# 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 context. 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`.

```
console.log('Hello, Node!');
```

```
# terminal
node hellonode.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 what 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 and 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 sitting 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.

```
// server.js
var http = require('http');

http.createServer(function(request, response) {
  response.writeHead(200, { "Content-type": "text/plain" });
  response.write('Hello, World');
  response.end();
}).listen(8888);
```

# NODE SERVER

```
# terminal
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.

```
function say(word) {
  console.log(word);
}

function execute(someFunction, someArgument) {
  someFunction(someArgument);
}

execute(say, 'hello');

// > hello
```

# FUNCTIONS REVIEW

Or functions can be defined in-place.

```
function execute(someFunction, value) {
  someFunction(value);
}

execute(function(word){
  console.log(word)
}, "hello");

// > hello
```

# EXERCISE

# ALL THE MATHS!!

Write four functions that perform basic arithmetic operations over a set of numbers (Add, Subtract, Multiply, Modulus). Store these functions in an array. Create an execute function that executes each arithmetic function with a given set of numbers (eg. 1-12). Write your program in node.

sample output executing just the add function once and passing the parameters 3 and 1.

```
3 + 1 = 4
```

sample output executing all functions in the array once

```
3 + 1 = 4
3 * 1 = 3
3 - 1 = 2
3 % 1 = 0
```

sample output of the full app, running all functions in the array on a bunch of numbers

```
1 + 1 = 2
...
12 + 12 = 24
1 * 1 = 1
...
12 * 12 = 144
1 - 1 = 0
...
12 - 12 = 0
1 % 1 = 0
...
12 % 12 = 0
```

# 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.

```
// server.js
var http = require('http');

http.createServer(function(request, response) {
  response.writeHead(200, { "Content-type": "text/plain" });
  response.write('Hello, World');
  response.end();
}).listen(8888);
```

# 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.

```
// server.js
var http = require('http');

http.createServer(onRequest).listen(8888);

function onRequest(request, response) {
  response.writeHead(200, { "Content-type": "text/plain" });
  response.write('Hello, World');
  response.end();
}
```

# WHICH ONE IS RIGHT?

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

# NPM

npm is a package manager for JavaScript. You can share and reuse packages(sometimes also called modules).

# NPM

To get things started with Node, we will start off in our projects root directory. There is an important command we will use before doing anything else

```
npm init
```

# NPM

This is going to start off a chain of prompts, which will require you to answer. Hitting `enter` will give the default answer NodeJS provides you, or you can type your own.

Once we finalize everything, check out the project folder. A new file called *package.json* has appeared!

# PACKAGE.JSON

package.json provides us with a JSON object strictly related to the current project you have started. It contains information such as the version, repository, and dependencies. Let's take a look at the structure

# INSTALLING PACKAGES

Sometimes you will want to use a library or module that isn't built into Node.

Use the command `npm install packagename` to install.

```
npm install express --save-dev

npm install gulp --save
```

# INSTALLING PACKAGES

Now that we've installed two different packages, let's check back to *package.json*

```
"devDependencies": {
  "express": "^4.14.0"
},
"dependencies": {
  "gulp": "^3.9.1"
}
```

Hey, check it out! We have two new properties inside our object

# EXERCISE!

# SIMPLE NODE APP

Write a simple node server that will return a fact about yourself.

# 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.

Assume a file `example.js`:

```
var x = 4;

function addX(value) {
  return 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.

```
var ex = require('./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.

```
var x = 4;

function addX(value) {
  return value + x;
}
module.exports.x = x;
module.exports.addX = addX;
```

# EXPORTS

Another option is to export an object.

```
var User = function (a, b) {

}

module.exports.user = User;
//vs
module.exports = User;
```

# EXPORTS

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

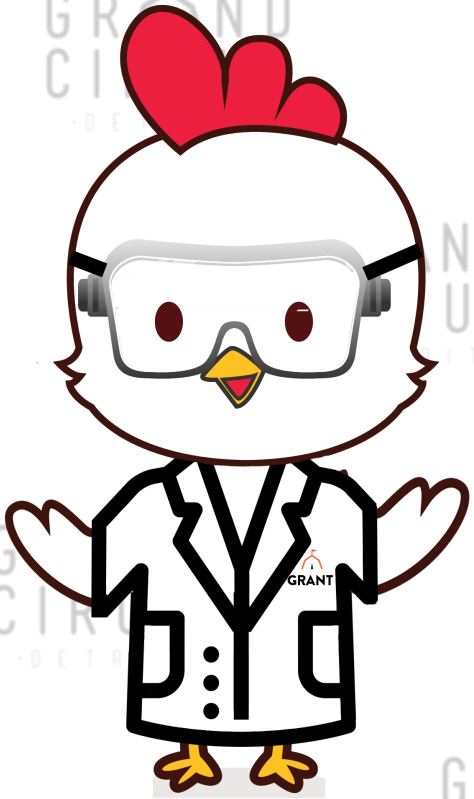
```
var user = require('./user')

var newUser = new user.User();
//vs
var newUser = new user();
```

# EXPORTS

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

# LAB 19

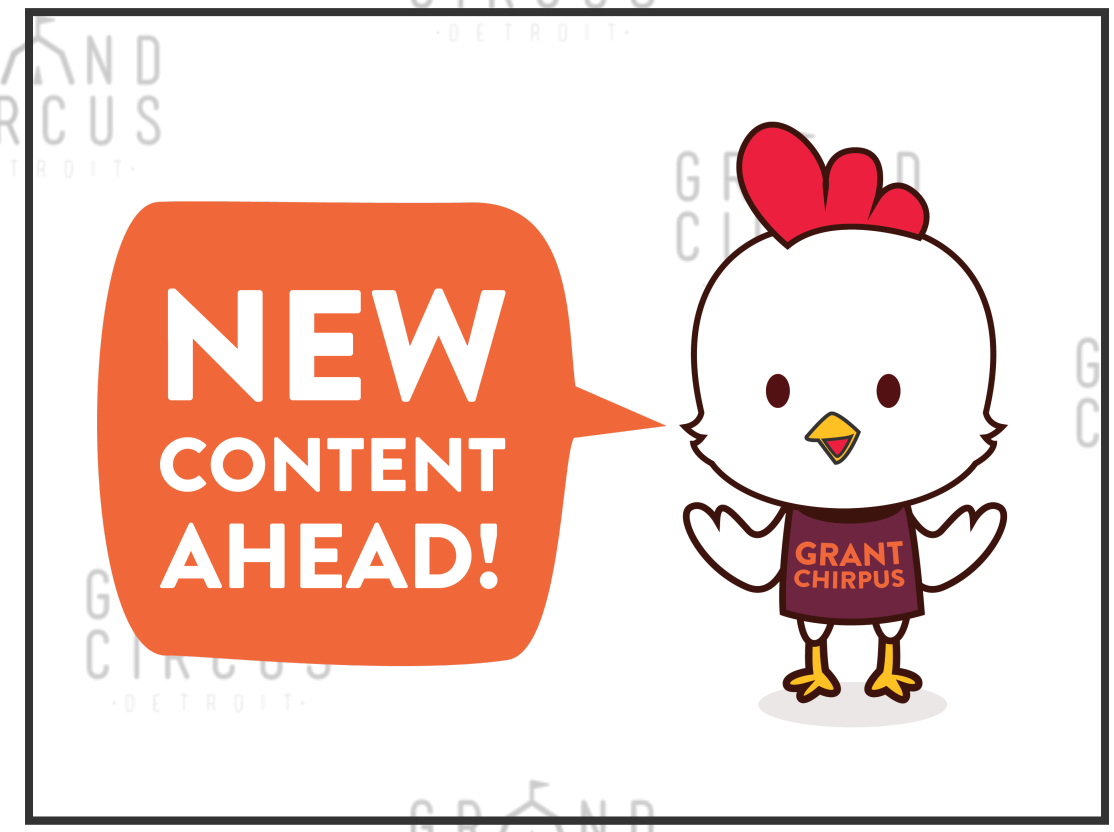
# NODE RANDOM TEXT GENERATOR



# INSTRUCTIONS

- Create a new file that is just a collection of favorite song lyrics, inspirational quotes, random facts, et. al.
- Add the collection to the `module.exports` object.
- Import the module in your main app file.
- Write a function to randomize the text pulled from the collection
- Change your server to return the random text.

# PRE-GAME


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

# GOALS FOR THIS UNIT

1. Express.js
    - REST
    - Postman
    - Full Stack JavaScript App

# 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

But first a bit about *database* concepts.

# DATABASE CONCEPTS

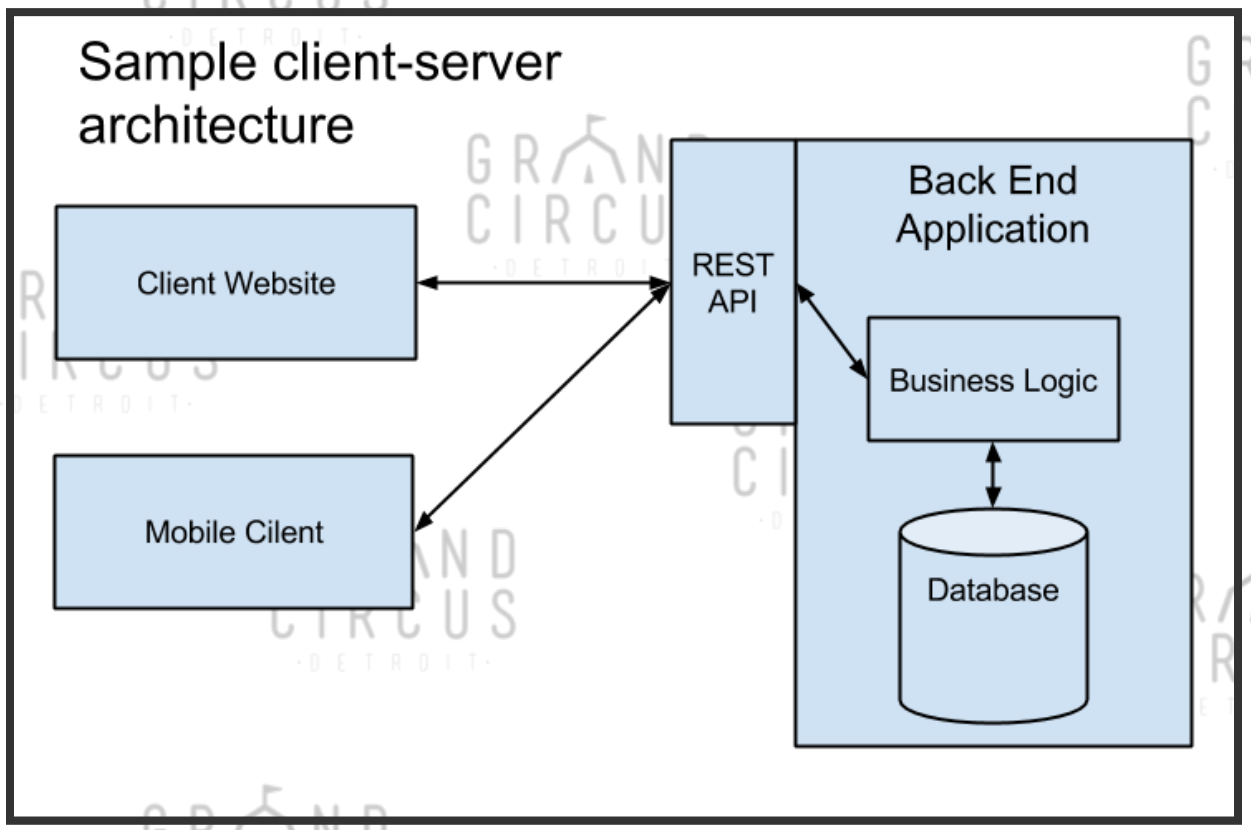
- Collections
- IDs (identifiers)

# REST

**Re**presentational **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 to make calls between machines.

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

# ARCHITECHTURE



# HTTP METHODS

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

# 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 store extension.

# LAUNCH POSTMAN

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

DEMO!

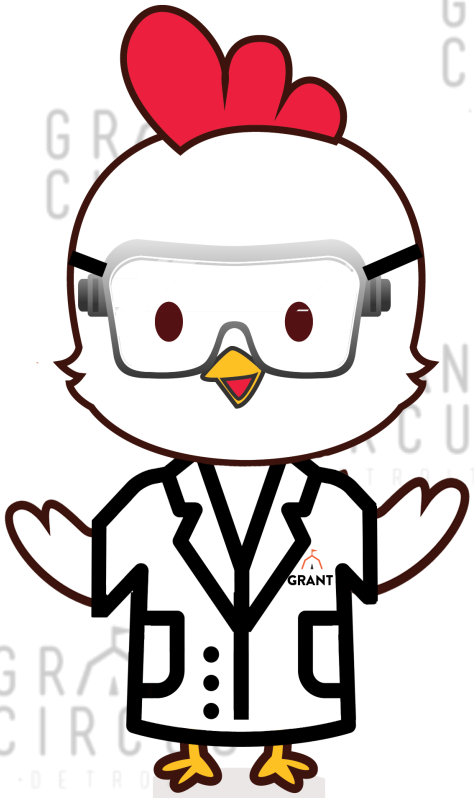
# EXPRESS JS EXAMPLE

```
var express = require('express');
var app = express();

// respond with "Hello World!" on the homepage
app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var port = server.address().port;
  console.log('Example app listening at http://localhost:%s', port);
});
```

```
// respond with "Hello World!" at /hello
app.get('/api/hello', function (req, res) {
  res.send('Hello World!');
});

// accept POST request at /hello
app.post('/api/hello', function (req, res) {
  res.send('Got a POST request');
});

// accept PUT request at /user
app.put('/api/user', function (req, res) {
  res.send('Got a PUT request at /user');
});

// accept DELETE request at /user
app.delete('/api/user', function (req, res) {
  res.send('Got a DELETE request at /user');
});
```

# LAB 21

# EXPRESS RANDOM TEXT GENERATOR



# INSTRUCTIONS

Expand on Lab 19, the text randomizer.

- Make a copy of Lab 19 in another directory.
- Convert the pure Node application to use Express.
- Add a route that retrieves the random lyric (e.g. `/api/random-lyric`).
- Add a route that retrieves the entire collection of text strings (e.g. `/api/lyrics`).
- Test both routes of your new application using Postman.

# EXAMPLE

http://localhost:3000/api/random-lyric

Response:

```
Looking for fun and feelin' groovy
```

# EXAMPLE

http://localhost:3000/api/lyrics

Response:

```
[
"I said, be careful, his bowtie is really a camera",
"Looking for fun and feelin’ groovy",
"Still a man hears what he wants to hear and disregards the rest",
"Orangutans are skeptical of changes in their cages"
]
```

# PRE-GAME

# FULL STACK JAVASCRIPT

# FULL STACK

An application's 'stack' is a jargon-y term that describes all of the technology involved in creating, deploying, and maintaining an application.

Most of this course has been concerned with the Front-End, the visible part of an application that a user interacts with. Node and Express give us a chance to peek at how Back-End programming works. When we combine the two, we're now dealing with what is referred to as "Full Stack".

# FRONT-END CONNECTED TO THE...

We can create a full stack application using express. To do this, first we need to create a directory and the appropriate files needed to work on our HTML, CSS, and JS in our project.

```
# terminal, in your express app's root directory (e.g. Lab 21)
mkdir public
touch public/index.html
touch public/main.css
touch public/script.js
```

# PUBLIC FOLDER

The `public` directory we just created isn't magic. It's just the conventional name for the Front-End in an app like this. We could call it anything we want.

# PUBLIC FOLDER

As it stands, we would need our users to navigate to the `public` folder in order to see our front end content, such as `example.com/public`. That's not really ideal so we need a way to forward the content in the public folder to our users when they visit our root domain `example.com`. Express allows to do this fairly simply.

```
// server.js, any file with a created and configured an express app
app.use(express.static(__dirname + '/public'))
```

# __dirname

*Bonus knowledge:* `__dirname` is a node-specific property that refers to the directory that the current file (whereever you use `__dirname`) resides in. This is how we direct express to the `public` folder, as it is one level below the root folder.

# PUBLIC FOLDER

Once we've set the `public` folder as the home of our front-end content, everything works in that folder like a normal website. The index page pulls in css and javascript files the way we expect them to. We are free to use any library or framework we wish (Angular, jQuery, et. al.). The front-end application in the `public` directory can be as simple or complex as we need it to be.

# FINAL FOLDER STRUCTURE

Example:

```
Root
 |-- lyrics.js
 |-- package.json
 |-- index.js
 |-- public
     |-- index.html
     |-- main.css
     |-- script.js
     └-- npm-debug.log
 |-- node_modules
     └-- express
         └-- many many files
```

# CONNECTING THE TWO

We can now make AJAX calls to our Back-End application in all the normal ways we've seen (`XHR`, `$.get`, `$http`). We simply use the RESTful API endpoints in our express app.

```
// script.js, any js file that might perform AJAX operations
var lyric = '';
$.get('/api/lyric', function(data){
    lyric = data;
});

// logic to display lyric on the page
```

# LAB 22

## RANDOM TEXT FRONT-END

# INSTRUCTIONS

Create a front-end for the application from lab 21.

- Make a copy of the Lab 21 in another directory.
- Add a `public` folder with basic front-end project files (index.html, main.css, script.js)
- Configure your express application to point to the `public` folder for this front-end content.
- On the front-end, use an AJAX call to retrieve the random text from your Express application and display it on your index page.

# PRE-GAME