Getting MEAN

From CSE330 Wiki

Contents

Installation

To install MEAN, make sure you already have all of the following components installed and configured correctly.
1. Node.js (should come with npm)
To find out if you have node and npm, try checking their versions (may not be the same):

```
node --version
v0.10.25
npm --version
1.3.24
```

IF you don't have node.js, install it here (http://nodejs.org/) .

2. Bower

Bower is a web application manager used by MEAN. To install bower, type the following command in your terminal:

```
npm install -g bower
```

3. MongoDB To install MongoDB, follow the instructions for your OS here (http://docs.mongodb.org/manual/installation/) .

4. Grunt To install Grunt type the following command:

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

5. and finally, MEAN In your terminal, type:

```
> sudo npm install -g mean-cli
```

- MEAN stack documentations are available here (http://mean.io/#!/docs) . They'll help you to get familiar with the file structure and idioms.

What did all that mean?

You just installed four different command line tools. Here's what they all do.

- **npm** is the package manager for Node.JS. It installs libraries for use by your Node.JS applications. The configuration file for *npm* (the list of packages for *npm* to install) is *package.json*.
- **bower** is a package manager for front-end libraries. Rather than manually downloading your jQuery, Bootstrap, Angular, and other front-end JavaScript and CSS (and remembering to update them when a new version of one of those libraries comes out), *bower* installs them for you. The configuration file for *bower* (the list of packages for *bower* to install) is *bower.json*.
- **grunt** is a build tool designed for JavaScript. It's like *make* or *ant*, but for JavaScript. The configuration file for *grunt* (the list of commands to run when building your project) is *Gruntfile.js* (a name inspired by the *Makefile* for *make*, the dominant C and C++ build tool).
- **mean** is a utility designed to help you carry out routine tasks associated with developing for the MEAN stack.

Don't be confused by the fact that you use *npm* to install *bower*, *grunt*, and *mean*. The people who made *bower*, *grunt*, and *mean* just all decided to use *npm* as the place to publish their projects, probably because the executables themselves are written in Node.JS. They could just as well have chosen *apt-get* or *yum* or any of the other package managers you've been using in this class.

Running the Default App

- Init and run an app.

```
> mean init myApp // create your first app
> cd myApp
> npm install // Install dependencies
> grunt // Launch mean
```

Make sure MongoDB is connected and running (if you run into problems, refer to the section on MongoDB)

- Go ahead and explore features of the default Article app (such as logging in, adding/deleting articles).

Generating a Package

- By now you should know what a package is in MEAN. If you don't, read this (http://learn.mean.io/#mean-io-packages) .
- Try creating a package named myPackage by typing:

```
> mean package myPackage
```

Locate it inside the packages folder:

- Compare the structure of myPackage folder to that of the default app. Make sure you see both **public** and **server** folders and their subfolders.
- Restart your app, make sure you can see a link to the newly created package.

A Closer Look at the App

Take a look at the directory where you app resides, you will see two folders **public** and **server**, which contain code for the client side and server side respectively.

Models

Open up article.js in /server/models

MEAN is MVC on the client side and the server side. Since the model is tied to the backend to MongoDB, you will only see the models folder on the server side, which is normal. The models are mongoose models that's why we need to require

mongoose (what the following line does):

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

The following part then defines the schema:

```
var ArticleSchema = new Schema({
    created: {
        type: Date,
        default: Date.now
    },
    title: {
        type: String,
        default: '',
        trim: true
    },
    content: {
        type: String,
        default: '',
        trim: true
    },
    user: {
        type: Schema.ObjectId,
        ref: 'User'
    }
});
```

The next part is validations, or criteria an object needs to pass before it can be saved to the DB. The following code is to make sure that an article has a title:

```
ArticleSchema.path('title').validate(function(title) {
    return title.length;
}, 'Title cannot be blank');
```

Finally, Mongoose allows us to define methods to our data objects as well. These are called statics. From the Mongoose doc: Schemas not only define the structure of your document and casting of properties, they also define document instance methods, static Model methods, compound indexes and document lifecycle hooks called middleware. The following code defines a method called load for the article schema:

```
ArticleSchema.statics.load = function(id, cb) {
    this.findOne({
        _id: id
    }).populate('user', 'name username').exec(cb);
};
```

If you have trouble understanding what the code does, review the Mongoose quick start (http://mongoosejs.com/docs/index.html) guide.

Models Practice

Define a schema for a model named **Phone**, name it phone.js and save it in the corresponding folder in 'myPackage' (hint: /packages/myPackage/server/models/) A product needs to have the following fields with specific types. Here is an example of a Phone object can be stored in the MongoDB:

```
    {
        "age": 0,
        "id": "motorola-xoom-with-wi-fi",
```

```
        "imageUrl": "img/phones/motorola-xoom-with-wi-fi.0.jpg",
        "name": "Motorola XOOM\u2122 with Wi-Fi",
        "snippet": "The Next, Next Generation\r\n\r\nExperience the future
with Motorola XOOM wi.. d by Android 3.0 (Honeycomb)."
    }
```

This is taken from phones.json used by the AngularJS tutorial. If you have gone through the tutorial, you should have realized that instead of using a DB, json files were used to simplify the tutorial. Find phones.json in /app/phones in the angular tutorial folder and try importing its content to MongoDB so you can use them in your current MEAN package. Here is how (assuming mongod is running):

```
> mongo --shell // connect to mongo shell
> use mean-dev // switches to database used by mean.
> mongoimport --db mean-dev --collection phones --type json --file ''<path to
phones.json>'' --jsonArray // imports json file to the DB
> show collections // check to see if phones are added to your database
> coll = db.phones // to see the content of the file
```

Controllers

You'll find a controllers folder on both the server side and the client (public) side.

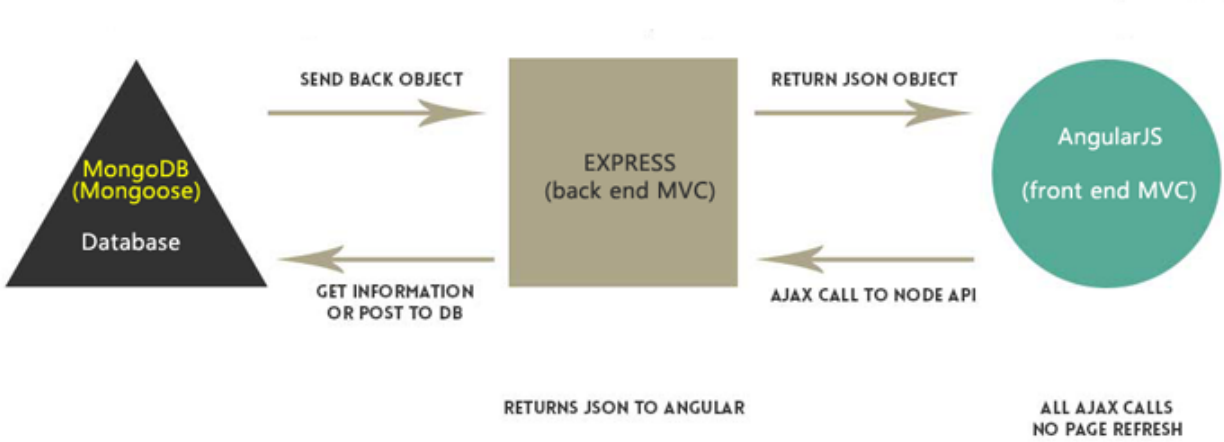
- The client side controllers are part of the Angular MVC scheme that contains client side logic and may issue requests to the server.
- The server side controllers deal with the models and requests from the client side.

Server side:

- Find articles.js in the controllers folder. Try to understand what each part does.

Client side: open up articles.js in /articles/public/controllers Compare the functions defined in ArticlesController, can you find the server side implementation of corresponding functions?

Here is a diagram to help you understand how this works:



Views & Routing

Even though typically views are implemented and used for the client end, routing happens on both ends.

- Find articles.js in /server/routes

You'll see that it requires the articles controller and its methods for querying the database.

The first section requires the articles controller, which defines different methods

For example, code below tells the server to use a method named "all" in the articles controller when the url is pointed to /articles:

```
app.route('/articles')
      .get(articles.all)
```

- Go to http://localhost:3000/articles, what do you see? Does it make sense?

Let's switch to the public folder where the views reside. Go to articles/public/views, open up list.html:
The first line tells the framework that ArticlesController is responsible for this particular view and when this view is loaded, use find() to initiate data binding.

```
<section data-ng-controller="ArticlesController" data-ng-init="find()">
```

Feel free to take a look at what find() does. Make sure you understand that what find() does is actually done by the backend, where a corresponding find() is written (as mentioned above). Hopefully you have a better understanding of how these technologies all work together now.

Review the AngularJS section if you have any questions regarding the view data binding.

The view files in articles/views only defines the code for displaying the articles, where does the navigation bar gets defined. The answer is in /public/system.

- The .html files in public/system/views provide a container/layout for the application.

For example, header.html includes the menu bar (displayed on the top of the page):

```
<div class="left pull-left">
     <ul class="navbar-nav nav">
         <li data-ng-repeat="item in menus.main" ui-route="/{{item.link}}" ng-
class="{active: $uiRoute}">
             <a mean-token="item.link" ui-sref='{{item.link}}'>{{item.title}}
</a>
        </li>
   </ul>
</div>
```

- Try changing {{item.title}} inside the <a> tags to "my articles" see what happens to the page.
- The ui-sref is "href" in ui-route, an AngularJS plugin, try changing {{item.link}} to #!/myarticles,
- What happens to the application now? When you click on "my articles"?

The page to articles is no longer served!

But how does the framework know which page to serve given a certain url? The file responsible for this (called routing in web development) is in public/articles/routes

open up articles.js in this folder and find the line that specifies the routing scheme for /articles:

```
            .state('all articles', {
                url: '/articles',
                templateUrl: 'public/articles/views/list.html',
                resolve: {
                    loggedin: checkLoggedin
                }
            }
```

This is ui-route's way of telling the server what to serve the page specified by the templateUrl when it sees "localhost:3000/#!/phones". The #! is a hash prefix used for MEAN to make ajax calls so make sure to include it when coding your views. Now try changing the url to "/myarticles", save and go back to your app, click on "my articles" now, what happens?

- This is important if you want to customize your navigation bar and links. Just remember that each url needs an entry in the route file, in the format of the example above.

Practice

Can you implement a view in your myPackage to display all phones injected in the DB? Hint: myPackage has the same file structure as the default app.