Lets spin up a new project!

Before we get rolling, let's make sure that we have Node and MongoDB installed.
Let's enter in these commands in a Terminal window:

```
brew install node
brew install mongo
mkdir -p /data/db
```

If the above line does <u>not</u> work, we will perform the action as a super user:

```
sudo mkdir -p /data/db
sudo chmod 0755 /data/db
sudo chown $USER /data/db
```

It will prompt you for a password. Type in your computer password.
NOTE: It will not show what you type, this is expected behavior, trust me when I say you are in fact typing.

Create a new WebStorm Project

First, we are going to create a new project in WebStorm.

Let's give it a name, for this one, we will call it "walking_skeleton".

Initialize Node Package Manager (npm)

Now, let's open Terminal in WebStorm.

In the terminal, type in the command

```
npm init
```

This will initialize the Node Package Manager and prompt you to enter some information about the app you are going to create. Hit the return key for any values listed as (default) below.

field	**value**	**explanation**
name	(default)	Leave the NAME and VERSION alone. We will talk more about app naming and semantic versioning as we continue at Prime, for now the defaults are fine.
version	(default)	
description	`My first MEAN application`	
entry point	(default)	Leave index.js as the entry point. You read that correctly, index.js (not .html).

test command	(default)	Also, go ahead and leave test command, git repository, and keywords alone (note that we can change all of this later).
git repository	(default)	
keywords	(default)	
author	<your name>	Make sure to enter your name (gotta give yourself credit!)
license	(default)	License is fine at ISC for now.

Is this ok? Sure is, hit enter.

Package.json

Now in your project structure, look at the ‘walking_skeleton’ folder and twirl it open. You will notice a ‘package.json’ file that it created. This is information about your project that is important and we will reference it later on.

Install Express

Now we are going to install Express into the project. In the command line, enter

```
npm install express --save
```

This will install express and record it as a dependency in the package.json file, SO THAT if someone else runs your project, express will come up as something that needs to be installed.

Node_modules

Look at the new ‘node_modules’ folder that got included in the project. Inside of there, you will see an ‘express’ directory that was added. This is the Express middleware that Node will use to make life easier. Basically you can think of this as a library for Node that will make working Node a more enjoyable experience.

Create project source directories and files

Now, in the main project file (walking_skeleton), right click on the main project file and create a new directory, let’s call this folder ‘src’ (short for source). In that folder, let’s create another file that we will call ‘app.js’.

In app.js, let’s write a little code.

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

app.get( '/', function(req, res) {
     res.send('Hello!');
});

var server = app.listen(3000, function() {
      var port = server.address().port;
      console.log('Listening on port: ', port);
});
```

First we are ‘requiring’ Express. We installed it previously, so Node and its package manager are aware of Express. So we are then telling the code to go ahead and bring in Express. Once we have done that, we are creating an instance of Express called ‘app’.

From there, we are getting up a handler for when we receive a ‘get’ request to the home ‘route’. Route is a word we have only started using, but will be one that we start to use quite a bit more going forward. We then call an anonymous function that takes in two arguments, the request object (incoming), and the response object (outgoing). In the function body, we attach “Hello!” to the response and send it back. We will test this out in a moment.

Now we will set up a server. We will set this equal to the listen method on the app, taking in two arguments. The first is which ‘port’ we should have our app listen on, the next is the callback function (which is also anonymous) that just lets us know that the server is in fact up. Notice that we are setting the ‘port’ variable equal to the global server variables address method, that has a port property. Then we simply give ourselves a little message that lets us know that we are up and rolling.

Start the server

Now let’s get this up and rolling.

In the terminal, let’s start up the server. To do this, type in

```
node src/app.js
```

This tells node to go ahead and run the app.js file as the server file. If this works, we will see the ‘Listening on port: 3000’ message in the console. Now that the server is running, head over to a browser and type in ‘localhost:3000’ as the address you would like to navigate to. If this worked correctly, we should see the message ‘Hello!’.

Stop the server

To shut down the server so we can do some more development, hold down the control (not command) key and press C. This is the command to close the server.

Edit package.json

OK, so typing ‘node src/app.js’ is a little annoying. Let’s get a little fancy. Head over to your package.json file. In the scripts object, let’s add another property just below the test property. Add a comma after the test property, then press enter. In the new line, enter:

```
'start' : 'node src/app.js'
```

Basically what we have done, is alias ‘node src/app.js’ to the word ‘start’. To call it, however, we use npm instead of node.

Test alias (optional)

So the final command to start our server now is

```
npm start
```

This should now have the same functionality as before if you test everything out. Just make sure to close down the server (control + C)to begin developing again.

Create routes and views

Back in our project structure (i.e. under our ‘walking_skeleton’ directory), let’s create ‘routes’ and ‘view’ folders.

Create and edit routes/index.js

Then, in the ‘routes’ folder, add ‘index.js’. Lets cut out the ‘app.get’ code from our app.js file and move it into the ‘index.js’ file. Let’s do some more work in the ‘index.js’ file.

Here is how the index.js file should look:

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res, next) {
      console.log('Here is a console log');
      res.send('Hello!');
      next();
});

module.exports = router;
```

Once again, we are bringing in Express. Then we are also bringing in ‘Router’, which is set to a variable from a return of the router method of the express object. Routers will help us manage how incoming requests are handled. More on this later. But now, instead of calling the ‘get’ method on ‘app’, we are calling it on the ‘router’. So we set up the get method on the router object with a few more intricacies.

More specifically, we are adding another argument to the callback function of the get method. It’s called ‘next’, which has to do with how express handles middleware. (A LOT more on this later, for now, let’s keep going.) After the response sends back “Hello!”, we send back an additional ‘next();’ command, which is once again specific to Express.

As a final command, we issue an export order to the router to be a module. This makes it available to us throughout the rest of the application. Basically, we have set this up to be how we handle routes for the entire application.

Edit app.js

We need to make some additional changes in our app.js file now. The first is that we need to require our index information that we just created. So we make a variable called index and set it to our index.js module we created. Then, we are telling the application that when we get a request at the root path (“ / “) we are going to use that index variable we created. Those changes look like this:

```
...
var app = express();

var index = require('../routes/index');

app.use('/', index);

var server = app.listen(3000, function() {
...
```

Edit index.js & verify response

Let’s run another test of our code to make sure everything is working. Change “Hello!” in our index.js to be “Hello World!” This will just make sure that the response we are getting is unique to the changes we just made, and that everything is in fact, working great.

Great. We have Node and Express up and rolling!

Setup MongoDB

Now let’s work on getting Mongo all set up. Mongo is our database for this exercise, our persistent data if you will. Mongoose is a technology that allows Node to communicate with Mongo with ease. It’s a communication interface. So let’s get it installed!

Start Mongo

From a terminal, run the command:

```
mongod
```

This will turn on mongo, which we will use later. To stop mongo, press “control + c”.

Let’s install mongoose with this command:

```
npm install mongoose --save
```

Great! With mongoose installed, we can now easily communicate with our Mongo database! Over in our index.js file, let’s add a few lines of code after we declare our router variable:

```
var mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/basic_walking_skeleton');

var Cat = mongoose.model('Cat', {name:String});
```

Our first line of course is us declaring the use of Mongoose under the variable mongoose. Then we connect to a database with the connect command. We are specifically connecting to a database called 'basic_walking_skeleton' which is something we just made up right now. Finally, we are declaring a basic model for our 'Cat' data that we will be bringing into the database. Nothing super fancy, the cats will just have names for the time being.

Refactor 'src' folder

Let's go ahead and rename our 'src' folder to 'server'. In Webstorm, we need to do this a very special way. We need to use the 'refactor' option when we right click on the folder name. While not perfect, we can make Webstorm look through our application and see if there are references to the old 'src' name in which is could possibly make updates. For example, if we look in our package.json, we will see that it missed updating our 'start' command. So we will need to update that path to 'server/app.js'.

Move 'routes' folder

Next we will need to refactor our Routes folder, but this time we will perform a 'move'. Specifically, we are going to move it into the 'server' folder. This will require a change in how we target our index route.

Edit 'app.js'

So we will need to head over to the 'app.js' file and update our path to:

```
var index = require('./routes/index');
```

Rather than,

```
var index = require('../routes/index');
```

Notice the subtle difference in one '.' versus two ('..'). Go ahead and start your server, run a quick test to make sure things are going good still. Debug if needed. Shut down the server.

Install Path

Great, let's install our next technology with NPM. 'Path' is a technology that will help us serve down files from the server to the client. Let's go ahead and install that now. To do so, you will need to enter in this command into terminal:

```
npm install path --save
```

Add supporting directories and files

Next, let’s add a ‘public’ folder in the server folder. Inside the public folder, let’s create folders with the names: ‘assets’, ‘vendors’, and ‘views’. Inside of our views folder, let’s now create an index.html file. Inside of the html file, include some stub content like we have below:

```
<!DOCTYPE html>
<html>
  <head lang="en">
      <meta charset="UTF-8">
      <title>MEAN Stack</title>
  </head>
  <body>
      <div >
        <h1>MEAN STACK APP</h1>
      </div>
  </body>
</html>
```

Edit index.js

Now let’s head over to index.js and add a little code to change what is being served back to our client. So we have path installed, now we just need to hook it into our code, add this code under where we declare our router:

```
var path = require('path');
```

Then, we should replace our Hello World line with this line of code:

```
var file = req.params[0] || 'views/index.html';

res.sendFile(path.join(__dirname, '../public', file));
//next();
```

So a couple different things happening here. The first is that we are declaring file and making sure we are either setting it equal to some possible parameters coming in on the request OR, if they are not there, we are setting it equal to our index view. Then we are setting up our response to ensure it has the proper path to the needed file using sendFile with some path magic at the helm. Finally, we can comment out our next() functionality, because this will be our ending call where we send information back to the client. Let’s test. You should see your HTML rendering at this point. Close down the server and let’s move on.

NOTE: ‘next’ is there for us to augment the response in additional ways if we decided that is what we wanted to do. We will get into a little bit of this later.

Install Grunt & Uglify

Lets chat about build automation. Grunt is a technology we use to help us assemble our files. Uglify is another technology that minifies our code down. This becomes important when our applications get big. It takes down the size to something more friendly. So let’s get Grunt installed and start hooking it up!
In the console, let’s enter a couple commands:

```
sudo npm install -g grunt-cli
```

This will allow us to run grunt from the command line! Woot! Our next line is:

```
npm install grunt --save
```

Installs Grunt! Next line up is:

```
npm install grunt-contrib-copy --save
```

This is used to allow Grunt to copy files that it needs. The final line we will enter is:

```
npm install grunt-contrib-uglify --save
```

This allows Grunt to use Uglify for minification.

Install Angular

Let’s go ahead and include Angular now as well. We can do so with this command:

```
npm install angular --save
```

Create Gruntfile.js

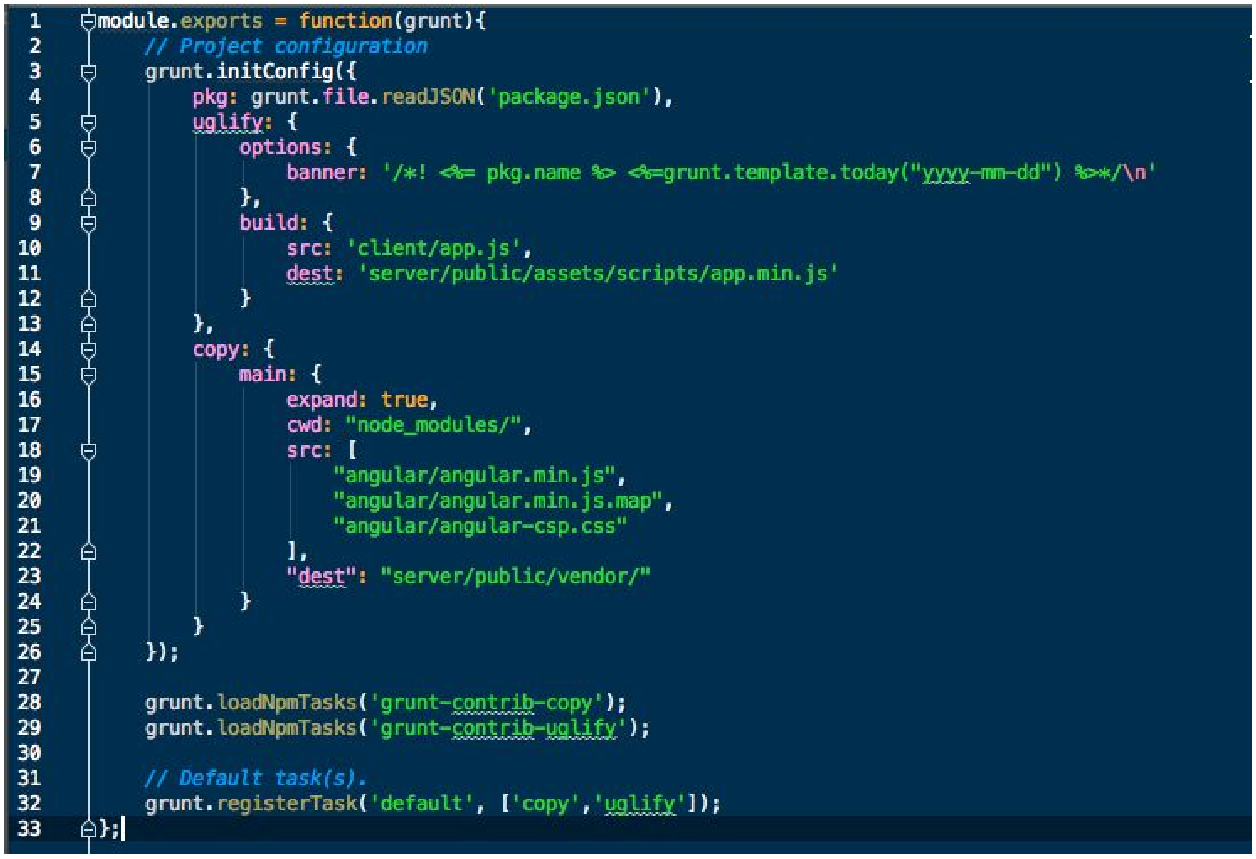
Now in our root project directory, let’s create a ‘Gruntfile.js’. The code we need to add here is extensive, so let’s add it and then chat about it on the other side:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
     pkg: grunt.file.readJSON('package.json'),
     uglify: {
        options: {
           banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
        },
        build: {
           src: 'client/app.js',
           dest: 'server/public/assets/scripts/app.min.js'
        }
     },
     copy: {
        main: {
           expand: true,
           cwd: "node_modules/",
                src: [
                    "angular/angular.min.js",
                    "angular/angular.min.js.map",
                    "angular/angular-csp.css"
                ],
                "dest": "server/public/vendor/"
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-copy');
    grunt.loadNpmTasks('grunt-contrib-uglify');

    // Default task(s).
    grunt.registerTask('default', ['copy', 'uglify']);

};
```



```
module.exports = function(grunt){
    // Project configuration
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        uglify: {
            options: {
                banner: '/*! <%= pkg.name %> <%=grunt.template.today("yyyy-mm-dd") %>*/\n'
            },
            build: {
                src: 'client/app.js',
                dest: 'server/public/assets/scripts/app.min.js'
            }
        },
        copy: {
            main: {
                expand: true,
                cwd: "node_modules/",
                src: [
                    "angular/angular.min.js",
                    "angular/angular.min.js.map",
                    "angular/angular-csp.css"
                ],
                "dest": "server/public/vendor/"
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-copy');
    grunt.loadNpmTasks('grunt-contrib-uglify');

    // Default task(s).
    grunt.registerTask('default', ['copy','uglify']);
};
```

This is our configuration file for Grunt. It starts with the Grunt.init (which stand for initialization) where we pass in an object. First we will reference the package.json file we need for the configuration. This is important, because you can see where we are specifying the Uglify options, we use that pkg reference. In the build options of Uglify, you can see where it references the start point and the destinations point. Basically, “where do I start? And where do I end?”

Down in the Copy options, you will see ‘cwd’, which stands for ‘Current Working Directory’. Here it copies the information out, then writes it to the needed directory. Specifically for Angular as that is our core to our Client side code experience.

Down in the .loadNpmTasks, we register the tasks we need when we run Grunt. That is the short of it. Basically, Grunt is doing a ton of automation for us in terms of copying and moving files. It will also minimize our code in the case with Uglify so it is more web friendly when it gets to the client.

Add client directory and supporting files

Let’s make just a couple last additions before we use Grunt. First, in your root directory, add a ‘client’ folder. In that folder, add an ‘app.js’ file. Keep it empty for now.

Run grunt

We will deep dive Grunt in the near future, so for now, just briefly understand what it is doing. I think in order to do that though, we need to see it in action. Take a look at your ‘assets’ and ‘vendor’ folder. Nothing in there. Now let’s go to the command line and enter:

```
grunt
```

Now check out those folders! Yahtzee! Files we can use client side! You will see in the assets file, we have a minified version of the app.js client side code (which is currently empty!) and then in the vendor file, we have minified versions of Angular. Not that much will be changed, but let’s go ahead and create some peace of mind. Start up your server and hit it with your browser. Nothing should be different (which, no news is good news!), so we know we are ready to move on.

Include Angular

Now that we have fancy build processes and Angular, let’s go ahead and start hooking up Angular. In our **server-side app.js** file, require path, and add the following line to serve the static content.

```
var path = require('path');
```

```
app.use(express.static(path.join(__dirname, './public')));
```

Now we can remove the file parameter from our **index.js**, since all static files now are served from the /server/public folder. Change the get method to the following:

```
router.get('/', function(req,res,next){
  res.sendFile(path.join(__dirname, '../public/views/index.html'));
});
```

Let’s head over to our index.html in our server/public/views/ folder(s). Here, let’s actually bring in Angular and our Client Side app code (currently empty) into build, so that when the files are sent back to the client, it knows that it needs Angular and our App and where to get it. So let’s add:

```
<script src="/vendor/angular/angular.min.js" type="text/javascript"></script>
<script src="/assets/scripts/app.min.js" type="text/javascript"></script>
```

The first line brings in Angular, and the second brings in our Client Side app code. Let’s go ahead and give this a build so we know things are working OK. If everything is working OK, we can check our ‘sources’ tab in our inspector in Chrome and see both Angular and our App min files loading up. Once you confirm that things look good, let’s go ahead and shut down the server.

Head back over to the index.html file and let’s change our opening HTML tag to:
```
<html ng-app="app">
```

This lets the project know that the HTML page that is being loaded is controlled by Angular. Now let’s head down to the body and add some other code:

```
<body>
  <div ng-controller="IndexController">
     <p ng-show="cat.name">Your new cats name is: {{ cat.name }}</p>
     <p ng-hide="cat.name">Enter your new cat's name!</p>

     <form novalidate>
        Name: <input type="text" ng-model="cat.name" /><br />
        <input type="submit" ng-click="add(cat)" value="Save"/>
     </form>

     <h4>Cats</h4>
     <ul>
        <li ng-repeat="cat in cats">{{ cat.name }}</li>
     </ul>
  </div>
</body>
```

The first Div is being driven by something called a “Controller”, which is an Angular specific functionality. Next, we have our P tags that are being bound to whether or not there is information present. Then, we have a form to be filled out, nothing special with the form aesthetic, but the data is being bound to a ‘model’. More on all of this later. More Angular specific technology being bound to the submit button. Finally, some repeat functionality being set up for listing multiple listings of information.

If you build it now, there is not much going on that you would not expect. Eventually, the “{{ }}” stuff will get bound to variable information. Shut down the server and let’s move to some more functionality.
Before we head over to some more code, let’s install another NPM module. In the command line:

```
npm install body-parser --save
```

This will help our application render some of the information that we are throwing at it. More on this later, but let’s head over to code.

Head over to our client folder at the root directory and open up the **app.js**. Here we are going to plug in some code:

```
var app = angular.module('app', []);
app.controller("IndexController", ['$scope', '$http', function($scope, $http){
  $scope.cat = {};
  $scope.cats = [];
  var fetchCats = function() {
    return $http.get('/cats').then(function(response){
      if(response.status !== 200){
        throw new Error('Failed to fetch cats from the API');
      }
      $scope.cat = {};
      $scope.cats = response.data;
      return response.data;
    })
  };
  $scope.add = function(cat){
    return $http.post('/add', cat).then(fetchCats);
  };
  fetchCats();
}]);
```

First, we are defining our angular app as ‘app’. Then we start to hook up our controller for how the information will be displayed. We define it as ‘IndexController’, then include some important modules needed to communicate back and forth with the server.

Down in the main part of the code, we define 1 ‘cat’ and then also, a list of cats. This will be used in conjunction with the ng-repeat later. Our fetchCats function goes and gets our cat information from our server and database and sets that equal to cats.

Down in the $scope.add function, we are ‘posting’ information to our server, which will then update the server and the database. You can see that we chain ‘then’ as another function to do after that is complete, which is just the fetchCats function (thus updating our list of info). Then we give fetchCats() a run right away when we load the app in case there are cats already existing in the database.

So, we have a way to receive cats and get them to the server, but now we need a way to write them to our database from our server. Before we do that, let’s head over to **app.js server side** and make a couple updates. Under where we require express, let’s add:

```
var bodyParser = require('body-parser');
```

Then, after where we create and define our server, let’s add:

```
app.use(bodyParser.json());
```

Then, let’s ***move*** this line of code to be right after the one we just added:

```
app.use('/', index);
```

Finally, as a last statement in that line of code:

```
module.exports = app;
```

This all allows us to use BodyParser and tightens up a few last things for use of the code throughout the rest of the application. Let’s now head over to the **server/routes folder(s)** and make some changes to the **index.js** file. Specifically, after where we define cat, add this code:

```
router.post('/add', function(request, response, next){
  var kitty = new Cat({name: request.body.name});
  kitty.save(function(err){
      if(err) console.log('meow %s', err);
      response.send(kitty.toJSON());
      next();
  });
});

router.get('/cats', function(request, response, next){
  return Cat.find({}).exec(function(err, cats){
      if(err) throw new Error(err);
      response.send(JSON.stringify(cats));
      next();
  });
});
```

These are the calls that handle GET and POST, specifically when we hit the /cats route, and the /add route. In the post call, we create a new instance of the Cat object and set the name equal to the requests.body.name (from the Angular input field). We then call the .save functionality of mongoose to toss it back to the database.

In the get call, we return the results of querying the database for everything. In the response, we send down a JSON version of the cats database.

If we start Mongo, the Server, and the application now, everything should be all hooked up! (Don’t forget to run grunt before starting the app.) Type in a name of a cat, it will go back to the server, then the database, then come all the way back and be rendered to the page.

And that is the short of it! OBVIOUSLY we will deep dive this topic by topic for the next couple weeks, but at this point, you should have a working MEAN application. If you can, I would recommend going through this guide a couple times over the next couple weeks to work toward an understanding of everything that is happening conceptually.

This is tough content, and if you made it here, pat yourself on the back (seriously). Help those around you who might be struggling with this content, it will help you understand it much better.