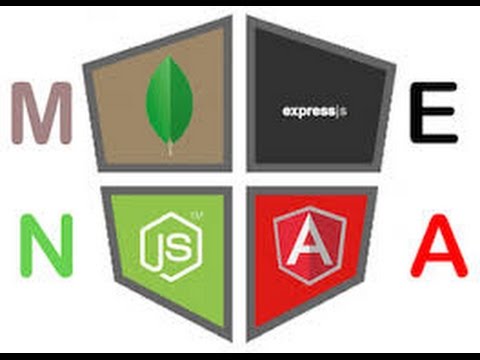
**Exercise06\_01\_01 – Step 1**

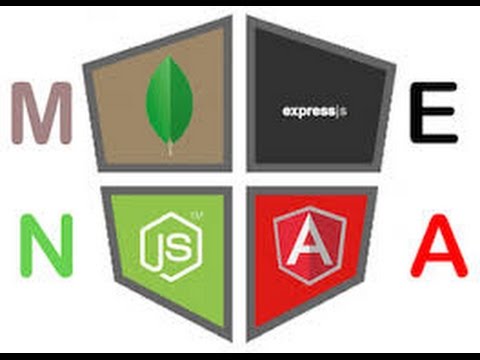
1. Create a project folder named ***Exercise06\_01\_01***. Open the project folder with your IDE. Using your IDE, or a terminal, we will first use ***npm*** to install a ***scaffolding*** tool called ***Yeoman*** in the project folder:  
   ***npm install –g yo***
2. Go to the ***Yeoman.io*** website and click on ***Discovering Generators***. We can see that there are many generators for many different kinds of projects. Up near the top, we can see that one of the most popular is called ***gulp-angular***. Click on the generator, and the documentation tells us that the generator has several ***dependencies***. We will use ***npm*** to install these in the project folder.
3. The first two we will install will be ***gulp*** and ***bower***. Gulp will provide our ***tasking*** utility. Bower is a ***package*** ***manager*** for the front end that works with AngularJS:  
   ***npm install –g gulp bower generator-gulp-angular***  
   Test this in the browser with a Live Preview, with Developer Tools on, and in the Console tab, to make sure there are no errors.
4. Build a new folder in the project folder named ***front-end*** and navigate to it in the IDE, or the terminal, if necessary. Enter the following command and respond to the questions as indicated after:  
   ***yo gulp-angular my-message-front  
   >Angular version 1.5.x  
   >Deselect all extra Angular modules to keep it simple  
   >None on jQuery  
   >None, $http is enough  
   >UI Router  
   >Bootstrap  
   >Angular UI Bootstrap  
   >None, only CSS  
   >ES6 (Babel)  
   >None, standard HTML***Now ***npm install*** and ***bower install*** will run automatically to install the necessary dependencies. This will take a few minutes and there might be a lot of warnings.
5. At the command line, run ***gulp*** to initialize everything. If ***bower*** throws any errors, run ***bower install***, then call gulp:  
   ***gulp***
6. Open up the file ***/gulp/server.js*** and look for the assignment to ***browserSync.instance***. Modify it to set the port we would like the gulp server to run on as follows:  
    browserSync.instance = browserSync.init({  
    ***port: 8081,*** startPath: '/',  
    server: server,  
    browser: browser  
    });  
   Save that file and now let’s test the app setup by calling ***gulp serve*** at the command line. This should bring up the scaffolded application in the browser. It is a Yeoman demo app, and is the starting point for our project.
7. Let’s familiarize ourselves with the app structure, which we will use as our basis:
   1. There is a ***/src/assets/images*** folder, which will contain the various images we will be using.
   2. The ***/src/app*** folder contains everything else, such as layouts, scripts, and styles.
   3. ***index.module.js*** contains our main front-end app ***definition*** , called ***myMessageFront***, and ***dependencies***, ES6 style. We can see a whole lot of new methods here for the module structure.
   4. In ***/src***, open ***index.html***, and it is nothing but bare bones code and comments. Notice that there is a ***ui-view*** directive, replacing an ***ng-view***, because we are using the ***ui.router*** instead of the AngularJS ***ngRoute*** module.
   5. ***index.route.js*** will handle our app routing, and we can see it is similar to ngRoute, with some differences in methods and terminology. In particular, it refers to routes as ***states***.
   6. ***main.html*** contains the partial that makes up the Yeoman demo.

**Exercise06\_01\_01 – Step 2**



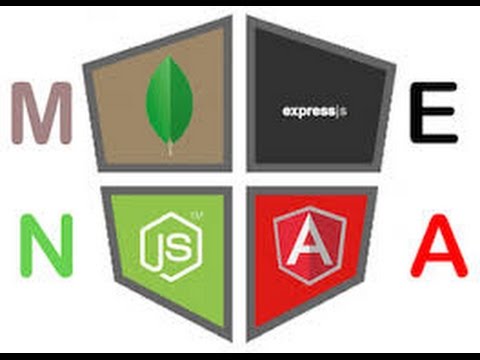
1. If ***gulp serve*** is not already running, let’s start it up in a terminal window and keep it running, it is set to be dynamic with a package called ***browser-sync***. We will start to develop our front end in the ***main.html*** partial file, most of which we can get rid of. We will just leave the outer ***<div>*** container, and the ***<acme-navbar>***:  
   ***<div class="container">  
     
    <div>  
    <acme-navbar creation-date="main.creationDate"></acme-navbar>  
    </div>  
     
   </div>***Take a look at this in the browser.
2. Now let’s add our own components to the partial, starting with another ***<div>***, below the nav bar:  
    ***<div class="panel panel-default">  
    <div class="panel-heading">Post A Message</div>  
    <div class="panel-body">  
    </div>  
    </div>***Let’s test this in the browser.
3. Let’s do some more construction of the ***panel-body*** in the ***<div>***. we will add a <div> to hold a control group consisting of a ***<textarea>*** and a custom ***button*** as follows:  
    <div class="panel-body">  
    ***<div class="input-group">  
    <textarea class="form-control custom-control"   
    rows="3" style="resize: none">  
    </textarea>  
    <span class="input-group-addon btn   
    btn-primary">Post</span>  
    </div>*** </div>  
   Let’s test this in the browser, we have a much better look and feel.
4. Let’s start to do some of the scripting work. Open ***main.controller.js***. We can see that we have moved forward to ***ES6*** syntax, using a ***class*** to build our ***Controller***, which has a ***constructor*** to build it. It also uses ***ngInject*** to clean up the ES5 syntax of controller dependency injection. To review, the ***Controller*** will be used to ***bind*** things to our **View**. Our object here will be to bond a post ***function*** to our post ***button***. First we need to clean up some of the generator code. Let’s get rid of all of the ***functions***, and the ***parameters*** and ***property*** assignments of the constructor:  
   ***export class MainController {  
    constructor() {  
    'ngInject';  
     
    }  
     
   }***
5. Now we can add a new ***function*** to post our message, ***postMessage()***, directly below the constructor:  
    ***postMessage() {  
    console.log("post");  
    }***
6. For this to work, we need to ***bind*** that function to our button. We will use a new AngularJS directive called ***ng-click*** to do this:  
    <span class="input-group-addon btn btn-primary"   
    ***ng-click="main.postMessage()"***>Post</span>  
   What is the ***main*** syntax signifying to make this connection? Look in ***main.controller.js***, and we will see that our **ui.router** gives us a new ***property*** called **controllerAs**, which we can use to shortcut, or ***alias***, our Controller for references to it. Turn on Developer Tools to the Console and let’s give this behavior a test.

**Exercise06\_01\_01 – Step 3**



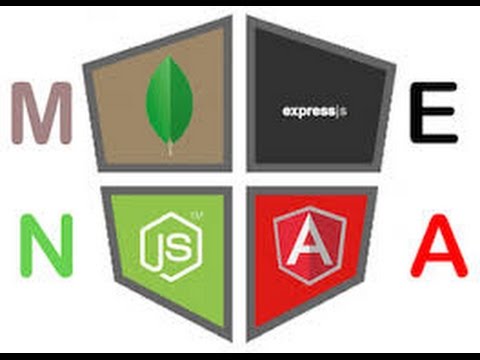
1. Build a new folder in the project folder named ***back-end*** and navigate to it in the IDE, and the terminal, if necessary. Let’s get it set up for ***Node.js*** by entering the following command and respond to the questions with the ***name*** being ***my-message-backend***, the ***author*** being yourself, and the rest as the ***default***:  
   ***npm init***Take a look at this in the IDE and open up the ***package.json*** file.
2. Now let’s install our local dependencies:  
   ***npm install express -save***Check the ***package.json*** file to make sure it has been installed and saved correctly.
3. Let’s start to build our server by creating a new file named ***server.js***. We will first ***require()*** in the express module, then ***instantiate*** the express app and store it in a variable named ***app***. Let’s also set a var to hold the ***port*** for the server to listen on, preferably 8080:  
   ***var express = require('express');  
   var app = express();  
   var port = 8080;***
4. We can now set our server up and running with the ExpressJS ***listen()*** method, passing it the ***port*** and a ***callback*** as follows:  
   ***var server = app.listen(port, () => {  
    console.log('Server listening on port %s', port);  
   });***Shut down gulp for now, a it will be blocking the server port. Now run the node.js server to test, then shut it down.
5. We will need an ***endpoint***, or ***route***, in our server to receive posted messages from the front-end. This could contain significant content, so we should use an ***HTTP*** ***POST*** for the endpoint. Build the endpoint as follows, with some ***debug*** code in its callback:  
   ***app.post('/api/message', (req, res) => {  
    console.log(req.body);  
    res.status(200);  
   });***
6. We will now try a way of testing our server endpoint without sending a message from our front-end, by downloading and using an application called ***Postman***. Set up a POST request to our endpoint as follows:  
   ***http://localhost:8080/api/message***
7. Select ***Body***, ***raw***, and select ***JSON*** from the dropdown. Enter the following JSON data in the Postman editing window:  
   ***{  
    "test": "works"  
   }***Make sure the server is running then hit ***SEND***. Check the node console, and we can see that a message came up showing ***undefined***.
8. In order to get content into the server endpoint for a POST, it must come in the request body, which we set up in Postman. However, we are going to need some middleware, ***body-parser***, in our server to make use of it, so we need to install it with npm first:  
   ***npm install body-parser –save***Check ***package.json*** for correctness.
9. Now we need to ***require()*** it in, then mount it as ***middleware*** with the ***app.use()*** method, setting it up for ***JSON***:  
   var port = 8080;  
   ***var bodyParser = require('body-parser');  
     
   app.use(bodyParser.json());***Run the server again, do a SEND again, and check the node console. Looks like we have response to our server endpoint.

**Exercise06\_01\_01 – Step 4**



1. Let’s switch back to the ***front-end*** in the IDE. We will implement sending a ***POST*** request from AngularJS. To do this we will make use of its ***$http*** service. Let’s do some work in ***main.controller.js***. First we must ***inject*** the $http service into the ***constructor***. We then can set a ***property*** of the controller object to the service:  
    constructor***($http***) {  
    'ngInject';  
    ***this.$http = $http;*** }
2. We can now modify our ***postMessage()*** function to make use of the ***property*** that is holding the service, and use it to issue the POST:  
    postMessage() {  
    ***this.$http.post('http://localhost:8080/api/message', {  
    msg: 'hello'  
    });*** }  
   Let’s give this a test. Make sure that ***gulp*** ***serve*** is not running, we must have our ***node*** server running first, so let’s start that up. Then run ***gulp*** serve, and notice on the URL bar that it comes up on port ***8081***. Turn on Developer Tools in the Console Tab, and do a Post. Notice that we get what is known as a Cross-Origin Error in the console, or ***CORS***, and nothing in our node console.
3. We can fix our ***CORS*** error with some ***CORES custom middleware*** in **server.js** that will set the proper ***headers*** to avoid it:  
   app.use(bodyParser.json());  
    ***app.use((req, res, next) => {  
    res.header('Access-Control-Allow-Origin', '\*');  
    res.header('Access-Control-Allow-Headers',   
    'Content-Type, Authorization');  
    next();  
   }));***Let’s restart our server, regenerate our browser, and give this another test. No browser Console error. Switch to the ***Network*** tab and click on the first ***message***. Look at its headers, and we can see our Access-Control instructions on the ***Response***. Click on the second message, and we can look at the ***Request Payload*** and it is our JSON. Check node, and our message has made it to the node console.
4. Let’s try to send an actual data from our ***<textarea>*** so open ***main.html***. We will make the ***<textarea>*** a model by adding an ***ng-model*** directive as follows:  
    <textarea class="form-control custom-control"   
    rows="3" style="resize: none"  
    ***ng-model="main.message">*** </textarea>  
   Shut down gulp for now, a it will be blocking the server port. Now run the node.js server to test, then shut it down.
5. Back in ***main.controller.js***, we can modify our JSON ***msg***:  
    this.$http.post('http://localhost:8080/api/message', {  
    msg: ***this.message*** });  
   Let’s check this out and we are now getting the actual message from our <textarea> visible in the node console.

**Exercise06\_01\_01 – Step 5**



1. For our database storage on the ***back-end***, we will use MongoDB. There are two ways we can implement this, by using the official Node.js MongoDB driver, or a package called Mongoose, which is built on top of it. We will start by using the official driver, so we will install it with npm:  
   ***npm install mongodb -save***
2. In ***server.js***, let’s require() in the driver, MongoClient, from the module:  
    postMessage() {  
   var bodyParser = require('body-parser');  
   ***var mongo = require('mongodb').MongoClient;***
3. Run a MongoDB server in a terminal. Let’s first set up a few variables to support connecting to MongoDB:  
   var mongo = require('mongodb').MongoClient;  
   ***var url = 'mongodb://localhost:27017';  
   var dbName = 'test';  
   var database;***
4. Now let’s build the code to ***connect*** to MongoDB, directly below our app.post() endpoint:  
   ***mongo.connect(url, (err, client) => {  
    if (err) {  
    return console.log('Error: ' + err);  
    }  
    database = client.db(dbName);  
    console.log('Connected to database: ' + dbName);  
   });***Run the server and give this a test to make sure we have a database connection.
5. Let’s try to ***insert*** a document into the database with a debug test as follows:  
    console.log('Connected to database: ' + dbName);  
    ***database.collection(messages).insertOne({'msg': 'test'});***Run RoboMongo and let’s check that our database ***test*** was created, that we got a collection named ***messages***, and that a document was inserted with the correct data. Let’s drop the database, and also get rid of our ***debug*** test code.
6. Now we will modify our ***app.post()*** endpoint to enter actual message posts into the database:  
   app.post('/api/message', (req, res) => {  
    console.log(req.body);  
    ***database.collection('messages').insertOne(req.body);***  
    res.status(200);  
   });  
   Go to RoboMongo and refresh. Let’s check that once again our database ***test*** was created, that we got a collection named ***messages***, and that a document was inserted with the data from our actual message post.