# Building Web Applications

**Background**: We’ll be using NodeJS as our web server and api to connect to the database and interface with the client side framework.

The server side will consist of:

* NodeJS (Adam)
  + Libraries:
    - Express
      * Used for server side routing and primarily building APIs to interface with the client side framework
    - UnderscoreJs
      * Utility library to better manage arrays, objects and other everyday functions
    - MySQL (connector)
      * Used to connect to the MySQL Database
    - Bower
      * Used for managing client side libraries
* MySQL (Trevor)
  + Database server

The client side will consist of:

* AngularJS (Walter & Vivian)
  + https://angularjs.org/
* AngularJS UI-Router (D. Chang)
  + URL Routing on the client-side
* UnderscoreJS (Austin)
  + This utility is a javascript function/utility library and can be used either server-side or client-side
  + http://underscorejs.org/
* Bootstrap (Michelle)
  + CSS and javascript scaffolding to make a great site.
  + <http://getbootstrap.com/>
* Font-Awesome (Michelle, Sparsh)
  + Awesome fonts
* jQuery (Trevor)
  + Bootstrap dependency for their javascript plugins
* Toastr (Ateev & Tim)
  + Easy notification library for application.

## Setting up NodeJS

1. Install NodeJS
   1. <https://nodejs.org/>
2. Create a directory on your OS from which to make your application.
   1. Create new directory called <name>-webapp
3. From inside the directory begin downloading all your NodeJS libraries.
   1. (Optional) You can create a packages.json file to list your dependencies and project name.
   2. If you didn’t create the packages.json file, you can manually install.
   3. Perform the following:
   4. cool-webapp:\> **npm install express**
   5. cool-webapp:\> **npm install underscore**
   6. cool-webapp:\> **npm install -g bower**
      1. The –g mean global; bower actually becomes a command. The option is library specific.
   7. cool-webapp:\> **npm install mysql**
4. You have everything to start writing your server side code.

Notes: **npm** command has shorthand functions, you can check them out by typing **npm** and enter.

## Setting up Client-side

Before we begin, it’s good to have a firm understanding of your directory layout.

* At the root of your cool-webapp folder, you should find a folder called **node\_modules**. This is where all your node modules you downloaded earlier, except for bower because that was globally installed and resides in a global node\_modules directory for your OS.
* You need to create a folder called **public** (or any name) which will become the folder where all your client-side code will be served from.
* You need to create a file called server.js at the root, where you will launch your server using node.
  + cool-webapp:\>**node server.js**

1. Now that you have your **public** folder, go inside of that folder and we will begin downloading the client-side libraries
   1. cool-webapp\public:\> **bower install angularjs**
   2. cool-webapp\public:\> **bower install ui-router**
   3. cool-webapp\public:\> **bower install underscore**
   4. cool-webapp\public:\> **bower install bootstrap**
      1. Bower already installed jQuery as a dependency.
2. Bower will create a directory called **bower\_components** inside your **public** folder, it is very similar in function to **node\_modules**.
3. Inside of your **public** folder, you should create a folder called **app**
   1. This folder will contain the Javascript and HTML pages that you build for your client-side application.

# Coding the server.js

// loading the modules, to variable names

var express = require('express');

var \_ = require('underscore');

var mysql = require('mysql');

// instantiating the express

var server = express();

// tell the server to serve everything in the public folder in your root directory.

server.use(express.static(\_\_dirname + "/public"));

// server api here, used by client-side application

server.get(‘/api/hi, function(req,res) {

res.send(‘hello world’);

});

// tell the server to start serving your pages

server.listen(8080);

# Setting up the public folder

1. Create the below directory structure inside your public folder
   1. public\app
   2. public\app\js
   3. public\app\css
   4. public\app\views
2. In the public folder create the following files
   1. public\index.html
      1. This is the first page that loads in your browser when visiting http://localhost:8080, remember in the server.js code you served all the contents of the public folder.
   2. public\app\js\app.js
      1. app.js will be where you will write your angularjs code. This javascript will be referenced in the <head></head> of your index.html file.
   3. public\app\css\site.css
      1. Site.css will be where you will have css specific to override any bootstrap css code.
   4. public\app\views\<your html pages>
      1. This folder will store all your partial HTML pages that you will serve.

# Coding your index.html

<!DOCTYPE html>

<html lang="en" **ng-app="cool-app"**>

<head>

<title>Cool WebApp </title>

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<script src="/bower\_components/jquery/dist/**jquery.min.js**"></script>

<script src="/bower\_components/angular/**angular.js**"></script>

<script src="/bower\_components/ui-router/release/**angular-ui-router.min.js**"></script>

<script src="/bower\_components/underscore/**underscore-min.js**"></script>

<script src="/bower\_components/bootstrap/dist/js/**bootstrap.min.js**"></script>

<script src="/app/js/**app.js**"></script>

<link rel="stylesheet" href="/bower\_components/bootstrap/dist/css/**bootstrap.min.css**" type="text/css"/>

<link rel="stylesheet" href="/bower\_components/font-awesome/css/**font-awesome.min.css**" type="text/css"/>

<link rel="stylesheet" href="/app/css/site.css" type="text/css"/>

</head>

<body>

<div **ui-view=”navigation”**></div>

<div style="margin-top:75px;" **ui-view=”content”**></div>

</body>

</html>

* This should be pretty straight forward index.html page.
  + Since the public folder is treated as the root folder from the browser’s perspective, none of the server-side code is exposed, such as the server.js file.
    - Only contents inside of public directory.
* In the html tag, there is an attribute called **ng-app** and has the value cool-app. This attribute is used by AngularJS, so that it understands it has processing to do for that page.
* In the body, there is a new attribute called **ui-view,** this is from the ui-router library. They are named so that it can be programmatically called in the javascript code to populate with correct HTML pages when needed.
  + For instance, the ui-view “navigation” will be populated with a navigation.html, and shouldn’t change from page to page, it is fairly static code. You don’t want/need to copy and paste ‘navigation’ code to every webpage. This would create a change nightmare, if you had a new navigation item to create.
  + The ui-view “content” will be populate based on what page the user is on.

# Coding your app.js

Note: I prefer calling my app, app. You can prefer it call it anything else.

* In your app.js, you will be primarily coding in the AngularJS framework.
* The app.js will be fairly structured and once you understand it, adding new items to it is fairly simple.

Structure:

* You will first initialize an angular.module with the name ‘cool-app’. This ‘cool-app’ value was the value from the **ng-app=”cool-app”** attribute in the html tag of the index.html page. AngularJS understands that this javascript code applies to that page.
* You will then build your angular app object with the below functions:
  + app.config
    - This function provides general AngularJs routing options
  + app.filter
    - This allows you to create custom filters, you will find more about them later. This becomes very useful when formatting values.
  + app.factory
    - This function allows you to create factory objects. This will be the primary way to communicate with the server-side API.
  + app.controller
    - This is a page specific logic. You will be creating many of them based on what you want to show on that page.

// the app variable is now an angular object

var app = angular.module('cool-app', ['ui.router']);

// configure the angular app by calling its functions

app.config(['$stateProvider','$urlRouterProvider', function($stateProvider, $urlRouterProvider) {

$urlRouterProvider.otherwise(‘/’);

$stateProvider.state(‘app’, {

abstract: true,

views: {

‘navigation’: {

templateUrl: ‘app/views/navigation.html’

}

})

.state(‘app.search’, {

url: ‘/search’,

views : {

‘content@’: {

templateUrl: ‘app/views/search.html’,

controller: ‘SearchCtrl’

}

}

});

}]);

app.factory(‘api’, [‘$http’, function($http) {

var api = {};

api.searchRunner = function(runner) {

return $http.get(‘/api/getRunner&firstname=’+runner.firstName+’?lastname=’+runner.lastName);

}

return api;

}]);

app.controller(‘SearchCtrl’, [‘$scope’, ‘api’, function($scope, api) {

$scope.runners = [];

var search = function(runner) {

api.getRunners(runner).success(function(runners, status, headers, config) {

$scope.runners = runners;

}).error(function(data, status, headers, config){

alert(‘error’);

});

};

}]);

# Coding your partial html pages:

In your main index.html, you referenced the ui-view=”navigation” and ui-view=”content”. These are connected to the $StateProvider object in your app.js. In those $StateProvider states, you specified a templateUrl under “views” object that referenced the programmatic name “navigation” and “content@”, those are the partial html pages.

## navigation.html

<nav class="navbar navbar-default navbar-fixed-top">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

<a class="navbar-brand" **ui-sref=**"app.search ">Cool WebApp</a>

</div>

<div id="navbar" class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li **ui-sref-active**="active"><a **ui-sref=**"app.search"><i class="fa fa-fw fa-server"> </i> Search</a></li>

</ul>

</div><!--/.nav-collapse -->

</div>

</nav>

* Most of the navigation code was cut from bootstrap components section.
* The key points are:
  + The attribute **ui-sref** which has the value of the $StateProvider’s states. You don’t have to specific the url of the page, only the state names.
  + The attribute **ui-sref-active** which contains the name of the css class to apply if the state is active. In the above code, I called the css class “active” which is a class from the Bootstrap library.

## content.html

The pages referenced for the “content@” states in your StateProvider follow regular AngularJs application flow. The partial html you load will have the controller you specified in the StateProvider applied to that page. You do not need to define an “ng-controller” attribute because the StateProvider already initializes it for you.