

Lecture 2: Modules and Applications

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
- server.js; initializes and runs a server
- views/
- static/

package.json

The package.json stores information about your project

- Name
- Repository
- License Type
- List of dependencies
- List of developer dependencies
- Start command

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

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.**

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**

Using the app

Once your dependencies are installed, you can start your app

- `npm start`
- This will scan your `package.json` folder and run the file that is registered as your application entry point

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-WS-Summer-1/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-WS-Summer-1.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-WS-Summer-1

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 `my_module.js` and `requiring_my_module.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.

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-WS-Summer-1/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-WS-Summer-1/blob/master/Lecture%20Code/lecture_02/calculator_module_example/calculator.js
- https://github.com/Stevens-CS546/CS-546-WS-Summer-1/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-WS-Summer-1/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?
