
GRAND
CIRCUS
·DETROIT·

NODE.JS

WELCOME BACK!

GOALS FOR THIS UNIT

1. Review Angular / JavaScript
2. NodeJS
3. Node server
   - review of functions
4. NPM
   - `package.json`

REVIEW

NODEJS

NODEJS

Since its inception, JavaScript has run in the browser. But really, that's just a contex
It defines what you can do with the language. It doesn't encompass all of what JavaScript is capable of. JavaScript is a complete language with all of the features of other mature programming languages.

NODEJS

NodeJS is just another context. By using the same runtime engine as Chrome (Google's V8 VM), NodeJS has become a platform on which we may run JavaScript code on a server.

GOODBYE, HOST OBJECTS!

Keep in mind that a lot of the browser-related objects we're used to having access to are no longer available when we remove our JavaScript from the browser! Objects like `Document`, `Window`, and `XMLHttpRequest` are known as host objects and are context-specific.

HELLO, NODE!

Follow along. Make a new file called `hellonode.js`.

```
'Hello, Node!');
```

```
node.js
```

You should see 'Hello, Node!' printed in your terminal.

WHY NODE?

BLOCKING...

Imagine you're standing in line at your favorite Coney Island. You know exactly wha you want and you'd like to be in and out quickly. Unfortunately, there's a tourist ahead of you who is more concerned about the fact that it's called a Coney Island an isn't in New York than ordering food. They're *blocking* you from doing what you went there to do.

VS. NON-BLOCKING

Now imagine the person at the counter looks up, sees you ready and waiting, and takes your order while the obnoxious tourist is making a decision. This is how Node processes requests!

EFFICIENT

Rather than running multiple threads (having more than one line at the Coney Island), Node figures out which tasks are ready to be completed so that it's not sittin and waiting around for a single process to complete.

NODE SERVER

NODE SERVER

An example of a very simple HTTP server in implemented in node.

```
uire('http');

ver(function(request, response) {
teHead(200, { Content-type: "text/plain" });
te('Hello, World');
();
);
```

NODE SERVER

```
js
```

Navigate to `http://localhost:8888` and you should see a page that says `hello world`

THE BROWSER?

I know what you might be thinking. "I thought you said we were done with the browser." We are, our javascript in this case is being executed on the *server* instead of inside the browser. Furthermore what we said about `document` and `window` holds true. Don't believe me? Try console logging out the `document` object and see what happens.

...but it's still a web server so we visit it using a browser.

FUNCTIONS REVIEW

FUNCTIONS REVIEW

Functions can be passed as arguments.

```
ord) {
word);


te(someFunction, someArgument) {
(someArgument);


hello');
```

FUNCTIONS REVIEW

Or functions can be defined in-place.

```
te(someFunction, value) {
(value);


on(word){
word)
```

CALLBACK SANDBOX Build a callback that 1. logs a variable that is local to the callback 2. logs a variable that is local to the function your callback was passed into. 3. pass another argument along with your callback (not a fcn, a string or int), and using a for loop and an array, return a value that corresponds to the passed argument.

DISSECTING OUR SERVER

So this ability to define functions in place gives us some freedom as to how we implement our server. Let's look at two different ways to do it.

HTTP SERVER AGAIN

Here is the method we used before. As you can see, we're defining this function in place.

```
uire('http');

ver(function(request, response) {
teHead(200, { Content-type: "text/plain" });
te('Hello, World');
();
);
```

HTTP SERVER AGAIN

In this example we define a function separately and then pass that function's *reference* into the `createServer` function. This example should work exactly the same.

```
uire('http');

ver(onRequest).listen(8888);

uest(request, response) {
teHead(200, { Content-type: "text/plain" });
te('Hello, World');
();
```

WHICH ONE IS RIGHT?

In this case, there's no real 'right' answer. It's more of a matter of taste and a decisio that a team will typically standardize on. If you're working by yourself, do whichevei method makes the most sense to you.

EXERCISE!

SIMPLE NODE APP

Write a simple node server that will return a favorite lyric.

REQUIRE AND EXPORTS

REQUIRE AND EXPORTS

In Node, variables and functions are only available in the file in which they are declared i.e. variables, functions, classes and class members. So assuming a file called `example.js`

```
value) {
 + x;
```

Another file cannot access the variable `x` or the function `addX`. Node's primary building block is the module which corresponds to a file. So you can think of the file `example.js` as a module in which everything is private.

REQUIRE

We've already seen how to `require` Node's built in modules like `http`. We can also use `require` to import our own modules.

```
re('./example');
```

The argument in the `require` function is a path to the file we want to import. The trailing '.js' is optional.

REQUIRE

So using this functionality we can build self-contained modules of code that only expose whatever data and functionality that we want.

But this will only work if our module is exporting something.

EXPORT

In order to expose a variable or function from a node module you must add it to the `module.exports` object.

```
value) {
 + x;

.x = x;
.addX = addX;
```

EXPORTS

Another option is to export an object.

```
ction (a, b) {


.user = User;

 = User;
```

EXPORTS

The difference ends up being in how you use it later.

```
uire('./user')

new user.User();

new user();
```

EXPORTS

This practice helps to keep from polluting the global scope with too many variables.

LAB 15

LYRIC RANDOMIZER

INSTRUCTIONS

Expand on the previous example.

- Create a new file that is just a collection of lines of lyrics from songs you like.
- Add the array to the module.exports object.
- Import the module in your main app file.
- Change your server to return a single random lyric.

or

- Alternatively, you can put the randomization logic in the module itself.


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

NPM

EXERCISE!!!

Array + Callback - Using a server.js file and an external library of your own creation, pass a callback and a filter (".jpg") to your module and return an array of all files that match that type (.jpg)

EXERCISE!!!

Given an array with two urls for two different APIs, write code that hits both APIs, one after the other, but ensures that the responses come back in the sequence in which they were sent. Once your API calls have been made, their responses will not necessarily come back in the order in which they were sent. You must write code tha ensures this is the case. (HINT: The original array index of your array of urls is crucial for solving this problem)

THE PROBLEM

Sometimes you might need a module or functionality for your application that is not built into Node. For these you can import libraries (modules) the same way we required `http`.

THE SOLUTION

But first these modules must be downloaded. Luckily Node comes with a package manager (aptly named Node Package Manager - npm). This allows you to easily download packages making them available to your project.

```
express
```

```
require('express');
```

KEEPING TRACK

You can also maintain a sort of manifest of your node app's meta-data in a file called `package.json`. NPM will also walk you through the process of bootstrapping a new application.

Try it:

You can answer the questions as they come up, if you're not sure just hit enter and it will enter a default value.

KEEPING TRACK

In the end you should end up with a `package.json` file that looks something like this:

```
Ex",
1.0.0",
": "An example of bootstrapping a node app with npm init.",
ex.js",

cho \"Error: no test specified\" && exit 1"

anye West",
ISC"
```

THE WHOLE PACKAGE

This `package.json` file acts as this manifest for your app. It stores information such as your app's version, git repo, and any dependencies. Take a look at this file for a second.

You can also add dependencies to your project.

```
express --save
```

SAVING FOR LATER

Running this command downloads the dependency but adding `--save` also saves it to the `package.json`. This makes it easier to install a fresh version of the app with a its requisite dependencies.

```
Ex",
0.0.1",
": "NPM example",
ex.js",
ames York",
MIT",
s": { /* added by npm install --save */
 "^4.12.4"
```

LAB 16

MAKE A PROJECT USING NPM

INSTRUCTIONS

- Create a new project using `npm init`.
- Google and install the npm module `mkdirp`.
- Save `mkdirp` to your project as a dependency.
- Using the mkdirp documentation create a small node script that, when run, create a new directory.

BONUS: Using the command line arguments, refine your script so that it will accept a parameter and use it for the new directory's name.


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

TOOLS

GOALS FOR THIS UNIT

1. Review
2. Preprocessors
    - Stylus
3. Taskrunners
    - Gulp

QUESTIONS?

REVIEW

WHAT'S A PREPROCESSOR?

CSS preprocessors convert whatever we write in our chosen preprocessor's syntax into the same vanilla CSS we've been writing for the past month or so.

Since we're not writing vanilla (plain) CSS anymore, we're not confined by the limits of CSS's syntax.


Toto, I have a feeling we're not in Kansas anymore.

PREPROCESSORS

COMMON FEATURES

- Variables
- Mixins
- Nested Rules
- Color functions
- Scope
- Conditionals
- Operations
- Expression evaluation

SOUND FAMILIAR?

Preprocessors extend the CSS language and allow us to think about our styles in a more programmatic way.

THE CONTENDERS

LESS

LESS is a very popular preprocessor. If you download the source files for Bootstrap you'll find the styles written in LESS.

SASS

Sass (or Syntactically Awesome Stylesheets) is a stylesheet language initially designed waaaaay back in 2006. It is also an extremely popular option.

STYLUS

The most obvious difference when writing **Stylus**, right off the bat, is the syntax. While Stylus will also compile vanilla CSS, it will also take variations that make curly braces, semi-colons, and colons optional.

SHOOTOUT!

We're going to be delving into Stylus in this course, but we also don't want you to be sheeple. Take a look at this **comparison of preprocessors** that Tuts+ put together!

STYLUS

MIXED SYNTAX

We can write our Stylus files in a mix of variations of the syntax. Even if we mix variations within the same Stylus file, the code will be compiled to the same CSS.

```
*/

C1;


ts */

B1;

 and semi-colons */

3
```

TRANSPARENT MIXINS

Unique to Stylus, transparent mixins enhance our stylesheets by allowing us to write properties normally.

```
)
er-radius: arguments
radius: arguments
s: arguments


s: 5px 10px;
```

```
er-radius: 5px 10px;
radius: 5px 10px;
s: 5px 10px;
```

NAKED VARIABLES

We don't need to prefix our Stylus variables. They may optionally be prefixed with a '$' character.

```
00px
 -(w / 2)
```

```
;
 -100px;
```

BLOCK PROPERTY ACCESS

We can access values defined within the current block by prefixing the name of the property with '@' to reference the value.

```
-(@width / 2)
```

```
;
-100px;
```

GETTING SET UP

INSTALLATION

You should have Node and npm on your machine at this point, so just run the following command:

```
stylus -g
```

COMPILING CSS

Once Stylus is installed, seeing the compiled CSS is as simple as running the `stylus` command with the files you want stylus to compile as an argument.

```
s.styl
les.css
```

COMPILING CSS

Adding the `-w` will set up a watcher that runs in the background and recompiles the stylesheets on save:

```
yles.styl
r/local/lib/node_modules/stylus/lib/functions/index.styl
les.css
les.styl
```

ON GITHUB

Want even MORE information? Check out **the project on GitHub**.

LAB 17

SIMPLE WEBSITE

INSTRUCTIONS

Create a website project with some basic CSS Styling compiled from Stylus files.

1. Set up a new project. (index.html, styles.styl)
2. Style the project using Stylus syntax.
3. Set up the project so that the CSS file is updated each time the Stylus file is saved.


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

TASK RUNNERS

WHAT ARE THEY GOOD FOR?

Task runners allow us to be THAT much more lazy in development. They take care o
the many, many menial tasks associated with writing code that aren't actually
writing code.

AUTOMATION

We can automate all kinds of things!

- Compiling CSS
- Running a suite of tests
- Minifying styles & scripts
- Compressing images
- Launching a server
- Refreshing the browser

THE CONTENDERS

BROCCOLI

Broccoli is a relatively new option that produces more concise code because it relie more on the command line for tasks' parameters. It's a promising project, but still ir a somewhat unstable Beta phase.

GRUNT

Grunt is an extremely popular tool with a huge community around it. It is a very mature tool, and there are a lot of related resources as a result.

GULP

We'll be using **Gulp** throughout the rest of this course. Gulp uses a system that involves piping files through a series of functions. It is gaining popularity and has a ton of great plugins to get us started.

GETTING SET UP

INSTALLATION: GLOBAL

We need to install Gulp globally by running the following command:

```
--global gulp
```

INSTALLATION: LOCAL

We install Gulp to be used in a specific project by navigating to that project's root directory and running the following command:

```
--save-dev gulp
```

WORTH NOTICING

We've seen the flag `--save` used in connection with npm before. When we use `--save-dev`, it operates very similarly but it saves as a developer dependency. That means that this particular library or module is needed to develop this application bu is not necessary for this app to run.

GET READY

Create a gulpfile.js and save the following code in it:

```
uire('gulp');

ault', function() {
"Ooooh! Aaaaaah!");
```

TRY IT OUT!

Run the following command:

```
sing gulpfile ~/example/gulpfile.js
tarting 'default'...
ah!
inished 'default' after 118 µs
```

TASKS

WHAT'S GOING ON?!

It's okay if you're a little confused at this point. We haven't seen JavaScript look quit like this before. It can be a little shocking at first. Not to worry!

THE BREAKDOWN

1. We first need to require Gulp itself in our project. We do this by assigning it to a variable. By convention, such variables are typically named after the plugins they represent.

```
uire('gulp');
```

THE BREAKDOWN

2. The `.task()` method is run on the gulp object to create a Gulp task.

```
uire('gulp');
```

THE BREAKDOWN

3. The first argument passed to the `.task()` method is the name being given to the task. The task can be run from the command line with `gulp` followed by the task name. If that name is `default`, the task will be run when the user types the `gulp` command.

```
uire('gulp');

les');
```

THE BREAKDOWN

4. The second argument passed to the `.task()` method is an optional array of dependencies. These are other tasks that must be run before the current task.

```
uire('gulp');

les', ['clean:styles']);
```

THE BREAKDOWN

5. The third and final argument passed to the `.task()` method is a function that tell
the task what to do.

```
uire('gulp');

les', ['clean:styles'], function() {
```

PLUGINS

EXTENDING GULP

Just as when we were discussing jQuery, Gulp plugins increase the variety of tasks we're able to complete using Gulp. We'll go over some common tasks as well as plugins we can use to accomplish those tasks.

WORKFLOW

AUTOMATING WORKFLOW

There are several more general tasks (which may even be used by other tasks) that many developers find useful. These may include:

- Refreshing the browser
- Loading dependencies
- Starting a server
- Removing unused files

REFRESHING THE BROWSER

Though it takes only a couple of seconds to manually refresh the browser, those seconds add up when you consider how many hundreds of times you might do it in a day. We can use the `gulp-livereload` plugin to help us automate this small task.

STARTING A SERVER

We can easily start a local server using the `gulp-nodemon` plugin.

```
ve', function(  ) {
ript: 'app.js' });
```

CONCATENATING FILES

We can combine multiple files into one as part of the process of making our sites loa faster using the `gulp-concat` plugin. This can be used to combine any number of files into one and then pipe it into a destination folder.

```
catScripts", function() {

ry.js',
js/reveal.js',
x/js/lightbox.js',
.js'])
at("app.js"))
.dest('js'));
```

LAB 18

SIMPLE WEBSITE WITH GULP

INSTRUCTIONS

Work in pairs!

Refactor the morning's website project by automating some of the tasks associated with it. These should include:

- Compiling the CSS
- Concatenating all JavaScript files into one, minified file

Then, research and find a new gulp plugin (one that wasn't mentioned in the slides / lecture). Use the documentation to add it to your project, implement one of its basic tasks in your gulpfile, and build a small demo of what it does.

Bonus: Get `karma` and `gulp-jasmine` working together to run unit tests in your project.


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

EXPRESS.JS

EXPRESS

Express is a web application framework for Node JS. It simplifies creating HTTP servers and creating REST APIs. Which leads us to...

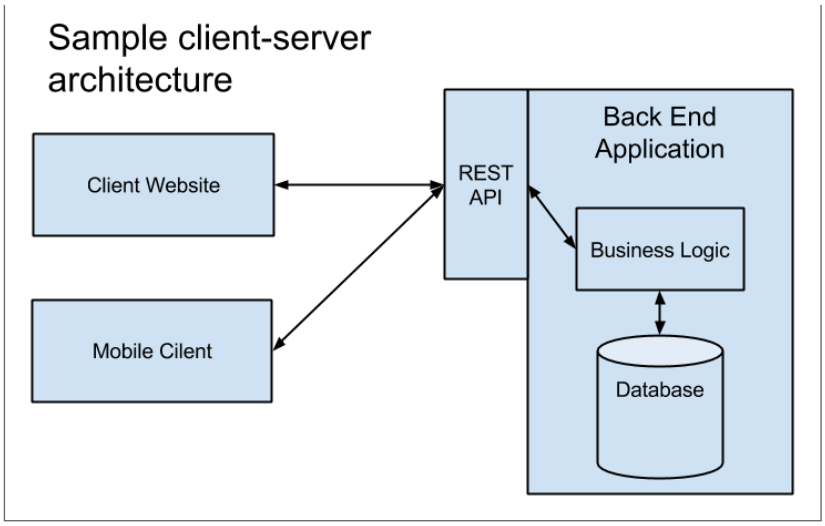
REST

REST

Representational **S**tate **T**ransfers (REST) is an **architecture style** for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used t make calls between machines.

REST relies on HTTP actions for its end points, meaning we can use all of the DOM' built in methods to interact with backend systems.

ARCHITECHTURE



EXPRESS JS EXAMPLE

```
require('express');
ess();

h "Hello World!" on the homepage
unction (req, res) {
llo World!');


pp.listen(3000, function () {
erver.address().address;
erver.address().port;

'Example app listening at http://%s:%s', host, port);
```

HTTP METHODS

- GET
- POST
- PUT
- DELETE others, but these are the big ones.

```
h "Hello World!" on the homepage
unction (req, res) {
llo World!');


 request on the homepage
function (req, res) {
t a POST request');


request at /user
', function (req, res) {
t a PUT request at /user');


TE request at /user
ser', function (req, res) {
t a DELETE request at /user');
```

POSTMAN

POSTMAN

Postman is a RESTful client that is available as a chrome webstore extension. Let's install it and take a look.

INSTALLING POSTMAN

You can just go to **getpostman.com** and it will link you directly to the chrome stor extension.

LAUNCH POSTMAN

The interface to postman can be a little confusing so we'll go through it bit by bit now.

DEMO!

LAB 19

SIMPLE EXPRESS API

INSTRUCTIONS

Create a simple REST API using Node and Express.

Test it using Postman.


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

DEPLOYING

DEPLOYING

So you've got an awesome new node app but it only runs on your laptop. Well, now we should talk about making it available to the rest of the world. Deploying our app means we will install it on a web server so that it can be accessed from anywhere on the internet.

HEROKU

HEROKU

Heroku is a platform as a service. Heroku supports a bunch of different languages, is fairly easy to use once you get used to it, and has a nice free tier for developers looking to rapidly prototype. It's a very popular service with web developers. It's what we'll use to learn to deploy applications, though there are a number of other options.

LAB 20

DEPLOY YOUR EXPRESS APP TO HEROKU

DEPLOY TO HEROKU

- Google and follow the "Getting Started with Node on Heroku" guide provided by the Heroku Dev Center.
- Once you deploy the sample app provided by the guide, replace the `index.js` with your own code and any other files you have created. Commit and re-deploy.
- Try testing your app with Postman.