would be very difficult to reinvent every component of an application, every time you start coding.

Node allows authors to publish their code online on Github or on NPM (Node Package Manager); anyone can then download their code through the *npm* application and install it as a dependency towards a project.

References to these dependencies can be saved to your package.json file.

Dependencies are exposed in the form of modules.

You can install dependencies with the following command:

- npm install **PACKAGENAME** --save
- You must include --save to save it to your package.json file
- **Points will be deducted if you do not save all your dependencies.**

The Scripts Object

Your package.json command contains a field, “**scripts**”, that is an object containing different script tasks. For each key in the scripts object, you would have a value that contains the command for running each script (as if from the terminal).

- https://docs.npmjs.com/cli/run-script

For example, you could have a script for testing your code, and running your app, like so:

```
“scripts”: {
    “start”: “node app.js”,
    “test”: “node test.js”
}
```

You can run the **test** command by running **npm run test**.

The start command can be run with **npm run start**, or the shorthand version, **npm start**.

What are packages and npm?

Node has a *massive* repository of published code that you can very easily pull into your assignments (where applicable) through the *node package manager* (npm).

You will require the modules that your packages export, and use code that other people have created, tested, and tried. You will then use these packages to expand on your own applications and build out fully functional applications.

Installing the app

Node runs off of a series of dependencies, which are managed through the package manager, NPM

When download a node application, you will be downloading it without dependencies, so you must install them on download

- npm install
- Dependencies are stored in the node_modules folder

When you submit an assignment, you must submit it without the node_modules folder

- **Points will be deducted if you submit the node_modules folder**

Running the app

Once your dependencies are installed, you can start your app with the **npm start** command.

The **npm start** command will run the **start** script from the **scripts** object in the **package.json** file.

Setting up your application

When setting up an application, you will do the following steps:

- Make a new folder for the application
- Run *npm init* and go through the walkthrough
- Open *package.json* and update the “scripts” section and add a property of:
  - **"start": "node app.js”**
  - Where **app.js** is the name of the file you want to run on start
- Install your dependencies and save them to the package
  - *npm install **PACKAGENAME** --save*
- Write some code in your starting file
- Run app with *npm start*

See package.json for Lecture-2 Code for example

- https://github.com/Stevens-CS546/CS-546/blob/master/Lecture%20Code/lecture_02/calculator_app_example/package.json

Modules

What is a module?

Generally, a module is an individual unit that can be plugged into another system or codebase with relative ease. Modules do not have to be related, allowing you to write a system that allows many different things to interact with each other by writing code that glues them all together.

They are very flexible, and allow you to organize your code very well!

In Node.js, you will be using modules *everywhere*. In our case, a module will be a specific object (think, an instance of a class) that has certain methods and data that you can access from other scripts. You will create your first module today.

Using Git to get today’s code

By this point, you should have also installed git. Git is a version control software, that you should be able to use via your command line. As part of this course, you will be learning how to version control your software.

Through your command line, issue the following command:

```
git clone https://github.com/Stevens-CS546/CS-546.git
```

By cloning this repository (a codebase with a version history) you will make a local copy of the code in a folder called CS-546

Navigate into this folder, so that you may run the following node scripts together.

Require

There is a special, global function called *require*, which will allow you to import code from other files, packages, etc.

When you require a file / package, you will be accessing whatever the programmer assigned to be exported in that file. From there, you can use the code.

This allows you to make very small, isolated code that performs related functions.

How do I make my code ‘requirable’?

There is another global variable called `module`, which has a property called `exports` on it.

When you require a file / package, it will take whatever is assigned to the `module.exports` variable in a package. You can export anything you want: a function, a number, or an object that allows you to do any combination of these things.

You can see an example of this in the `calculator_module_example/calculator.js` and `calculator_module_example/app.js` files.

You can use those files in order to get started making your own modules!

Why would I use modules?

The more unrelated code you have together, the messier your application will become and the harder it will be to maintain.

- You can have accidental name collisions
- It becomes harder to follow what the related components are in a large file
- It becomes less readable overall

Modules allow for many great things:

- You can strictly define what code is exported to be used, allowing you to make entire files with a defined structure
- You can change the internal workings of a module to make it more performant and add more features while not updating external code.

Error Checking Module Methods

Error Checking

Modules are intended to be fundamentally nuclear, and should be designed to act alone. One of the most important aspects of this nuclear design is that each method exported in a module should have full error checking for that.

You should make sure each method checks that the arguments passed are valid in many ways:

- Check that arguments are provided (check if undefined)
- Check that arguments are of the expected type (use typeof operator)
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof
- Check that arguments are within proper bounds (i.e., if you are writing a division method, make sure that you cannot divide by 0)

Throwing

When a method is given bad inputs, to prevent the method from running, you want to use the **throw** operator to stop execution of the current function with a user defined exception.

In **JavaScript**, you can throw any type. You can throw strings, numbers, booleans, objects, errors, or anything else.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw

By default, native JavaScript methods will throw an object that is an instance of the Error object.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

Catching Errors

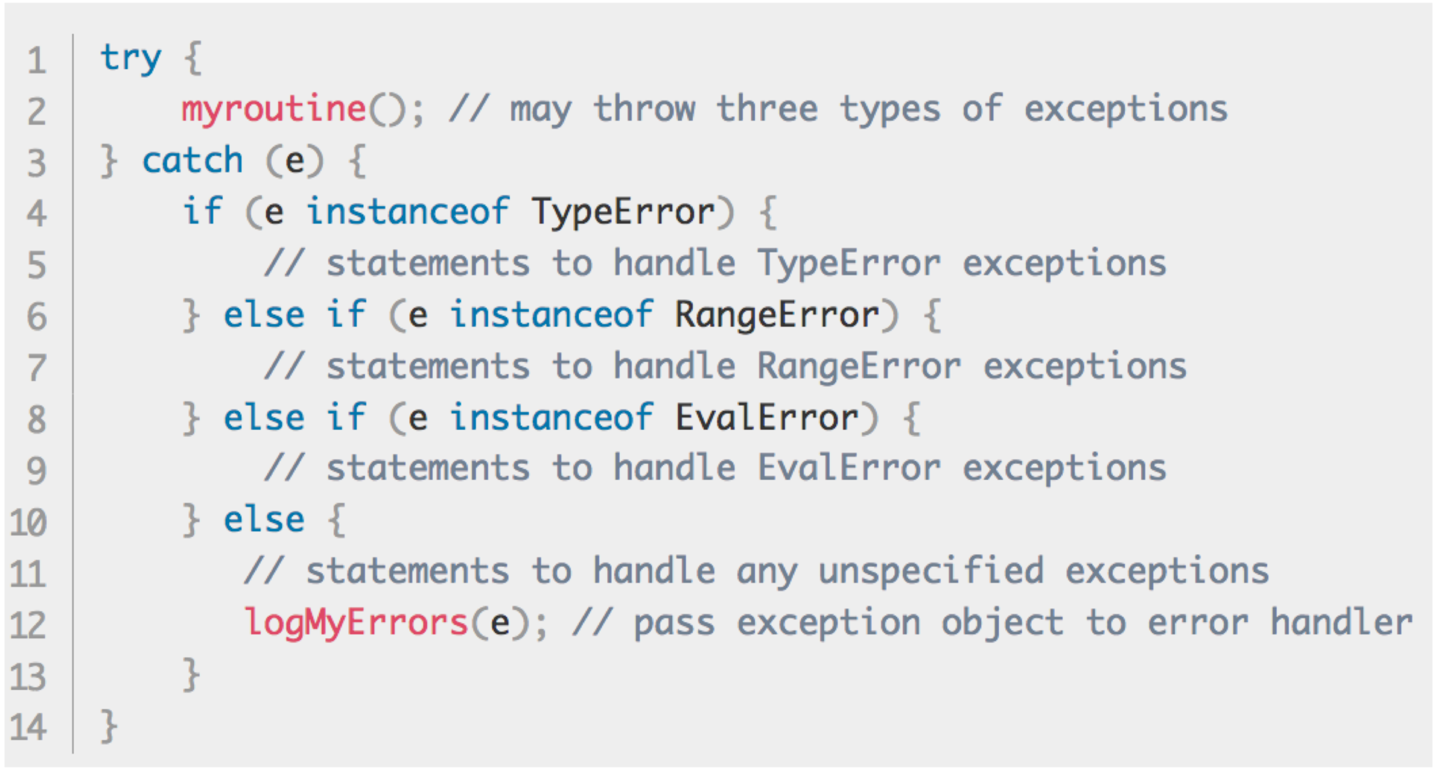
You can catch errors by surrounding the methods that you call in try / catch blocks.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

You do **not** want to catch the errors inside of your methods (unless your method can recover from errors).

- A recoverable error inside your method would be catching a failed file-save operation, catching that error, checking if it was because the filename was already in use, and changing over to save to a new filename.
- You would **not** want to catch the errors that **you** throw from your method, otherwise the developer running your code would never be able to tell that an error occurred.

You can catch particular error types by using the **instanceof** operator inside your catch statement.



```
try {
    myroutine(); // may throw three types of exceptions
} catch (e) {
    if (e instanceof TypeError) {
        // statements to handle TypeError exceptions
    } else if (e instanceof RangeError) {
        // statements to handle RangeError exceptions
    } else if (e instanceof EvalError) {
        // statements to handle EvalError exceptions
    } else {
       // statements to handle any unspecified exceptions
       logMyErrors(e); // pass exception object to error handler
    }
}
```

Making custom errors

While you can throw any data type, you may find it useful for your method to throw different types of errors for different exceptions

- You can make an ArgumentError for when your method is passed invalid arguments.

To make a custom error, you would extend the Error object with custom data so that you can check that particular type of error.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#Custom_Error_Types

Making a Calculator Module

The goal of your module

The goal of your module is simple: take numbers and perform a basic numerical operation on them.

The module should not concern itself with things like getting user input, but it should concern itself with arguments that are valid.

You can see an example of a calculator module on our code for this week

- https://github.com/Stevens-CS546/CS-546/blob/master/Lecture%20Code/lecture_02/calculator_module_example/calculator.js

Errors to Check For

addTwoNumbers(first, second)

- Check that you are provided with 2 numbers

subtractTwoNumbers(first, second)

- Check that you are provided with 2 numbers

multiplyTwoNumbers(first, second)

- Check that you are provided with 2 numbers

divideTwoNumbers(numerator, denominator)

- Check that you are provided with 2 numbers
- Check that denominator is not 0

Exporting Methods

You export methods by attaching properties to the `exports` / `module.exports` global object.

When you require this file, you will be given a copy of this object.

- https://github.com/Stevens-CS546/CS-546/blob/master/Lecture%20Code/lecture_02/calculator_module_example/calculator.js
- https://github.com/Stevens-CS546/CS-546/blob/master/Lecture%20Code/lecture_02/calculator_module_example/app.js

Writing a quick test driver

You can require your module by using a relative path in the require function:

- const calc = require('./calculator.js')

You can then test it by using exported methods.

- https://github.com/Stevens-CS546/CS-546/blob/master/Lecture%20Code/lecture_02/calculator_module_example/app.js

Making a Calculator App

Bringing it all together

We can now go through the following steps together:

- Make a new folder
- Create a file, app.js
- Run npm init and go through the instructions
- Use app.js as your entrance point
- When done, install the prompt package
  - npm install prompt --save
- Require your module in app.js
- Require prompt
- Begin creating your application!
- Run your app when ready
  - npm start

Using prompt

Many packages we use this term will require you to read a little documentation.

The *prompt* package is a simple package that allows you to easily get information from the terminal

- https://www.npmjs.com/package/prompt

You may find the *types* example on their Github repository very helpful

- https://github.com/flatiron/prompt/blob/master/examples/types.js

Questions?