6 : Introduction to MEAN

IT2406 - Web Application Development 1

Level I - Semester 2

Chapter 6
Introduction to MEAN

Three-tier Web application Development

Most of the web applications are built in a three-tier architecture.

The three layers are,

Data layer

Logic layer

Presentation layer.

Most popular paradigm which implements this model it the MCV (Model, View and Control) architecture pattern.

Three-tier Web application Development

In MVC, data, logic and presentation are separated into three types of objects.

View take care of the user interactions and the visual part of the application.

Controller responds to system and user events and commanding the Model and View to change appropriately.

Model handles the data manipulation.

In modern web development, web application broken down into database, server logic, client logic and client UI.

Introduction to MEAN

The Mean stack is a full stack JavaScript solution which has four major components,

Database - MongoDB

Web server framework - Express

Web client framework - AnjularJS

Server platform - Node.js

MEAN is an abbreviation for MongoDB, Express, AngularJS and Node.js

Introduction to MEAN

One of the main advantages of MEAN is,

It is only use JavaScript driven solutions to cover the different parts of the application. Which means a single language is used throughout the application.

And the other advantage is it supports the MVC architecture.

Installing MongoDB

It can be installed in Linux, MacOS and Windows.

Most recent version can be downloaded from the official web site,

http://mongodb.org/downloads

Installing MongoDB on Windows

Download the latest version from the official web site.

Unpack the archive file.

Move it to c:\mongodb

Run the MongoDB

Use the command prompt and issue the following command

```
>c:\mongodb\bin\mongod.exe
```

Installing MongoDB on MacOS and Linux

Download the latest version from the official web site.

Unpack the archive file.

Create a default folder to store files by using following command.

```
$ mkdir -p /data/db
```

Run the MongoDB by using following commands

```
$ cd mongodb/bin

$ mongodb
```

Installing Node.js on Windows and MacOS

Download the latest version of Node.js from the official web site.

http://nodejs.org/download/

Use the installer to installed Node.js

Installing Node.js on Linux

Download the latest version of Node.js from the official web site.

http://nodejs.org/download/

Expand the .tar.gz file using the following command

```
$ tar -zxf node-v12.16.3.tar.gz

$ cd node-v12.16.3

$ ./configure

$ make

$ sudo make install
```

NPM

Node.js platform has less number of features and APIs.

NPM is used to install more complex functionalities and features to Node.js

NPM can be used to download download, install, update and remove packages and third party modules.

How to use NPM

To start Node.js on command line interface, type the following command

```
$ node
```

To install packages using NPM, using the following command

```
$ npm install <package name>
```

How to use NPM

To remove a package using NPM

```
$ npm uninstall <package name>
```

To update packages using NPM

```
$ npm update <package name>
```

Managing package dependancies

When an application is using multiple packages, there should be a proper method to manage these packages.

In NPM this is managed by the file called package,json

This file contains properties of your application such as the name, version and the application dependancies.

Generating the package.json file

Some application packages may include this package.json file. But sometimes you might need to manually generate the package.json file.

For that you can use the init command

```
$ npm init
```

NPM will ask you to provide some information about you application and it will generate the package.json file automatically.

Generating the package.json file

After generating the package.json file you can install the application dependancies by issuing the following command on application’s root directory.

```
$ npm install
```

Update the dependancies by

```
$ npm update
```

Node.js core modules

Core modules are modules that were compiled into the Node binary.

They come pre-bundled with Node and are documented in great detail in its documentation.

The core modules provide most of the basic functionalities of Node, including filesystem access, HTTP and HTTPS interfaces, and much more.

Node.js core modules

To load a core module, need to use the *require* method in your JavaScript file.

```
fs = require(‘fs');

fs.readFile('/etc/hosts', 'utf8', function (err, data) {

    if (err) {

        return console.log(err);

    }

    console.log(data);

});
```

More about Node's core modules : https://nodejs.org/api/

Node.js third-party modules

To use third-party modules, need to require them.

Node will first look for the module in the core modules folder and then try to load the module from the module folder inside the *node_modules* folder.

For using *express* third party module

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

var app = express();
```

Node.js file modules

Load modules which are in separate directories by providing the folder path.

Relative path

```
var bit = require('./modules/bit');
```

Absolute path

```
var bit = require(‘/home/projects/ucsc/modules/bit');
```

Developing Node.js web applications

Basic Node web server

In your working folder, create a file named server.js, which contains the following code,

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

http.createServer(function(req, res) {

    res.writeHead(200, {

        'Content-Type': 'text/plain'

    });

    res.end('Hello World');

}).listen(3000);

console.log('Server running at http://localhost:3000/');
```

Developing Node.js web applications

To start your web server, use your command-line tool, and navigate to your working folder.

Then, run the node CLI tool and run the server.js file as follows:

```
$ node server
```

Now open http://localhost:3000 in your browser, and you'll see the Hello World response.

Connect module

Connect is a module built to support interception of requests in a more modular approach.

It uses a modular component called middleware, which allows you to simply register your application logic to predefined HTTP request scenarios.

Connect middleware are basically callback functions, which get executed when an HTTP request occurs.

The middleware can then perform some logic, return a response, or call the next registered middleware.

Connect module

Creating a Connect application

In your working folder, create a file named server.js that contains the following code,

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

var app = connect();

app.listen(3000);

console.log('Server running at http://localhost:3000/');
```

Connect module

Application file is using the connect module to create a new web server.

Connect isn't a core module, so you'll have to install it using NPM.

```
$ npm install connect
```

To run your Connect web server, just use Node's CLI and execute the following command:

```
$ node server
```

Connect module

Reaching your application in the browser by visiting http:// localhost:3000.

It will give a message

Cannot GET /

This is because that there isn't any middleware registered to handle the GET HTTP request

Connect middleware

Connect middleware is just JavaScript function with a unique signature.

Each middleware function is defined with the following three arguments:

req: This is an object that holds the HTTP request information

res: This is an object that holds the HTTP response information and allows you to set the response properties

next: This is the next middleware function defined in the ordered set of Connect middleware

Connect middleware

Register the middleware with the Connect application using the app.use() method. Change the server.js file

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

var app = connect();

var helloWorld = function(req, res, next) {

    res.setHeader('Content-Type', 'text/plain');

    res.end('Hello World');

};

app.use(helloWorld);

app.listen(3000);

console.log('Server running at http://localhost:3000/');
```

Connect middleware

Start the connect server

```
$ node server
```

Visit http://localhost:3000

Now it should display Hello World, in the browser window.

Order of Connect middleware

able to set a series of middleware functions that will be executed in a row to achieve maximum lexibility when writing your application.

In each middleware function, you can decide whether to call the next middleware function or stop at the current one

Notice that each Connect middleware function will be executed in irst-in-irst-out (FIFO) order using the next arguments until there are no more middleware functions to execute or the next middleware function is not called.

Order of Connect middleware

Update server.js file

```
var connect = require('connect');
var app = connect();
var logger = function(req, res, next) { // new middleware inserted. This will execute before helloWorld middleware
    console.log(req.method, req.url);
    next(); //this is responsible for calling helloWorld middle. If next() is not available, the request will hang forever
};
var helloWorld = function(req, res, next) {
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World');
};
app.use(logger);
app.use(helloWorld); app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Mounting Connect middleware

Connect middleware supports a feature called mounting, which enables to determine which request path is required for the middleware function to get executed.

Mounting is done by adding the path argument to the app. use() method.

server.js file

```
var connect = require('connect');
var app = connect();
var logger = function(req, res, next) {
   console.log(req.method, req.url);
   next();
};
var helloWorld = function(req, res, next) {
   res.setHeader('Content-Type', 'text/plain');
   res.end('Hello World');
};
var goodbyeWorld = function(req, res, next) {
   res.setHeader('Content-Type', 'text/plain');
   res.end('Goodbye World');
};
app.use(logger);
app.use('/hello', helloWorld); //hello middleware will respond only to requests made for /hello path.
app.use('/goodbye', goodbyeWorld); //goodbyeWorls middleware will respond only to requests made for /goodbye path.

app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Building an Express Web Application

Installing Express

Method 1:

```
$ npm install express
```

Method 2:

Creating a new working folder. Inside it create a new package.json file including the following code.

(name and the version are details about your application)

```
{
   "name" : "BIT",
   "version" : "0.1",
   "dependencies" : {
      "express" : "~4.8.8"
   }
}
```

Then issues the following command. This will install the dependancies.

```
$ npm install
```

Building an Express Web Application

Create the server.js file as follows, and run the application using *$ node server* command. Visiting http://localhost:3000 in the browser.

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

app.use('/', function(req, res) { //app.use() is used to mount the middleware function.
    res.send('Hello World');  //res.send() is used to send the response back
});

app.listen(3000);

console.log('Server running at http://localhost:3000/');
module.exports = app;
```

Application Object

Application object helps to configure the application. Following are some of the methods,

app.set(name, value): This is used to set environment variables that Express will use in its configuration.

app.get(name): This is used to get environment variables that Express is using in its configuration.

app.engine(ext, callback): This is used to define a given template engine to render certain file types, for example, you can tell the EJS template engine to use HTML ifles as templates like this: app.engine('html', require('ejs').renderFile).

app.use([path], callback): This is used to create an Express middleware to handle HTTP requests sent to the server. Optionally, you'll be able to mount middleware to respond to certain paths.

Application Object

Application object helps to configure the application. Following are some of the methods,

app.VERB(path, [callback...], callback): This is used to define one or more middleware functions to respond to HTTP requests made to a certain path in conjunction with the HTTP verb declared. For instance, when you want to respond to requests that are using the GET verb, then you can just assign the middleware using the app.get() method. For POST requests you'll use app.post(), and so on.

app.route(path).VERB([callback...], callback): This is used to define one or more middleware functions to respond to HTTP requests made to a certain unified path in conjunction with multiple HTTP verbs. For instance, when you want to respond to requests that are using the GET and POST verbs, you can just assign the appropriate middleware functions using app. route(path).get(callback).post(callback).

Request Object

Request object provide helping methods that contain information about the HTTP request. Following are some of the methods,

req.query: This is an object containing the parsed query-string parameters.

req.params: This is an object containing the parsed routing parameters.

req.body: This is an object used to retrieve the parsed request body. This property is included in the bodyParser() middleware.

req.param(name): This is used to retrieve a value of a request parameter. Note that the parameter can be a query-string parameter, a routing parameter, or a property from a JSON request body.

req.path, req.host, and req.ip: These are used to retrieve the current request path, host name, and remote IP.

req.cookies: This is used in conjunction with the cookieParser() middleware to retrieve the cookies sent by the user-agent.

Response Object

Requests send to the server are handled by using the response object methods,

res.status(code): This is used to set the response HTTP status code.

res.set(field, [value]): This is used to set the response HTTP header.

res.cookie(name, value, [options]): This is used to set a response cookie. The options argument is used to pass an object defining common cookie configuration, such as the maxAge property.

res.redirect([status], url): This is used to redirect the request to a given URL. Note that you can add an HTTP status code to the response. When not passing a status code, it will be defaulted to 302 Found.

res.send([body|status], [body]): This is used for non-streaming responses. This method does a lot of background work, such as setting the Content-Type and Content-Length headers, and responding with the proper cache headers.

res.json([status|body], [body]): This is identical to the res.send() method when sending an object or array. Most of the times, it is used as syntactic sugar, but sometimes you may need to use it to force a JSON response to non-objects, such as null or undefined.

res.render(view, [locals], callback): This is used to render a view and send an HTML response.

MVC pattern : Application folder structure

Horizontal folder structure

A horizontal project structure is based on the division of folders and files by their functional role rather than by the feature they implement, which means that all the application files are placed inside a main application folder that contains an MVC folder structure.

As an example; single controllers folder that contains all of the application controllers, a single models folder that contains all of the application models

MVC pattern : Application folder structure

Vertical folder structure

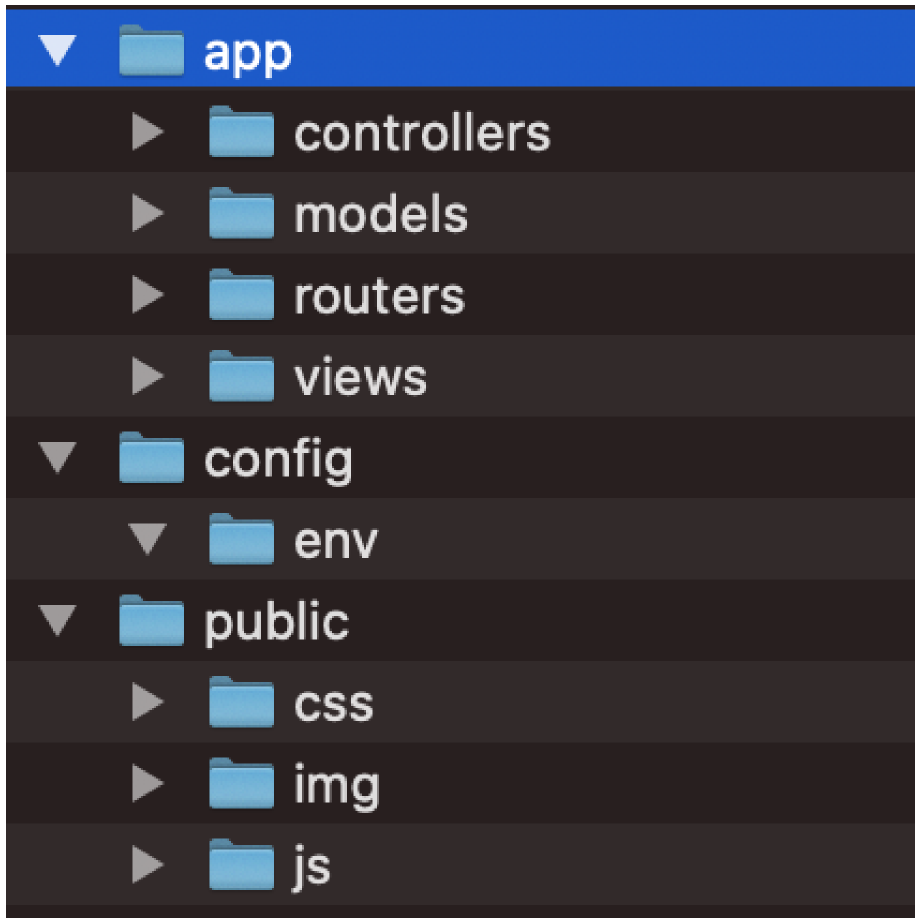
A vertical project structure is based on the division of folders and files by the feature they implement, which means each feature has its own autonomous folder that contains an MVC folder structure.

As an example; each feature has its own application-like folder structure.

MVC pattern : Application folder structure

Implementing horizontal file structure.

Create the folder structure as follows.



```
▼ app
    ▶ controllers
    ▶ models
    ▶ routers
    ▶ views
▼ config
    ▼ env
▼ public
    ▶ css
    ▶ img
    ▶ js
```

MVC pattern : Application folder structure

Create package.json file on root folder as follows.

```
{
"name" : “Project_name”,
"version" : “0.05",
   "dependencies" : {
      "express" : "~4.8.8"
   }
}
```

MVC pattern : Application folder structure

Inside app/controllers folder, create the file named as index.server.controller.js with the following code,

```
exports.render = function(req, res) {
    res.send('Hello World');
};
```

Handling request routing

Express supports the routing of requests using either the app.route(path). VERB(callback) method or the app.VERB(path, callback) method.

The VERB should be replaced with a lowercase HTTP verb.

```
app.get('/', function(req, res) {
    res.send('This is a GET request');
});
```

Handling request routing

Chain several middleware in a single routing definition.

```
var express = require('express');
var hasName = function(req, res, next) { //Middleware function
   if (req.param('name')) {
      next();
    } else {
      res.send('What is your name?');
   }
};
var sayHello = function(req, res, next) { //Middleware function
   res.send('Hello ' + req.param('name'));
};
var app = express();
app.get('/', hasName, sayHello); //Adding middleware functions in a row by using app.get function
                                              app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Adding routing file

In the app/routes folder, create a file named index.server.routes.js with the following code.

```
module.exports = function(app) {

    var index = require('../controllers/index.server.controller');

    app.get('/', index.render);

};
```

With this required the index controller and used its render() method as a middleware to GET requests made to the root path.

Adding routing file

Create the Express application object and bootstrap it using the controller and routing modules which has created.

```
var express = require('express'); //create new instance of Express application

module.exports = function() {//requires the routing file and calls it as a function passing it the application instance as an argument.

    var app = express();

    require(‘../app/routes/index.server.routes.js')(app);

    return app;

};
```

Adding routing file

Finally create a file named server.js in the root folder and add the following code.

```
var express = require('./config/express');

var app = express();

app.listen(3000); module.exports = app;


console.log('Server running at http://localhost:3000/');
```

Adding routing file

To start the application, navigate to the root folder of your application and issue the following commands.

```
$ npm install
```

After completing the installation, issue the following command.

```
$ node server
```

Goto the following link from the web browser

http://localhost:3000

Configuring an Express application

Download and install middleware as your project dependencies, edit the package.json file.

```
{
  "name": "BIT",
  "version": "0.2",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    “compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0"
  }
}
```

Configuring an Express application

The morgan module provides a simple logger middleware.

The compression module will provides response compression.

The body-parser module provides several middleware to handle request data.

The method-override module provides DELETE and PUT HTTP verbs legacy support.

Configuring an Express application

To use these modules, need to modify the config/express.js

```
var express = require('express'),
   morgan = require('morgan'),
   compress = require('compression'),
   bodyParser = require('body-parser'),
   methodOverride = require('method-override');

module.exports = function() {
   var app = express();
   if (process.env.NODE_ENV === 'development') { //determine the environment and configure the Express application

      app.use(morgan(‘dev')
   );
   } else if (process.env.NODE_ENV === 'production') {
   app.use(compress());
   }
   app.use(bodyParser.urlencoded({
   extended: true
   }));
   app.use(bodyParser.json());
   app.use(methodOverride());
   require('../app/routes/index.server.routes.js')(app);
    return app;
  };
```

Configuring an Express application

Finalize the configuration, edit the server.js file

```
process.env.NODE_ENV = process.env.NODE_ENV || 'development';


var express = require('./config/express');

var app = express();

app.listen(3000);

module.exports = app;


console.log('Server running at http://localhost:3000/');
```

Run the application by issuing following commands

```
$ npm install

$ node server
```

Navigate to http:// localhost:3000

Rendering Views

Configuring view system.

Install EJS module. Change the package.json file as follows.

```
{
  "name": "MEAN",
  "version": "0.0.3",
  "dependencies": {
    "express": "~4.8.8",
    "morgan": "~1.3.0",
    "compression": "~1.0.11",
    "body-parser": "~1.8.0",
    "method-override": "~2.2.0",
    "ejs": "~1.0.0"
  }
}
```

Run the following command

```
$ npm update
```

Rendering Views

Configure Express to use EJS as the default template engine. Change the config/express.js file.

```
var express = require('express'),
    morgan = require('morgan'),
    compress = require('compression'),
    bodyParser = require('body-parser'),
    methodOverride = require('method-override');


module.exports = function() {
    var app = express();
    if (process.env.NODE_ENV === 'development') {
        app.use(morgan('dev'));
    } else if (process.env.NODE_ENV === 'production') {
        app.use(compress());
    }
    app.use(bodyParser.urlencoded({
        extended: true
    }));
    app.use(bodyParser.json());
    app.use(methodOverride());
    app.set('views', './app/views');
    app.set('view engine', 'ejs');
    require('../app/routes/index.server.routes.js')(app);
     return app;
   };
```

Rendering EJS views

EJS views basically consist of HTML code mixed with EJS tags.

EJS templates will reside in the app/views folder and will have the .ejs extension.

It use the res.render() method, the EJS engine will look for the template in the views folder, and if it finds a complying template, it will render the HTML output.

Rendering EJS views

Go to app/views folder, and create a new file named index.ejs with the following code.

```
<!DOCTYPE html>

   <html>

<head>

<title><%= title %></title>

     </head>

     <body>

<h1><%= title %></h1> </body>

</html>
```

Rendering EJS views

<%= %> tag : tell the EJS template engine where to render the template variables

Which means the *title* is a variable

Change the app/controllers/index.server.controller.js file.

```
exports.render = function(req, res) {
   res.render('index', {
      title: 'Hello World'
    })
};
```

Run the following command and visit the application.

```
$ node server
```

Visit - http://localhost:3000

Serving static files

In any web application, there is always a need to serve static files.

Express comes prebundled with the express.static() middleware, which provides this feature

Changes the config/express.js file

```
var express = require('express'),
   morgan = require('morgan'),
   compress = require('compression'),
   bodyParser = require('body-parser'),
   methodOverride = require('method-override');
module.exports = function() {
   var app = express();
   if (process.env.NODE_ENV === 'development') {
      app.use(morgan('dev'));
   } else if (process.env.NODE_ENV === 'production') {
      app.use(compress());
   }
app.use(bodyParser.urlencoded({
   extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
app.set('views', './app/views');
app.set('view engine', 'ejs');
require('../app/routes/index.server.routes.js')(app);
app.use(express.static('./public')); //Determine the location of the static folder
    return app;
};
```

Change the app/views/index.ejs as follows,

```
<!DOCTYPE html>

   <html>

<head>

<title><%= title %></title>

     </head>

     <body>

<img src="img/logo.png" alt="Logo">

<h1><%= title %></h1> </body>

</html>
```

Run the application using $ node server command and visit

http://localhost:3000

Configuring sessions

Install and configure the express-session middleware.

Modifying your package.json file.

```
{
   "name": "MEAN",
   "version": "0.0.3", "dependencies": {
      "express": "~4.8.8",
      "morgan": "~1.3.0",
      "compression": "~1.0.11",
      "body-parser": "~1.8.0",
      "method-override": "~2.2.0",
      "express-session": "~1.7.6",
      "ejs": "~1.0.0"
   }
}
```

Use the command $ npm update

Configuring sessions

Adding the sessionSecret property. Change the config/env/development.js file as follows,

```
module.exports = {
   sessionSecret: ‘put_any_secret_string_here’
};
```

Change the config/express.js file

```
var config = require('./config'),
   express = require('express'),
   morgan = require('morgan'),
   compress = require('compression'),
   bodyParser = require('body-parser'),
   methodOverride = require('method-override'),
   session = require('express-session');
module.exports = function() {
   var app = express();
   if (process.env.NODE_ENV === 'development') {
      app.use(morgan('dev'));
   } else if (process.env.NODE_ENV === 'production') {
      app.use(compress());
   }
   app.use(bodyParser.urlencoded({
      extended: true
   }));
```

```
   app.use(bodyParser.json());
   app.use(methodOverride());
   app.use(session({
      saveUninitialized: true,
      resave: true,
      secret: config.sessionSecret
   }));
   app.set('views', './app/views');
   app.set('view engine', 'ejs');
   require('../app/routes/index.server.routes.js')(app);
app.use(express.static('./public'));
 return app;
};
```

Configuring sessions

To test the session, change the app/controller/index.server.controller.js file

```
exports.render = function(req, res) {
    if (req.session.lastVisit) {
        console.log(req.session.lastVisit);
    }
    req.session.lastVisit = new Date();
    res.render('index', { title: 'Hello World'
    });
};
```

Run the command $ node server and visit http://localhost:3000

MongoDB

MongoDB database

List all the other databases in the current MongoDB server

```
> show dbs
```

The MongoDB shell will automatically connect to the default test database. Let's switch to another database called ucsc by executing the following command.

```
> use ucsc
```

MongoDB CRUD operations

Creating a new document

Using insert()

```
> db.posts.insert({"title":"Second Post", "user": "alice"})
```

Using update()

```
> db.posts.update({
“user": "alice"
}, {
  "title": "Second Post",
  "user": "alice"
}, {
  upsert: true
})
```

MongoDB CRUD operations

Creating a new document

Using save()

```
> db.posts.save({"title":"Second Post", "user": "alice"})
```

Reading documents using find()

```
> db.posts.find({})
```

This will return all the documents in the posts collection.

MongoDB CRUD operations

Reading documents using equality statement

```
> db.posts.find({ "user": "alice" })
```

This will retrieve all the documents that have the user property equal to alice.

MongoDB CRUD operations

Update document using update()

```
> db.posts.update({
   "user": "alice"
}, {
   $set: {
         "title": "Second Post"
   }
}, {
   multi: true
})
```

First argument tell MongoDB to look for all the documents created by alice.

Second argument tell to updates the *title* field.

Third argument force to execute the update operation on all the documents it find.

MongoDB CRUD operations

Deleting documents

Deleting all documents

```
> db.posts.remove()
```

Deleting multiple documents

Remove multiple documents that match a criteria from a collection.

```
> db.posts.remove({ "user": "alice" })
```

Remove all the posts made by alice

Mongoose

After installing MongoDB, connect it using Mongoose.

Change the package.json file.

```
{
 "name": "UCSC",
 "version": “0.5",
   "dependencies": {
     "express": "~4.8.8",
     "morgan": "~1.3.0",
     "compression": "~1.0.11",
     "body-parser": "~1.8.0",
     "method-override": "~2.2.0",
     "express-session": "~1.7.6",
     "ejs": "~1.0.0",
     "mongoose": “~3.8.15" //Add this line to package.json
 }
}
```

Issue the command

```
$ npm install
```

Mongoose

Let’s learn how to define a user schema and model, and how to use a model instance to create, retrieve, and update user documents.

Create the schema.

In the app/models create a new file called, user.server.model.js.

Inside user.server.model.js, add the following code.

```
var mongoose = require('mongoose'),
Schema = mongoose.Schema;

var UserSchema = new Schema({
   firstName: String,
   lastName: String,
   email: String,
   username: String,
   password: String
});

mongoose.model('User', UserSchema);
```

CRUD

Registering the User model

To register User model, include the user. server.model.js file in Mongoose configuration file.

Change the config/mongoose.js file.

```
var config = require('./config'),
    mongoose = require('mongoose');

module.exports = function() {
    var db = mongoose.connect(config.db);
    require('../app/models/user.server.model');
    return db;
};
```

CRUD

Creating new users using save()

Create a user controller in app/controllers folder as users.server. controller.js and add the following code.

```
var User = require('mongoose').model('User');

exports.create = function(req, res, next) {
   var user = new User(req.body);
   user.save(function(err) {
      if (err) {
            return next(err);
      } else {
         res.json(user);
      }
   });
};
```

CRUD

Add user-related routes, creating a file named users.server.routes.js inside the app/routes folder and add the following code.

```
var users = require('../../app/controllers/users.server.controller');
   module.exports = function(app) {
      app.route('/users').post(users.create);
};
```

Change the config/express.js file

```
var config = require('./config'),
   express = require('express'),
   morgan = require('morgan'),
   compress = require('compression'),
   bodyParser = require('body-parser'),
   methodOverride = require('method-override'),
   session = require('express-session');

module.exports = function() {
   var app = express();
   if (process.env.NODE_ENV === 'development') {
      app.use(morgan('dev'));
   } else if (process.env.NODE_ENV === 'production') {
      app.use(compress());
   }

app.use(bodyParser.urlencoded({
   extended: true
}));
app.use(bodyParser.json());
app.use(methodOverride());
```

```
   app.use(session({
      saveUninitialized: true,
      resave: true,
      secret: config.sessionSecret
   }));

   app.set('views', './app/views');
   app.set('view engine', 'ejs');
   require('../app/routes/index.server.routes.js')(app);
   require('../app/routes/users.server.routes.js')(app);
   app.use(express.static('./public'));
      return app;
};
```

CRUD

To run the app issue the following code.

```
$ node server
```

When executing the app make sure the request body includes the following JSON:

```
{
    "firstName": "First",
    "lastName": "Last",
    "email": "user@example.com",
    "username": "username",
    "password": "password"
}
```

Angular Application

Angular is a platform and framework for building single-page client applications using HTML and TypeScript.

Angular is written in TypeScript.

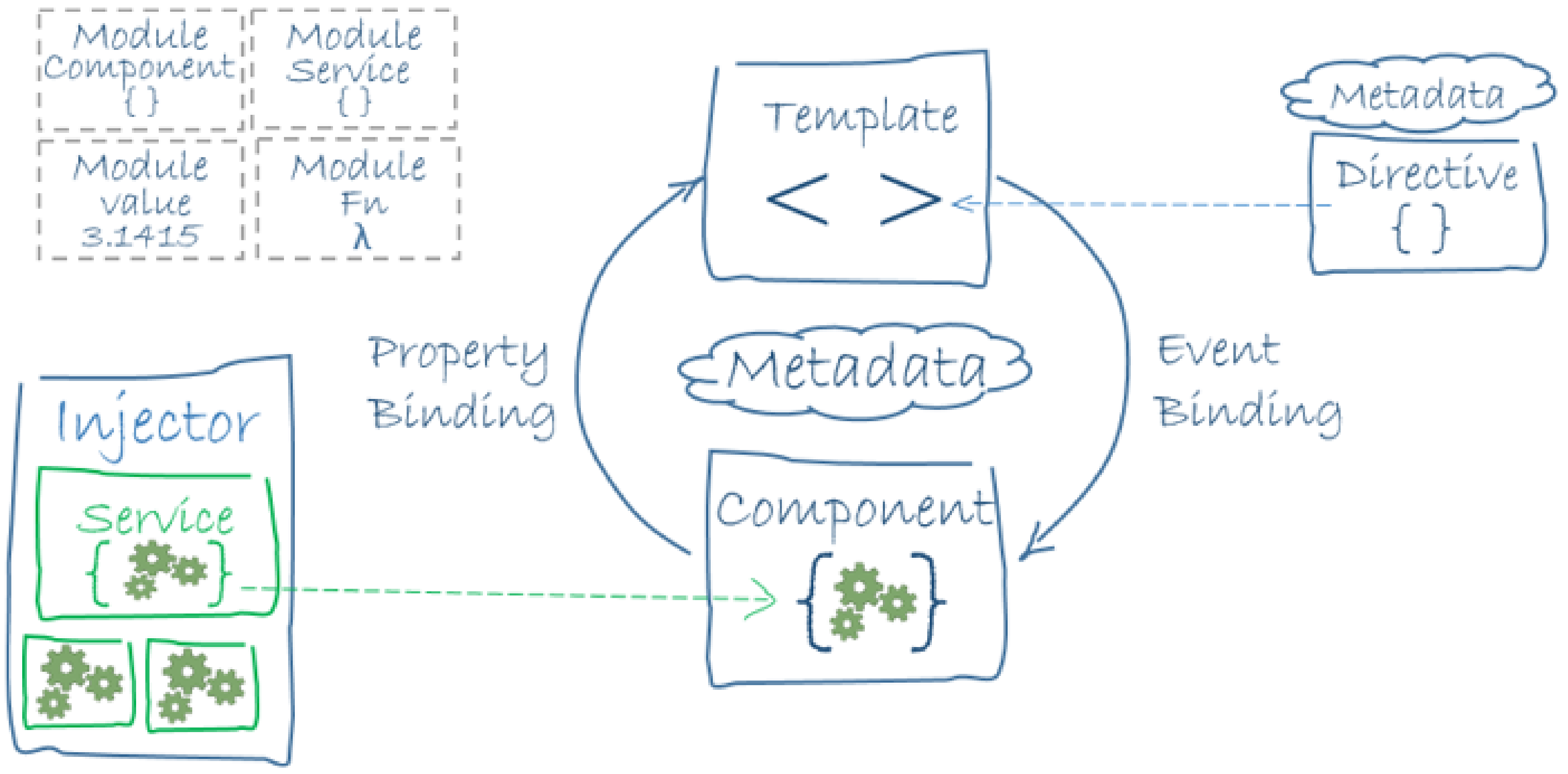
The architecture of an Angular application relies on certain fundamental concepts.

The basic building blocks are NgModules, which provide a compilation context for components.

https://angular.io/docs

Angular Application

NgModules, Components, Views, and Services are the basic building blocks of an angular application.



https://angular.io/guide/architecture

Angular Application

Module

Angular apps are modular and Angular has its own modularity system called NgModules.

NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

Every Angular app has at least one NgModule class, the root module, which is conventionally named AppModule and resides in a file named app.module.ts.

Angular Application

Simple root NgModule definition (src/app/app.module.ts)

```
import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

@NgModule({

  imports:      [ BrowserModule ],

  providers:    [ Logger ],

  declarations: [ AppComponent ],

  exports:      [ AppComponent ],

  bootstrap:    [ AppComponent ]

})

export class AppModule { }
```

Angular Application

declarations: The components, directives, and pipes that belong to this NgModule.

exports: The subset of declarations that should be visible and usable in the component templates of other NgModules.

imports: Other modules whose exported classes are needed by component templates declared in this NgModule.

providers: Creators of services that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)

bootstrap: The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the bootstrap property.

Angular Application

Component

A Component is a functional code block as a typescript class decorated with @Component(…) decorator having some metadata which defines a template.

Can add a component by using the following command

```
ng g c {ComponentName}
```

Angular Application

Simple component

```
import { Component } from '@angular/core';

@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.scss']
})
export class AppComponent {
   title = 'MyApp;
   backgroundColor = "blue";
   model: any = { username:"MyName", firstName:"FirstName", lastName:"LastName" }

   Submit() {
         console.log('Successful :', this.model);
   }
}
```

Angular Application

Template

These are the UI elements of the application.

which are defined by the Components.

It allows to including external HTML into the currently loaded page in browser.

Angular Application

Service

Services are used to encapsulate functionality that reused in an application.

Built-in services exist in AngularJS framework such as $http, $resource, $location, and $window.

https://docs.angularjs.org/api/ng/service

Angular Application

Routing

Routing is mechanism of mapping different URL request to different views of web application.

When user enter a specific url into the web browser then $routeProvider load that specific web page into the web browser and also load specific controller for that web page.

```
$routeProvider.when('/bitinfo', {
            templateUrl: 'bitinfo.html',
            controller:  'bitInfoController'
});
```

Angular Material UI

AngularJS Material is an implementation of Google's Material Design Specification

This project provides a set of reusable, well-tested, and accessible UI components for AngularJS developers.

https://material.angularjs.org/latest/

Angular Material UI

Install Material,

To install latest release

```
npm install angular-material
```

To install latest release and update package.json

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

Angular Material UI

Using the AngularJS Material Library

NPM install the files under /node_modules/angular-material/

```
<head>
  <meta name="viewport" content="initial-scale=1, maximum-scale=1, user-scalable=no" />
  <link rel="stylesheet" href="/node_modules/angular-material/angular-material.css">
</head>
```

```
<script src="/node_modules/angular/angular.js"></script>
<script src=“/node_modules/angular-aria/angular-aria.js"></script>
<script src=“/node_modules/angular-animate/angular-animate.js"></script>
<script src=“/node_modules/angular-messages/angular-messages.js"></script>
<script src=“/node_modules/angular-material/angular-material.js"></script>
```

Report generation

There are number of 3rd party packages available for report generation using node.js.

One of them is PDFKit. (https://pdfkit.org/)

Installing PDFKit

```
npm install pdfkit
```

Email with node.js

Nodemailer module can be used to generate emails using node.js

Install Nodemailer

```
npm install nodemailer
```

Include module in the application

https://nodemailer.com/about/

```
var nodemailer = require(‘nodemailer');
```

References

http://www.webdevelopmenthelp.net/2017/01/learn-mean-stack-development.html, Web Development Tutorial, 12 November 2019, 27 November 2019, MEAN Stack Tutorial

Haviv, A. Q. (2016) MEAN Web Development.