Lecture 3: Requests and Responses

CS-546 – WEB PROGRAMMING

Recap and Intro

Recap

JavaScript is a loosely typed language

- Node.js is just a way of running JavaScript on your system

There are a number of inbuilt string and math functions that are easy to use

JavaScript has functional scoping

We often use modules to contain code.

This week

Package Management in Node.js

Requests

- What can we request?
- How can we make a request in Node.js?

Responses

- What is REST?
- Using an Express.js server
- Responding to requests
- Returning JSON

Lecture 3 Repositories

Lecture Code

- `git clone https://github.com/Stevens-CS546/Lecture-3.git`

Package Management

What is a package?

Node.js is a very community-driven language. Programmers around the world have collectively created thousands of packages that they freely distribute, most popularly through the Node Package Manager (npm).

A package is a file or direcotry that is described by a package.json file. You can see a more technically correct explanation on the npm website.

- https://docs.npmjs.com/how-npm-works/packages

Many packages are usable via a module, but do not have to be! That means you can `require` many packages you will be using.

There are two types of packages you may install:

- Local packages, which are installed for your project / current folder
- Global packages, which you install to be used throughout your entire system; many of these come with command line tools to use them, such as the popular Gulp tool, or TypeScript.

package.json

For each project you create, you will be creating a package.json file. This file contains details on your current project, as well as its dependencies. This will allow you to distribute your project to others (like myself, CAs, etc) in a way that allows us to install your dependencies easily.

By following the JSON format, this data is very readable, both for humans and other programs.

You can create a package for your current project by navigating to your project's directory and issuing the `npm init` command. This will walk you through some very basic info about your package.

After you've initialized your package for that project, you can begin installing dependencies.

Dependency Types

You can't possibly be expected to rebuild the wheel and repeat code that's already been written and tested for each program you ever write, so we often find code that solves problems we have and use them as dependencies. These dependencies are stored in the `node_modules` folder.

When your program's functionality requires a package, you can install it as a normal dependency. You would use npm to save the package as a dependency in your package.json file with the following command:

- `npm install packageName --save`

Some dependencies are only for the actual *creation* of your code, so you would install it as a dev-dependency with the following command:

- `npm install packageName --save-dev`
- This is useful to do things like install testing frameworks or code compilers; someone who wishes to contribute to your code would want to install dev-dependencies, while someone who just wishes to use it would only want the normal dependencies.

What are some packages we will be using?

Express.js

- A simple web framework that allows you to easily create a server.
- http://expressjs.com/
- `npm install express --save`

Request-Promise

- A package that allows you to easily make HTTP requests using a Promise API
- https://github.com/request/request-promise
- `npm install request-promise --save`

How do I reinstall packages?

Traditionally, when you clone a project (like you will be doing for later parts of the lecture, and lab) there is a special file called a `.gitignore` that forces the `node_modules` folder to be hidden from the repository.

- This means that when you clone a package, attempting to run the project will result in all the dependencies missing.

We mentioned before that the `package.json` file was used in order to store a list of dependencies. You can reinstall dependencies with the following command:

- `npm install`

Testing npm

First off, clone lecture 3's code from the lecture 3 repository:

- `git clone https://github.com/Stevens-CS546/Lecture-3.git`

Now that you've cloned it, navigate to the projects folder and run the following command:

- `npm install`

This may take a few seconds, but once it's done you will be able to run your first server!

- `node server_test.js`

When you run that file, you'll notice two things:

- A url is printed, with a port, pointing you to your server! Go check it out!
- It remains running until you send a command to kill the process (ctrl+c, or whatnot)

Congratulations; you just setup your first server (for this course, at least).

- You may want to read through my code in your spare time; you may find some interesting tidbits. I documented it all for your convenience.

Requests

What does a request signify?

In terms of HTTP, a request is a formal way of asking for a server to perform some function.

- You can request for data to be provided in a response (ie: information about a resource)
- You can request for an action to be performed on your behalf (ie: updating a resource)

These requests are made through some form of network (internet, intranet, etc).

Requests have meta-data associated with them, through headers that are sent with each request.

What do I need to make a request?

When making a request, you need to decide a few things:

- Where is the resource located? You will need to determine the URI of the request.
- What do you want to do to the resource? You will need to determine a request verb to submit to the server to signify what you want to accomplish:
  - GET: You are requesting data that is stored at that URI
  - POST: You are requesting that the server use the message body you send in order to create a new entity
  - PUT: You are requesting that the server use the message body you send in order to update an entity
  - DELETE: You are requesting that the server deletes the data stored at that URI
- What headers do you supply?
  - Headers are meta-data about the request or the requester; such as who your user is, how long they wish to cache data for, cookies they have associated with them, etc.
- What do you want in your request body?
  - The request body is optional; generally, for this course, you will submit JSON data representing some form of data you want to create or update.

Including data in your request

You can include a ton of data in your request:

- body: provides information supplied in the request body
- headers: any number of headers that you want to send to the server
- cookies: provides an object of all cookies in the request
- params: provides information from any URL parameters you may have matched
- query: provides information supplied through querystring data

```
POST /api/perMonthRetirementSavings?
years=10&perMonth=200&interestRate=0.03 HTTP/1.1
Cookie: firstVisit=false; userId=10;
RandomHeaderName: any_value
Host: localhost:3000
Connection: close
User-Agent: Paw/2.2.9 (Macintosh; OS X/10.11.3) GCDHTTPRequest
Content-Length: 19

{"debugMode": true}
```

Understanding the request

body: we are sending a JSON string, detailing a simple object of `{debugMode: true}`

headers: the keys and values of `RandomHeaderName` and `Cookie`

cookies: we are sending cookies of `firstVisit` of `false` and `userId` of `10`

params: if a route was defined as `/api/:method`, our `method` variable would be `perMonthRetirementSavings`

query: `years`, `perMonth`, and `interestRate` are **all** query parameters that will have values of `10`, `200`, and `0.03` respectively.

```
1  POST /api/perMonthRetirementSavings?
   years=10&perMonth=200&interestRate=0.03 HTTP/1.1
2  Cookie: firstVisit=false; userId=10;
3  RandomHeaderName: any_value
4  Host: localhost:3000
5  Connection: close
6  User-Agent: Paw/2.2.9 (Macintosh; OS X/10.11.3) GCDHTTPRequest
7  Content-Length: 19
8
9  {"debugMode": true}
```

How do I send the request?

For the sake of this course, we will be using the Request-Promise package. This will make it very easy to compose and send HTTP requests.

After installing the request-promise package in your project, you can use the package very easily to make HTTP Requests of any type.

First, you will store a reference to the module that is exported by the package:

```
var rp = require('request-promise');
```

You can send a very simple request to a page by calling the rp function:

```
rp('http://www.google.com')
    .then(function(result) { console.log(result); });
```

What is the Request-Promise package?

It becomes very messy to continually nest callbacks inside of callbacks, and very hard to read. As such, an Object called a *Promise* was designed in order to make a much more sane way of handling asynchronous code.

A *Promise* will allow you to write asynchronous code in a format that syntactically mimics synchronous code in readability. Check out `test_request.js` in order to see an example of how promises work.

The Request-Promise package takes the Request package, which gives an easy way to make HTTP calls, and wraps it in the form of promises.

How can I configure my request?

When you passed a url to the request function, you were allowing the package to assume that you were making a GET request without any headers or other additional configurations.

The Request package's github page details what you can add to a configuration object

- https://github.com/request/request#requestoptions-callback

You will see an example when using `server_test.js` and `api_test.js`

Configuring our request using RP

In api_test.js, you will see the following examples:

- Setting querystring values, see `querySecondPerson`
- Sending post data, see `createNewPersonConfig`
- Using URL params, see `querySamePerson`

We will see headers and the likes in lectures to come.

How can I use my API data?

First, you'll want to run `server_test.js` in one terminal window, and open up another. Then navigate your browser to http://localhost:3000/api/people in order to see some resulting data!

By configuring our request manually, we can set request to automatically parse our response as a JSON string and into a normal JavaScript object for us.

Run `api_test.js` for an example, and you'll see it use and manipulate that data.

Responses

What does a response contain?

A response is very similar to a request; it contains headers and other metadata, as well as a body.

Responses will also have a **status code**, which indicates the state of the response. Common codes are:

- **200**: Response is okay!
- **301**: Moved permanently
- **400:** Bad Request
- **403:** Unauthorized
- **404:** Not found
- **418:** I'm a teapot! I may be short and stout.
- **451:** Unavailable for legal reason
- **500:** Internal Server Error

What is Express?

Express is a node package that is freely distributed to allow you to easily configure and run a server.

It is a minimalist an un-opinionated framework, which means it does not shoe-horn the user into structuring things in a particular fashion.

Express makes it easy to add layers between the request a user may send and when your code for routes attains that request, called Middleware. This will become later in the course as you add a user authentication system!

How can we run a server?

Let's take a look at `server_test.js` and dissect it. There are comments there that we will run through.

The notable parts are:

- You must require the *express* package
- You must then create an app as shown
  - You'll
- You will define routes
- You will setup your app to listen for requests on a port.

What is Middleware?

When you make a request, Express will run it through a series of functions that will take the request and response objects as input, and manipulate them accordingly.

This is very, very useful! You can write middleware to:

- Log every request
- Add user authentication
- Routing events
- Log errors and redirect to pages differently depending on each error
- Do anything else you can think of!

There are many third party middlewares for Express. We are using one this week, body-parser, to allow an easy way to accept JSON data in request bodies and use that info in our server code.

How can we listen for requests?

In order to setup your application for requests, you define a series of routes before you start the server.

- These routes are composed of an HTTP Verb and a path
- Routes will listen to requests and send a response

In Express, route paths are composed of regex, strings, or string patterns.

Rather than matching on highest level of specificity (read: the route with the most specific path matching is chosen), requests will be routed to the first route path they match.

How can I retrieve data from that request?

By using another package, body-parser, you can set Express to parse any data in the body of the request that your routes may be expecting to receive.

There are a number of other sources of data you can retrieve from those request; a full listing can be found on Express' API page

- http://expressjs.com/en/api.html#req

Of note, you will be using the following properties often:

- body: provides information supplied in the request body
- cookies: provides an object of all cookies in the request
- params: provides information from any URL parameters you may have matched
- query: provides information supplied through querystring data
- get('headerName'): a method to get the header a user may have supplied

How can I send a response?

In order to send your response, you must first determine:

- Was this a successful request?
  - If so, feel free to skip setting a status code of 200; it's the default Express sends
  - You can set this with the `response.status` method.
  - If not, figure out what status code to send (see: `server_test.js`)
- What headers do I need to send?
  - We will see this later on in term
  - This will be where you set cookies, metadata, etc
- What data should I send to the consumer?
  - For now, you will be sending JSON or plain text.
  - This is populated in the body, by calling the `response.send` method

API Design

What is an API?

Formally, we can now start creating APIs in Node.js.

An API (**Application Program Interface**) is a set of capabilities that allow applications to access the application you have built.

In the modern era, most APIs are designed following the REST architectural style; this makes it very easy to use.

In short, your API is how you allow the world to access and manipulate your data in a programmer-friendly manner; it's not for making content, it's for sending it back and forth between applications.

What is REST?

REST (Representational State Transfer) is a software architectural style commonly used for designing applications that are meant to be accessed across a network

- Like an API!

When designing a REST application, you do not actually transfer the object you are requesting or manipulating, but rather pass a representation of it back and forth between the client and server.

- This is often sent back and forth in JSON format.

Designing and creating a REST API

While designing your API, it is very common to split up your different object types / classes into resources. You would create at least one endpoint per resource type.

For example, say you were creating a simple To-Do list application. You would have two simple data types: users and tasks.

You would make two routes

- /api/users
- /api/tasks

Each endpoint would allow you to perform CRUD actions on that type of resource.

While designing your REST API, you will also want to keep two common code philosophies in mind: **DRY** and **DAMP**.

DRY: Don't Repeat Yourself

DRY is a philosophy that stands for *don't repeat yourself*. That means that while writing the code for your REST API, *don't repeat yourself*. In general, with coding, this is great practice.

The DRY premise is that you should keep your common code centralized, so that changes to it only have to be made in one place.

- This makes for more testable, more stable code
- This makes changes much easier

An example of keeping your code DRY:

- If you often have to pull up a blog post by id, make a data module that has a function such as **getPost(id);**
- Rather than manually checking session / cookie data to see if a user is authenticated in each route, you would write a middleware that figures that out.

Undry DRY Example

```
function readPost(id) {
  post = my_posts[id];
  if (post === null) throw new Exception("Post doesn't exist");
  return post;
}
function updatePost(id, title, text) {
  thePost = my_posts[id];
  if (thePost === null) throw new Exception("Post doesn't exist");
  thePost['title'] = title;
  thePost['description'] = text;
}
```

DRY Example

```
function getPost(id) {
  post = my_posts[id];
  if (post === null) throw new Exception("Post doesn't exist");
  return post;
}
function readPost(id) {
  return getPost(id);
}
function updatePost(id, title, text) {
  thePost = getPost(id);
  thePost['title'] = title;
  thePost['description'] = text;
}
```

DAMP: Descriptive and Meaningful Phrases

Contrary to common belief, DAMP is not the opposite of DRY; in fact, they refer to two entirely separate things!

The DAMP pattern deals with code maintainability and ease of understanding. Programmers have a tendency of being able to read what they write as they write it, but not a month later; or worse, to assume that everyone else can read what they just wrote.

Coding DAMP means that you name your variables very specifically and focus on code readability, rather than the most clever way to accomplish a task.

The DAMP pattern is **very** often seen in the descriptive language of unit tests, where you describe your actions in very plain terms.

DAMP Example

NOT DAMP

```
var data = ['hello', 'world', 'this',
'is', 'a', 'message'];

var resultStr = "";

for (var i = 0; i < data.length; i++) {

  resultStr += data[i] + " ";

}
```

VERY DAMP

```
var wordsInMessage = ['hello', 'world', 'this',
'is', 'a', 'message'];

var combineEachWord = function(messageList) {

  var fullText = "";

  messageList.forEach(function(messageWord) {

   fullText = fullText + messageWord + " ";

  });

  return fullText;

};

var completeMessage =
combineEachWord(wordsInMessage);
```

How do we interface with a REST API?

Whenever you interface with a REST API you will be following the format of:

- Determine the resource location (URI)
- Determine the action you want to perform on said resource
- Send a network request
- Receive a **representation** of the resource / action you have performed.
- Use that data as you need to; you are now using a copy, and will not be affecting the original data (until you make other requests that may affect the data!)

How would I use an API?

When you want your server to access another API, you would make a request from your server to the external server and interpret the result

- You would then use that data, store it locally, etc.

From your product's frontend, you would use an AJAX request to access your own API

- You would set GET and POST data depending on the state of your page
- You would then interpret the result and update the page accordingly.

Accessing REST API's always follows that basic format

- Commonly, you have to set some form of authentication, as well

Preparing for Next Week

Readings For Next Week

Read up on HTML

- https://developer.mozilla.org/en-US/docs/Web/HTML

Afterwards, read up on the DOM

- https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Lab 2

Lab 2

This week, you will have your first lab!

This lab is estimated to take about 60 minutes; you can find it on Canvas. You will have 24 hours to complete it.

Questions?