Introduction

The goal of this tutorial is to guide you through the creation of a Reddit/Hacker News clone using the MEAN stack. By completing this tutorial, you will gain a basic understanding of the MEAN stack including building a REST interface with Express.js on top of Node.js and using that interface to perform CRUD operations on a database via an AngularJS frontend.

Why MEAN Stack?

The acronym "MEAN" stands for "MongoDB Express.js AngularJS Node.js" and represents a group of technologies which are known to synergize well together. The major benefit of the MEAN stack is that it's extremely quick to prototype with. Node.js allows you to use Javascript on the backend as well as the frontend which can save you from having to learn a separate language. In addition, the NoSQL nature of MongoDB allows you to quickly change and alter the data layer without having to worry about migrations, which is a very valuable attribute when you're trying to build a product without clear specifications. Finally, these technologies have a lot of community support behind them so finding answers to questions or hiring help is going to be much easier using these technologies.

Prerequisites

This course assumes knowledge of programming and at least basic knowledge of JavaScript. In addition, you should be comfortable with basic web application concepts including REST and CRUD. Before you begin, you will also need to have Node.js and MongoDB already installed. Because you will need to install various packages for Node.js, you should follow these installation instructions which use npm. Installation instructions for MongoDB can be found on the Mongo website. This tutorial will be based on AngularJS v1.3.10, Node.js v0.10.31 and MongoDB 2.6.7.

Use checkboxes to save your progress

Install Node.js

Install MongoDB

Recommendations for Completing this Tutorial

Throughout the course of this tutorial, links to additional concepts and information will be included. You can use these links as supplementary material which can help you gain insight into the stack and its various components. As always, if you have any questions Google and Stackoverflow are your best bet. If you're unsure about something specific to this tutorial, feel free to ping me on twitter at @IAmMattGreen!

We at Thinkster are firm believers in actually writing code. Therefore we **strongly** encourage you to type out all the code instead of copy+pasting it.

Project Specifications

Before beginning work on any project, it's usually a good idea to know what you're building. Below is a basic list of things we want our users to be able to do:

- Create new posts
- View all posts ordered by upvotes
- Add comments about a given post
- View comments for a given post
- Upvote posts and comments

In addition to technologies that make up MEAN, we're going to enlist the help of several other libraries to help us achieve our goals:

- Mongoose.js for adding structure to MongoDB
- Angular ui-router for client-side routing
- Twitter Bootstrap for some quick styling

Jumping in with Angular

To begin this tutorial, we're going to start with the Angular side of things. AngularJS is a frontend framework developed by Google with the goal of making single page web applications easier to build, test, and maintain. Throughout this tutorials, we'll be linking to our A Better Way to Learn Angular guide which can provide supplementary information.

Without further ado, let's jump in...

Getting Started

As mentioned before, this tutorial will take you through building out a Hacker News/Reddit clone, which we're going to name "Flapper News". To keep things simple, we're going to kick things off with two files.

Create two empty files called `index.html` (for writing the template) and `app.js` (for defining our AngularJS logic)

To begin, our `index.html` will look like this:

```
<html>
  <head>
    <title>My Angular App!</title>
    <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.10/angular.min.js"></script>
    <script src="app.js"></script>
  </head>
  <body ng-app="flapperNews" ng-controller="MainCtrl">
    <div>
      {{test}}
    </div>
  </body>
</html>
```

Our `app.js` is going to look like this:

```
var app = angular.module('flapperNews', []);

app.controller('MainCtrl', [
'$scope',
function($scope){
  $scope.test = 'Hello world!';
}]);
```

With these several lines of code, we've set up a new AngularJS app and created a new controller. You'll notice that we declare a variable `test` in the controller and display its contents using the AngularJS `{{}}` notation. This is demonstrating one of the most powerful features of AngularJS, which is its two-way data-bindings.

If you aren't familiar with data-binding in AngularJS, read the AngularJS Guide on two-way data-binding

Displaying Lists

One thing that's is going to be absolutely fundamental to our app is displaying lists. Fortunately, angular makes this really easy using the ng-repeat directive.

To begin, we're going to modify our controller to include a new `$scope` variable that defines a list of post titles. Add the following code inside the controller function in `app.js`:

```
$scope.posts = [
  'post 1',
  'post 2',
  'post 3',
  'post 4',
  'post 5'
];
```

{info} The `$scope` variable serves as the bridge between Angular controllers and Angular templates. If you want something to be accessible in the template such as a function or variable, bind it to `$scope`

Now that we have a list of data we want to repeat, let's use `ng-repeat` to do it. Add the following code to line 8 of index.html, replacing the div that was there before:

```
<div ng-repeat="post in posts">
  {{post}}
</div>
```

When you refresh the page you should see a list of posts!

Now what if we want to display additional information about our posts? `ng-repeat` lets us do that too!

Let's amend our `posts` object to include some additional information we might be interested in displaying like the number of upvotes:

```
$scope.posts = [
  {title: 'post 1', upvotes: 5},
  {title: 'post 2', upvotes: 2},
  {title: 'post 3', upvotes: 15},
  {title: 'post 4', upvotes: 9},
  {title: 'post 5', upvotes: 4}
];
```

Now we change our `ng-repeat` directive to display the new information:

```
<div ng-repeat="post in posts">
  {{post.title}} - upvotes: {{post.upvotes}}
</div>
```

Of course it is important to order posts by number of upvotes, and we can tap into an angular filter to make it happen.

Add a filter to our posts based on the number of upvotes in descending order:

```
<div ng-repeat="post in posts | orderBy: '-upvotes'">
  {{post.title}} - upvotes: {{post.upvotes}}
```

```
</div>
```

AngularJS comes with several built in filters but you can also write custom filters tailored to your specific needs.

Getting User Input

Now that we've learned how to display lists of information with Angular, it'd really be great if we could have the user add posts. To do this, we first need to add a function to our `$scope` variable.

Create a $scope function that will add an object into the posts array:

```
$scope.addPost = function(){
  $scope.posts.push({title: 'A new post!', upvotes: 0});
};
```

When this function is invoked, it will append a new post to our `$scope.posts` variable. Now we're going to have to allow the user to actually execute this function.

Create a button that executes our addPost $scope function using the ng-click directive:

```
<button ng-click="addPost()">Post</button>
```

Great, we can now click a button and have a new post show up! Let's extend this by allowing the user to actually specify what they want the title to be. First, we need to build out the form in HTML and sprinkle it with some Angular Magic.

Create a form below the ng-repeat div that will allow us to enter custom posts:

```
<form ng-submit="addPost()">
  <input type="text" ng-model="title"></input>
  <button type="submit">Post</button>
</form>
```

Here we've created a form that encompasses our title text-box and 'Post' button. We are also now calling our `addPost()`function using the ng-submit directive, which has the added benefit of the user being able to press the 'enter' key to submit the form. Finally, we're using the ng-model directive to bind the contents of the text box to `$scope`. This will allow our controller to access the contents of the text box using `$scope.title`.

To accompany the changes to our template, we need to make some tweaks to `addPost()`.

Have the addPost function retrieve the title entered into our form, which is bound to the $scope variable `title`, and set `title` to blank once it has been added to the posts array:

```
$scope.addPost = function(){
  $scope.posts.push({title: $scope.title, upvotes: 0});
  $scope.title = '';
};
```

When we add a post we are now getting the title from `$scope.title`, which we then clear after the post has been created. At this point, it makes sense to prevent the user from posting a blank title.

Prevent a user from submitting a post with a blank title by adding the following line to the beginning of `addPost()`:

```
if(!$scope.title || $scope.title === '') { return; }
```

Enable Upvoting

Now that we can add some new posts, why don't we allow a user to upvote existing ones? To get started, lets revisit our `ng-repeat` directive.

Next to each post, we need to place a click-able element that will increment the corresponding post's upvote counter:

```
<div ng-repeat="post in posts | orderBy:'-upvotes'">
  <span ng-click="incrementUpvotes(post)">^</span>
  {{post.title}} - upvotes: {{post.upvotes}}
</div>
```

We've now added a `^` character inside a `<span>` tag that when clicked, calls the `incrementUpvotes()` function in our controller, but we don't have that function in our controller yet!

Define the `incrementUpvotes()` function in our controller:

```
$scope.incrementUpvotes = function(post) {
  post.upvotes += 1;
```

```
};
```

Notice that for this function we are passing the current instance of `post` to the function. This is happening by reference so when we increment upvotes, it gets automatically reflected back to the HTML page.

Submitting Links

Ultimately, Flapper News is about sharing links to content, so lets enable users to submit links along with their titles. We'll start by adding a second text box to our form that a user can use to submit a link. We'll also add some placeholder text to make it clear which form is which:

Add a link field to our form that is bound to the scope variable `link`:

```
<form ng-submit="addPost()">
  <input type="text" placeholder="Title" ng-model="title"></input>
  <br>
  <input type="text" placeholder="Link" ng-model="link"></input>
  <br>
  <button type="submit">Post</button>
</form>
```

Next we're going to want to modify our `addPost()` function to include the link (notice that we aren't going to force the user to submit a link if they don't want to):

```
$scope.addPost = function(){
  if(!$scope.title || $scope.title === '') { return; }
  $scope.posts.push({
    title: $scope.title,
    link: $scope.link,
```

```
    upvotes: 0
  });
  $scope.title = '';
  $scope.link = '';
};
```

Finally we need to modify the `ng-repeat` section to make the title a link to the content, but only if a link was specified.

We'll do this by using a new directive called ng-hide, which hides elements when an Angular expression evaluates to true.

Use to `ng-hide` to hide the linked version of the title if no link exists and correspondingly show the unlinked version:

```
<div ng-repeat="post in posts | orderBy:'-upvotes'">
  <span ng-click="incrementUpvotes(post)">^</span>
  <a ng-show="post.link" href="{{post.link}}">
    {{post.title}}
  </a>
  <span ng-hide="post.link">
    {{post.title}}
  </span>
  - upvotes: {{post.upvotes}}
</div>
```

It is worth noting that ng-show is merely the inverse of `ng-hide`. You can use either one for programmatically displaying or hiding elements.

Adding Some Style

At this point, we have the basics of an application - a user can add new posts which are automatically ordered based on the number of upvotes. Up until now, however, our interface has been lacking in the looks department. We can spruce it up a bit using some basic Bootstrap styling.

Change your `index.html` file to include Twitter Bootstrap along with a new layout:

```
<html>
<head>
  <title>Flapper News</title>
  <link href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css"
rel="stylesheet">

  <script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.10/angular.min.js"></script>
  <script src="app.js"></script>
  <style> .glyphicon-thumbs-up { cursor:pointer } </style>
</head>
<body ng-app="flapperNews" ng-controller="MainCtrl">
  <div class="row">
    <div class="col-md-6 col-md-offset-3">

      <div class="page-header">
        <h1>Flapper News</h1>
      </div>

      <div ng-repeat="post in posts | orderBy:'-upvotes'">
        <span class="glyphicon glyphicon-thumbs-up"
          ng-click="incrementUpvotes(post)"></span>
        {{post.upvotes}}
        <span style="font-size:20px; margin-left:10px;">
          <a ng-show="post.link" href="{{post.link}}">
            {{post.title}}
          </a>
          <span ng-hide="post.link">
            {{post.title}}
          </span>
```

```
        </span>
      </div>

      <form ng-submit="addPost()"
        style="margin-top:30px;">
        <h3>Add a new post</h3>

        <div class="form-group">
          <input type="text"
            class="form-control"
            placeholder="Title"
            ng-model="title"></input>
        </div>
        <div class="form-group">
          <input type="text"
          class="form-control"
          placeholder="Link"
          ng-model="link"></input>
        </div>
        <button type="submit" class="btn btn-primary">Post</button>
      </form>

    </div>
  </div>
</body>
</html>
```

At the top we've included Bootstrap from a CDN. In the body tag, we've made use of Bootstrap's grid system to align our content in the middle of the screen. We've also stylized the posts list and "Add a new post" form to make things a little easier to read. There's a lot more that could be done on this front so feel free to mess around with more styling before (or after) you continue.

(optional) Add your own styles!

Angular Services

Up to this point, we've been storing important data directly in the controller. While this works, it has some disadvantages:

- when the controller goes out of scope, we lose the data
- that data cannot be easily accessed from other controllers or directives
- the data is difficult to mock, which is important when writing automated tests

To rectify this problem, we're going to refactor our `$scope.posts` variable into a service.

My First Service... Is Really a Factory

In Angular, services are declared much like controllers. Inside `app.js`, we're going to attach a new service to our `flapperNews`module.

Create a factory for posts in `app.js` (our `MainCtrl` controller should appear below this):

```
app.factory('posts', [function(){
  // service body
}])
```

{info} By Angular conventions, lowerCamelCase is used for factory names that won't be `new`'ed.

You may be wondering why we're using the keyword *factory* instead of *service*. In angular, factory and service are related in that they are both instances of a third entity called *provider*.

The difference between them are nuanced but if you'd like to learn more check out Angular Factory vs Service vs Provider.

Now that we've created our service, lets go ahead and move our posts variable to it:

```
app.factory('posts', [function(){
  var o = {
    posts: []
  };
  return o;
}]);
```

What we're doing here is creating a new object that has an array property called `posts`. We then return that variable so that our `o` object essentially becomes exposed to any other Angular module that cares to inject it. You'll note that we could have simply exported the `posts` array directly, however, by exporting an object that contains the `posts` array we can add new objects and methods to our services in the future.

Injecting the Service

Our next step is to inject the service into our controller so we can access its data. Simply add the name of the service as a parameter to the controller we wish to access it from:

Inject posts service into `MainCtrl`

```
app.controller('MainCtrl', [
'$scope',
'posts',
```

```
function($scope, posts){
  ...

});
```

As you'll recall, two-way data-binding only applies to variables bound to `$scope`. To display our array of posts that exist in the posts factory (`posts.posts`), we'll need to set a scope variable in our controller to mirror the array returned from the service.

Bind the `$scope.posts` variable in our controller to the posts array in our service:

```
$scope.posts = posts.posts;
```

Now any change or modification made to `$scope.posts` will be stored in the service and immediately accessible by any other module that injects the `posts` service.

Angular Routing

Now that we understand the basics of Angular templates, controllers, and services, we're going to start diving into some of the concepts that make client side web applications so dynamic and powerful. To do this, we're going to need to learn how to deal with multiple views and controllers, which we will accomplish using the wonderful ui-router library.

To get started, lets include the library after we include Angular in our `index.html`:

```
<script
src="http://cdnjs.cloudflare.com/ajax/libs/angular-ui-router/0.2.10/angular-ui-router.js"></script>
```

Because we are adding an external module, we need to be sure to include it as a dependency in our app:

```
angular.module('flapperNews', ['ui.router'])
```

You may be wondering why we have chosen to use `ui-router` instead of the more standard ngRoute module - `ui-router` is newer and provides more flexibility and features than `ngRoute`. We will be using a few of these in this tutorial.

Adding a New State

Now that we have `ui-router` included, we need to configure it. In our `app.js`, we're going to use the Angular `config()`function to setup a *home* state.

Create the config block for our app and configure a home state using $stateProvider and $urlRouterProvider. Use `otherwise()` to redirect unspecified routes.

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

  $stateProvider
    .state('home', {
      url: '/home',
      templateUrl: '/home.html',
      controller: 'MainCtrl'
    });

  $urlRouterProvider.otherwise('home');
```

```
}]);
```

Here we set up our *home* route. You'll notice that the state is given a name ('home'), URL ('/home'), and template URL ('/home.html'). We've also told Angular that our new state should be controlled by `MainCtrl`. Finally, using the `otherwise()`method we've specified what should happen if the app receives a URL that is not defined. All that's left to do is define the`home.html` template. Instead of creating a new file, we are going to move most of our HTML into an inline template.

Add our inline home template right before the `</body>` tag in `index.html`:

```
<script type="text/ng-template" id="/home.html">
  <div class="page-header">
    <h1>Flapper News</h1>
  </div>

  <!-- rest of template -->
</script>
```

Using this syntax we can create templates inside our HTML and reference them in JavaScript.

Finally, we need to tell `ui-router` where to place the template of the active state.

Insert the `<ui-view>` tag where template should be rendered:

```
<body ng-app="flapperNews">
  <div class="row">
    <div class="col-md-6 col-md-offset-3">
      <ui-view></ui-view>
    </div>
```

```
  </div>

  ...
```

Whenever `ui-router` detects a route change, it will place the new state's template inside the tag and initialize the controller we specified in our state configuration. Notice how we have removed the `ng-controller="MainCtrl"` line from the opening `<body>` tag.

The Posts Page

Now that we've figured out how to create a state with `ui-router`, let's create a new one called *posts* that will display comments associated with a post.

In the config block in `app.js`, add a state where an individual post can be accessed:

```
.state('posts', {
  url: '/posts/{id}',
  templateUrl: '/posts.html',
  controller: 'PostsCtrl'
});
```

Notice that we define our URL with brackets around 'id'. This means that 'id' is actually a route parameter that will be made available to our controller.

As with the *home* state, we're also going to need to define both a new template and a new controller. Because we're going to associate comments with posts, we want to ensure our `posts` factory is injected into this controller so that it may access the comments data.

Create a new controller in `app.js` called PostsCtrl and be sure to inject the `posts` service:

```
app.controller('PostsCtrl', [
'$scope',
'$stateParams',
'posts',
function($scope, $stateParams, posts){

}]);
```

Faking comment data

Before we go any further, let's take a second to add some fake comment data to our `posts` model. This will help us mock up the basic comments view as well as ensure our routing is working properly.

In our addPost function in MainCtrl, add an array of fake comments to the posts object:

```
$scope.posts.push({
  title: $scope.title,
  link: $scope.link,
  upvotes: 0,
  comments: [
    {author: 'Joe', body: 'Cool post!', upvotes: 0},
    {author: 'Bob', body: 'Great idea but everything is wrong!', upvotes: 0}
  ]
});
```

Getting the Right Post

Since the *posts* page is about viewing the comments on a particular post, we need to use the `id` route parameter to grab the post and associated information. For now, we will consider the index of the post to be its id. We can use $stateParams to retrieve the id from the URL and load the appropriate post.

In PostsCtrl, set a scope object called `post` that grabs the appropriate post from the `posts` service using the id from $stateParams:

```
$scope.post = posts.posts[$stateParams.id];
```

Now that we have the `post` variable in our controller, we can display that information in our template.

Create a new inline template called "/posts.html" in `index.html` right before the `</body>` tag:

```
<script type="text/ng-template" id="/posts.html">
  <div class="page-header">
    <h3>
      <a ng-show="post.link" href="{{post.link}}">
        {{post.title}}
      </a>
      <span ng-hide="post.link">
        {{post.title}}
      </span>
    </h3>
  </div>

  <div ng-repeat="comment in post.comments | orderBy:'-upvotes'">
```

```
    <span class="glyphicon glyphicon-thumbs-up"
      ng-click="incrementUpvotes(comment)"></span>
    {{comment.upvotes}} - by {{comment.author}}
    <span style="font-size:20px; margin-left:10px;">
      {{comment.body}}
    </span>
  </div>
</script>
```

Finally, we'll add a link to the post's comment page next to the headline on the front page.

In the "/home.html" inline template, add the code below right after the title `</span>` tag:

```
<span>
  <a href="#/posts/{{$index}}">Comments</a>
</span>
```

{info} When iterating over an array, the `ng-repeat` directive makes an `$index` variable available along with each item in the array.

Creating New Comments

As with the creation of posts, we're going to want to allow our users to post new comments. This code looks very similar to what we've already written:

In our controller `PostsCtrl`, create an `addComment()` function:

```
$scope.addComment = function(){
  if($scope.body === '') { return; }
```

```
  $scope.post.comments.push({
    body: $scope.body,
    author: 'user',
    upvotes: 0
  });
  $scope.body = '';
};
```

Add the form below at the end of our inline template "/posts.html":

```
<script type="text/ng-template" id="/posts.html">

  <!-- post template -->

  <form ng-submit="addComment()"
    style="margin-top:30px;">
    <h3>Add a new comment</h3>

    <div class="form-group">
      <input type="text"
      class="form-control"
      placeholder="Comment"
      ng-model="body"></input>
    </div>
    <button type="submit" class="btn btn-primary">Post</button>
  </form>
</script>
```

Recap

In this first section, we've introduced you to some of the very basics of Angular.js including data-binding, controllers, services, and routing. In the process, we've created a skeleton web

application that allows a user to create a posting that can contain a link or a title, then create comments that are associated with those postings.

Up next we're going to learn how to use Node.js to implement a basic REST API for saving and retrieving posts and comments. Then we'll come back to the frontend and wire everything together into a single cohesive and functional web application.

Beginning Node

Now that we have the basic front-end for our Flapper News coded up with Angular, we're going to start work on our backend. Because Angular is able to handle the templating and rendering, our backend server will mainly serve as a communication layer with our database. Over the next few sections, we are going to focus on modeling our data needs and creating a REST API that our Angular app can use to interact with this data.

If you haven't already done so, make sure you have `node`, `npm`, and `mongodb` installed on your system. As a quick reminder, this tutorial assumes Node v0.10.31 and MongoDB 2.6.7.

Creating A New Project

We will begin by using express's generator. We can install this using the following command:

```
npm install -g express-generator
```

Now we can initiate a new node project with the following command:

```
express --ejs flapper-news
cd flapper-news
```

This will create a new folder called *flapper-news* and populate it with various files and folders.

{info} We are passing the `--ejs` flag because we'd like like our server to use standard HTML in its templates as opposed *jade*. Theoretically we are free to use Jade if we'd like to but the front-end is already written in plain HTML.

Our first order of business is to add the relevant npm modules we're going to need. When starting a new project, a generator will include a list of packages that are required by default.

Install these packages using the following command:

```
npm install
```

This will automatically download any packages specified in the `packages.json` file and store them in the `node_modules/`directory.

Next, we are going to install Mongoose, which is a library that provides schemas and models on top of MongoDB.

Run this command to install Mongoose via npm:

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

{info} The `--save` flag passed to npm install instructs the program to also save the package to the `packages.json` file. This allows you (or your teammates) to automatically install missing packages with the `npm install` command.

If you're using any version control software such as git, now is a good time to make your initial commit.

The Anatomy of Node Project

Right now the root directory of our Node project should look something like this:

```
app.js
bin/
node_modules/
package.json
public/
routes/
views/
```

Let's go step by step and explain what each folder/file is:

- `app.js` - This file is the launching point for our app. We use it to import all other server files including modules, configure routes, open database connections, and just about anything else we can think of.
- `bin/` - This directory is used to contain useful executable scripts. By default it contains one called `www`. A quick peak inside reveals that this script actually includes `app.js` and when invoked, starts our Node.js server.
- `node_modules` - This directory is home to all external modules used in the project. As mentioned earlier, these modules are usually installed using `npm install`. You will most likely not have to touch anything here.
- `package.json` - This file defines a JSON object that contains various properties of our project including things such as name and version number. It can also defines what

versions of Node are required and what modules our project depends on. A list of possible options can be found in npm's documentation.

- `public/` - As the name alludes to, anything in this folder will be made publicly available by the server. This is where we're going to store JavaScript, CSS, images, and templates we want the client to use.
- `routes/` - This directory houses our Node controllers and is usually where most of the backend code will be stored.
- `views/` - As the name says, we will put our views here. Because we specified the `--ejs` flag when initializing our project, views will have the `.ejs` extension as opposed to the '.jade' extension Jade views use. Although views are ultimately HTML, they are slightly different than any HTML file we might specify in the `public/` directory. Views are capable of being rendered directly by Node using the `render()` function and can contain logic that allows the server to dynamically change the content. Because we are using Angular to create a dynamic experience, we won't be using this feature.

In addition to the above files structure, we are going to add one more folder.

Create a new folder called 'models':

```
mkdir models
```

This folder will contain our Mongoose schema definitions.

Importing Our Angular Project

The final step before we begin building out the backend is to import our Angular portion into our Node.js project. First move the `index.html` file to the `views/` directory. Because we're using the *ejs* engine, Node is looking for files with the `.ejs`extension so we're going to have to rename our `index.html` to `index.ejs`, replacing the existing one.

Next, move the Angular `app.js` file to the `public/javascripts/` directory. To avoid confusion with Node's `app.js`, also rename the Angular file to `angularApp.js`.

Finally let's update the `<script>` tag in our `index.ejs` file to reflect these changes:

```
<script src="/javascripts/angularApp.js"></script>
```

Now we can start our application by running `npm start`.

If we point our browser to http://localhost:3000 we should be greeted with our Angular application.

Add our existing Angular code to the node project

Creating Schemas With Mongoose

Our first step in making a persistent data store is to configure our data models. To do this, we are going to be adding a schema layer on top of MongoDB using a nice library called Mongoose. Before we begin, let's make sure our MongoDB server is running.

If Mongo isn't running on your machine, enter this into your terminal:

```
mongod &
```

Next, we need to tell Node to connect to our local database on start.

Connect to our local MongoDB instance by adding the following code into our `app.js` file:

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

mongoose.connect('mongodb://localhost/news');
```

This will open a connection with the `news` database running on our Mongo server. Now we can create our first model.

In our `models/` directory create a new file called `Posts.js` and add the following code:

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

var PostSchema = new mongoose.Schema({
  title: String,
  link: String,
  upvotes: {type: Number, default: 0},
  comments: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Comment' }]
});

mongoose.model('Post', PostSchema);
```

Here we've defined a model called `Post` with several attributes corresponding to the type of data we'd like to store. We've declared our `upvotes` field to be initialized to 0 and we've set our `comments` field to an array of `Comment` references. This will allow us to use Mongoose's build in [populate()]mongoose populate method to easily retrieve all comments associated with a given `post`.

Next we register that model with with the global `mongoose` object we imported using `require()` so that it can be used to interact with the database anywhere else `mongoose` is imported.

In `app.js`, add the following line of code before our `mongoose.connect` call:

```
var mongoose = require('mongoose');
require('./models/Posts');

mongoose.connect('mongodb://localhost/news');
```

While were on the topic of making models, let's create a second model for storing comments in `models/Comments.js`:

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

var CommentSchema = new mongoose.Schema({
  body: String,
  author: String,
  upvotes: {type: Number, default: 0},
  post: { type: mongoose.Schema.Types.ObjectId, ref: 'Post' }
});

mongoose.model('Comment', CommentSchema);
```

In Mongoose, we can create relationships between different data models using the ObjectId type. The `ObjectId` data type refers to a 12 byte MongoDB ObjectId, which is actually what is stored in the database. The `ref` property tells Mongoose what type of object the ID references and enables us to retrieve both items simultaneously.

As always when you add a new file, remember to include it in `app.js`. Include this line of code right after the `require` for our Posts schema:

```
require('./models/Comments');
```

Opening REST Routes

With our backend models in place, it's now time to open some routes that the frontend client can interact with. The following are a list of actions a user can perform:

- view all posts
- Add a new post
- upvote a post
- view comments associated with a post
- add a comment
- upvote a comment

The actions map directly to several routes, which are described as follows:

- GET /posts - return a list of posts and associated metadata
- POST /posts - create a new post
- GET /posts/:id - return an individual post with associated comments
- PUT /posts/:id/upvote - upvote a post, notice we use the post ID in the URL
- POST /posts/:id/comments - add a new comment to a post by ID
- PUT /posts/:id/comments/:id/upvote - upvote a comment

Creating Our First Route

To keep things simple we will be defining these routes in the routes/index.js file. Let's begin by opening up the first route we listed, which should return a JSON list containing all posts.

Start by creating a *GET* route for retrieving posts in routes/index.js. It should return a JSON list containing all posts:

```
var mongoose = require('mongoose');
var Post = mongoose.model('Post');
var Comment = mongoose.model('Comment');

router.get('/posts', function(req, res, next) {
  Post.find(function(err, posts){
    if(err){ return next(err); }

    res.json(posts);
  });
});
```

First, we need to make sure that mongoose is imported and that we have handles to the Post and Comment models. Then we use the express get() method to define the URL for the route (/posts) and a function to handle the request. Inside our request handler, we query the database for all posts. If and error occurred, we pass the error to an error handling function otherwise we use res.json() to send the retrieved posts back to the client.

{info} When defining routes with Express.js, two variables will get passed to the handler function. req, which stands for "request", contains all the information about the request that was made to the server including data fields. res, which stands for "response", is the object used to respond to the client.

Before we can test that this route works, we need to have some data in our database. We can do this by creating a *POST* route for creating posts:

Notice that we're using router.post instead of router.get. This means that we will be making a POST request to the server (not to be confused with our Flapper News 'post' models and routes).

```
router.post('/posts', function(req, res, next) {
  var post = new Post(req.body);

  post.save(function(err, post){
    if(err){ return next(err); }

    res.json(post);
  });
});
```

The structure of the route is similar as above, however, we are using the `post()` method. We are also using `mongoose` to create a new `post` object in memory before saving it to the database.

Testing the Initial Routes

We can test these two routes using a command line tool called cURL.

First, we create a new `post`:

```
curl --data 'title=test&link=http://test.com' http://localhost:3000/posts
```

Next, we query the index route to insure it was saved:

```
curl http://localhost:3000/posts
```

If everything is functioning properly, you should see a JSON array of size 1 printed on the console with `title` and `link` set to 'test' and 'http://test.com' respectively.

Pre-loading Objects

One thing you might notice about the remaining routes we need to implement is that they all require us to load a `post` object by ID. Rather than replicating the same code across several different request handler functions, we can use Express's `param()`function to automatically load an object.

Create a route for preloading `post` objects in `routes/index.js`:

```
router.param('post', function(req, res, next, id) {
  var query = Post.findById(id);

  query.exec(function (err, post){
    if (err) { return next(err); }
    if (!post) { return next(new Error('can\'t find post')); }

    req.post = post;
    return next();
  });
});
```

In this example we are using mongoose's query interface which simply provides a more flexible way of interacting with the database.

Now when we define a route URL with `:post` in it, this function will be run first. Assuming the `:post` parameter contains an ID, our function will retrieve the `post` object from the database and attach it to the `req` object after which the route handler function will be called.

To demonstrate this, let's go ahead and create our route for returning a single post:

```
router.get('/posts/:post', function(req, res) {
  res.json(req.post);
});
```

Because the post object was retrieved using the middleware function and attached to the `req` object, all our request handler has to do is return the JSON back to the client.

(optional) Check out the query interface docs for Mongoose

Upvoting Posts

Now let's implement the route to allow our users to upvote posts. We'll do this by implementing a simple method in our `posts`schema to add one to the upvote count then save it.

Add an `upvote()` method to the `Posts` schema in `Posts.js`:

```
PostSchema.methods.upvote = function(cb) {
  this.upvotes += 1;
  this.save(cb);
};
```

Now we create the route in our `index.js`:

```
router.put('/posts/:post/upvote', function(req, res, next) {
  req.post.upvote(function(err, post){
    if (err) { return next(err); }

    res.json(post);
  });
});
```

To test this route, simply create a new post using the command above then run

```
curl -X PUT http://localhost:3000/posts/<POST ID>/upvote
```

You should see the post value returned back with the `upvote` property incremented.

Finishing Off With Comments

We've now completed all the basic routes for our `posts` object, now all we need to do is do the same for `comments`. Many of the same concepts apply with a few slight variations.

Firstly, when creating a new comment we need to be sure to include the `post` ID. Fortunately, this is already implicitly included in the request. In addition to creating and saving the `comment`, we're going to need to attach a reference to the new `comment`that refers to our `post` object.

Create `comments` route for a particular `post`:

```
router.post('/posts/:post/comments', function(req, res, next) {
  var comment = new Comment(req.body);
  comment.post = req.post;

  comment.save(function(err, comment){
    if(err){ return next(err); }

    req.post.comments.push(comment);
    req.post.save(function(err, post) {
      if(err){ return next(err); }

      res.json(comment);
    });
  });
});
```

Now, we can also take the exact same `upvote` method from `posts` and apply it to `comments`.

Implement the `/posts/:post/comments/:comment/upvote` route yourself. You should also implement another `router.param()` middleware function to automatically retrieve comments specified by the `:comment` route parameter.

Finally, we need to make a slight modification to our `GET /posts/:post` route

Use the `populate()` function to retrieve `comments` along with `posts`:

```
router.get('/posts/:post', function(req, res, next) {
  req.post.populate('comments', function(err, post) {
    if (err) { return next(err); }

    res.json(post);
  });
});
```

Using the `populate()` method, we can automatically load all the comments associated with that particular `post`.

Congratulations! If you made it this far, you should now have a functioning backend. There is still a significant amount of additional functionality we could add, however, you should now understand some of the basics of Node, Express, and Mongoose.

Up next, we'll learn how to use these routes in conjunction with our Angular.js frontend to create a web app where people can create `posts` and add `comments`.

Wiring Everything Up

Now that we have our basic backend implemented, we're going to wire up the angular app we made in the first part of this tutorial.

Loading Posts

Our first step is going to be to query our new backend for all `posts` using the index route. We do this by adding a new function inside our `posts` service.

Implement `getAll()` to retrieve posts in the `posts` service within `angularApp.js`:

```
o.getAll = function() {
  return $http.get('/posts').success(function(data){
    angular.copy(data, o.posts);
  });
};
```

{info} It's important to use the angular.copy() method to create a deep copy of the returned data. This ensures that the `$scope.posts` variable in `MainCtrl` will also be updated, ensuring the new values are reflect in our view.

Also be sure to inject the `$http` service:

```
app.factory('posts', ['$http', function($http){
  ...
}];
```

In this function we're using the Angular $http service to query our `posts` route. The `success()` function allows us to bind function that will be executed when the request returns. Because our route will return a list of `posts`, all we need to do is copy that list to the client side `posts` object. Notice that we're using the angular.copy() function to do this as it will make our UI update properly.

Now we need to call this function at an appropriate time to load the data. We do this by adding a property called resolve to our `home` state.

Use the `resolve` property of `ui-router` to ensure posts are loaded:

```
.state('home', {
  url: '/home',
  templateUrl: '/home.html',
  controller: 'MainCtrl',
  resolve: {
    postPromise: ['posts', function(posts){
      return posts.getAll();
    }]
  }
})
```

By using the `resolve` property in this way, we are ensuring that anytime our `home` state is entered, we will automatically query all posts from our backend before the state actually finishes loading.

Now, when you fire up the server and go to http://localhost:3000/#/home, you should see all the posts that exist in the database.

Creating New Posts

Up next, we need to enable creating new `posts`. As with loading posts, we're going to do this by adding another method to our `posts` service:

Add a method for creating new posts:

```
o.create = function(post) {
  return $http.post('/posts', post).success(function(data){
    o.posts.push(data);
```

```
  });
};
```

Now we can modify the `$scope.addPost()` method in `MainCtrl` to save `posts` to the server:

```
$scope.addPost = function(){
  if(!$scope.title || $scope.title === '') { return; }
  posts.create({
    title: $scope.title,
    link: $scope.link,
  });
  $scope.title = '';
  $scope.link = '';
};
```

Refresh the page then try adding a new post. You should immediately see it displayed and if you refresh the page, the post is still there! We now have persistent data.

Upvoting Posts

The last thing we need to wire up on the main page is upvoting posts.

Like in the previous examples, we'll add another method to our service:

```
o.upvote = function(post) {
  return $http.put('/posts/' + post._id + '/upvote')
    .success(function(data){
      post.upvotes += 1;
    });
```

```
};
```

And then in our controller, we simply replace `incrementUpvotes()` with this:

```
$scope.incrementUpvotes = function(post) {
  posts.upvote(post);
};
```

Here we use the `put()` method to upvote a `post`. When the call returns successfully, we update our local copy to reflect the changes. Now when you refresh the page, upvotes are persisted.

Finishing Off Comments

If you remember back to when we first wrote the template, we were treating the index of a `post` in the `posts` array to be its `id` field. Now that we are actually dealing with database entries, we need to make sure that when you click on the "Comments" link for a post, the application directs you to the proper URL.

Modify the `Comments` link to point to the proper route:

```
<a href="#/posts/{{post._id}}">Comments</a>
```

{info} MongoDB uses the `_id` property natively, so it's usually easier to just write our application with that in mind rather than have to translate it to an `id` field, which some might consider more intuitive.

When you click on the "Comments" link for a post, you should be directed to a new Angular URL that might look something like`http://localhost:3000/#/posts/53e27c7c363cf85ad8a84723`

What we are going to do is have Angular automatically load the full `post` object with `comments` when we enter this state. Like we did with the `home` state, we're going to use the `resolve` property in the route to accomplish this.

First, lets add a simple method in our `posts` service to retrieve a single `post` from our server:

```
o.get = function(id) {
  return $http.get('/posts/' + id).then(function(res){
    return res.data;
  });
};
```

Notice that instead of using the `success()` method we have traditionally used, we are instead using a promise.

Next we add the `resolve` object to our `posts` state:

```
.state('posts', {
  url: '/posts/{id}',
  templateUrl: '/posts.html',
  controller: 'PostsCtrl',
  resolve: {
    post: ['$stateParams', 'posts', function($stateParams, posts) {
      return posts.get($stateParams.id);
    }]
  }
```

```
});
```

The Angular ui-router detects we are entering the `posts` state and will then automatically query the server for the full post object, including `comments`. Only after the request has returned will the state finish loading.

To get access to the `post` object we just retrieved in the `PostsCtrl`, instead of going through the `posts` service, the specific object will be directly injected into our `PostsCtrl`.

We can modify our controller to gain access to it like so:

```
app.controller('PostsCtrl', [
'$scope',
'posts',
'post',
function($scope, posts, post){
  $scope.post = post;

  ...
});
```

Notice that we no longer have need to inject `$stateParams` into our controller. We're still going to want to inject `posts` to gain access to the methods for manipulating `comments`, however.

When you refresh the page, you should now see the title of the post displayed.

To enable adding `comments`, we can use the same technique we used for adding new `posts`.

First add a method to the `posts` service:

```
o.addComment = function(id, comment) {
  return $http.post('/posts/' + id + '/comments', comment);
};
```

Then call it from within the existing `addComment()` function:

```
$scope.addComment = function(){
  if($scope.body === '') { return; }
  posts.addComment(post._id, {
    body: $scope.body,
    author: 'user',
  }).success(function(comment) {
    $scope.post.comments.push(comment);
  });
  $scope.body = '';
};
```

Now you can add a comment that's persisted to the database.

Our final task is to enable the upvoting of comments:

```
o.upvoteComment = function(post, comment) {
  return $http.put('/posts/' + post._id + '/comments/'+ comment._id + '/upvote')
    .success(function(data){
      comment.upvotes += 1;
    });
};
```

In `PostsCtrl`:

```
$scope.incrementUpvotes = function(comment){
  posts.upvoteComment(post, comment);
};
```

Adding Authentication via Passport

Now that we have a application with a working server and client, let's add in the ability to create and authenticate users. To do this, we'll be using passport for authentications and JWT tokens for session management. This post has some great information about token vs session based authentication.

Creating the User model

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

var UserSchema = new mongoose.Schema({
  username: {type: String, lowercase: true, unique: true},
  hash: String,
  salt: String
});

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

Our users will be logging in with a username and password. Since we don't want to store our passwords in plain text, we'll need a field for storing the hash of the password. We'll also be generating and saving a salt whenever we set the password.

Now let's implement setting and validating passwords. We'll be using the `pbkdf2()` function which ships with node's native `crypto` module to hash our passwords.

Require the crypto module in our user model.

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

Create a `setPassword()` method on the `User` model that accepts a password then generates a salt and associated password hash.

```
UserSchema.methods.setPassword = function(password){
  this.salt = crypto.randomBytes(16).toString('hex');

  this.hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64).toString('hex');
};
```

{info} The `pbkdf2Sync()` function takes four parameters: password, salt, iterations, and key length. We'll need to make sure the iterations and key length in our `setPassword()` method match the ones in our `validPassword()` method

Create a `validPassword()` method that accepts a password and compares it to the hash stored, returning a boolean.

```
UserSchema.methods.validPassword = function(password) {
  var hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64).toString('hex');

  return this.hash === hash;
};
```

Finally, let's create instance method for generating a JWT token for the `user`. We'll be using the `jsonwebtoken()` module to help us generate tokens.

Install the `jsonwebtoken` module via `npm`.

```
npm install jsonwebtoken --save
```

Require the `jsonwebtoken` module inside the User model.

```
var jwt = require('jsonwebtoken');
```

Create the `generateJWT()` instance method on `UserSchema` with the following code:

```
UserSchema.methods.generateJWT = function() {

  // set expiration to 60 days
  var today = new Date();
  var exp = new Date(today);
  exp.setDate(today.getDate() + 60);

  return jwt.sign({
    _id: this._id,
    username: this.username,
    exp: parseInt(exp.getTime() / 1000),
  }, 'SECRET');
};
```

{info} The first argument of the `jwt.sign()` method is the payload that gets signed. Both the server and client will have access to the payload. The `exp` value in the payload is a Unix timestamp in seconds that will specify when the token expires. For this example we set it to 60 days in the future. The second argument of `jwt.sign()` is the secret used to sign our tokens. We're hard-coding it in this example, but it is ***strongly recommended*** that you use an environment variable for referencing the secret and keep it out of your codebase.

Setting up Passport

Now that we have our methods in place on our `UserSchema` for creating and authenticating users, let's install and setup Passport in our application. Passport uses strategies for different authentication methods (password, GitHub, Facebook, etc.) that are split out into separate modules. We'll be using the `passport-local` strategy to handle username/password authentication. See the official Passport guide for more information about Passport.

Install `passport` and `passport-local` via `npm`.

```
npm install passport passport-local --save
```

Create a folder `config` in the root of our application.

Create `config/passport.js` with the following code:

```
var passport = require('passport');
var LocalStrategy = require('passport-local').Strategy;
var mongoose = require('mongoose');
var User = mongoose.model('User');
```

```
passport.use(new LocalStrategy(
  function(username, password, done) {
    User.findOne({ username: username }, function (err, user) {
      if (err) { return done(err); }
      if (!user) {
        return done(null, false, { message: 'Incorrect username.' });
      }
      if (!user.validPassword(password)) {
        return done(null, false, { message: 'Incorrect password.' });
      }
      return done(null, user);
    });
  }
));
```

{info} Here we create a new `LocalStrategy` where we have our logic on how to authenticate a user given a username and password. Note that this function calls the `validPassword()` function that we just created. See the official passport configuration guide for more information.

Adding Passport to Our Application

Now that we have a configuration file for passport ready to use, let's incorporate it into our application.

Require passport in `app.js` after where we required `mongoose`

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

Require the user model after where we required `Post` and `Comment` models

```
require('./models/Users');
```

Require the passport configuration we just created **after** where we required our `User` model.

```
require('./config/passport');
```

Initialize passport after the `express.static` middleware.

```
app.use(passport.initialize());
```

Creating Authentication Endpoints

Now that Passport is setup in our application, we can create a `/login` endpoint to authenticate our users and return a JWT token.

Add a `/register` route to `routes/index.js` that creates a user given a username and password:

```
router.post('/register', function(req, res, next){
  if(!req.body.username || !req.body.password){
    return res.status(400).json({message: 'Please fill out all fields'});
  }

  var user = new User();

  user.username = req.body.username;
```

```
  user.setPassword(req.body.password)

  user.save(function (err){
    if(err){ return next(err); }

    return res.json({token: user.generateJWT()})
  });
});
```

Require `passport` in `routes/index.js`

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

Import the `User` mongoose model

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

Add a `/login` route to `routes/index.js` that authenticates the user and returns a token to the client:

```
router.post('/login', function(req, res, next){
  if(!req.body.username || !req.body.password){
    return res.status(400).json({message: 'Please fill out all fields'});
  }

  passport.authenticate('local', function(err, user, info){
    if(err){ return next(err); }

    if(user){
      return res.json({token: user.generateJWT()});
```

```
    } else {
      return res.status(401).json(info);
    }
  })(req, res, next);
});
```

The `passport.authenticate('local')` middleware uses the `LocalStrategy` we created earlier. We're using a custom callback for the `authenticate` middleware so we can return error messages to the client. If authentication is successful we want to return a JWT token to the client just like our register route does.

Securing Endpoints and Associating Posts and Comments with Users

Install the `express-jwt` module via `npm`.

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

Require `express-jwt` in `routes/index.js`

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

Create a middleware for authenticating jwt tokens in `routes/index.js`:

```
var auth = jwt({secret: 'SECRET', userProperty: 'payload'});
```

The `userPropery` option specifies which property on `req` to put our payload from our tokens. By default it's set on `user` but we're using `payload` instead to avoid any conflicts with passport (it shouldn't be an issue since we aren't using both methods of authentication in the same request). This also avoids confusion since the payload isn't an instance of our User model.

{info} Make sure to use the same secret as the one in `models/User.js` for generating tokens. Again, we're hard-coding the token in this example but it is *strongly recommended* that you use an environment variable for referencing your secret.

Now we can use the middleware we just defined to require authentication on specific routes. In our case, we'd want to authenticate users whenever they try to write to our application, such as when they're posting or commenting.

Require authentication for creating a post

```
router.post('/posts', auth, function(req, res, next) {
```

Require authentication for upvoting

```
router.put('/posts/:post/upvote', auth, function(req, res, next) {
```

Require authentication for commenting and set the author for comments

```
router.post('/posts/:post/comments', auth, function(req, res, next) {
```

Require Authentication for upvoting comments

```
router.put('/posts/:post/comments/:comment/upvote', auth, function(req, res, next) {
```

Finally, let's associate the authors with their posts and comments. Since we're authenticating with JWT tokens, we can get the username directly from the token's payload, saving us a trip to the database.

Set the author field when creating posts

```
router.post('/posts', auth, function(req, res, next) {
  var post = new Post(req.body);
  post.author = req.payload.username;
```

Set the author field when creating comments

```
router.post('/posts/:post/comments', auth, function(req, res, next) {
  var comment = new Comment(req.body);
  comment.post = req.post;
  comment.author = req.payload.username;
```

Now that our backend is ready to register and authenticate users, let's update the angular side of our application to handle this.

Creating an Angular Service for Authentication

We'll be using localStorage for persisting data to the client. This gives us a much easier interface for persisting data across sessions without having to deal with parsing cookies or handling cookies across multiple domains. If a JWT token exists in localStorage, we can assume the user is logged in as long as the token isn't expired. To log a user out, simply remove the token from localStorage. Check out the localStorage documentation on MDN for a better understanding of localStorage.

Let's start out by creating our initial auth factory. We'll need to inject `$http` for interfacing with our server, and `$window` for interfacing with `localStorage`.

```
.factory('auth', ['$http', '$window', function($http, $window){
  var auth = {};

  return auth;
}])
```

Next, let's create a `saveToken` and `getToken` function in our factory for getting and setting our token to `localStorage`

```
auth.saveToken = function (token){
  $window.localStorage['flapper-news-token'] = token;
};

auth.getToken = function (){
  return $window.localStorage['flapper-news-token'];
}
```

Now we'll need to add a function `isLoggedIn()` to our `auth` factory to return a boolean value for if the user is logged in.

```
auth.isLoggedIn = function(){
  var token = auth.getToken();

  if(token){
    var payload = JSON.parse($window.atob(token.split('.')[1]));

    return payload.exp > Date.now() / 1000;
  } else {
    return false;
  }
};
```

If a token exists, we'll need to check the payload to see if the token has expired, otherwise we can assume the user is logged out. The payload is the middle part of the token between the two .s. It's a JSON object that has been base64'd. We can get it back to a stringified JSON by using `$window.atob()`, and then back to a javascript object with `JSON.parse`.

Let's create a function `currentUser()` that returns the username of the user that's logged in.

```
auth.currentUser = function(){
  if(auth.isLoggedIn()){
    var token = auth.getToken();
    var payload = JSON.parse($window.atob(token.split('.')[1]));

    return payload.username;
  }
};
```

Finally, we'll need methods to log in, register, and log users out.

Create a register function that posts a user to our `/register` route and saves the token returned.

```
auth.register = function(user){
  return $http.post('/register', user).success(function(data){
    auth.saveToken(data.token);
  });
};
```

Create a login function that posts a user to our `/login` route and saves the token returned.

```
auth.logIn = function(user){
  return $http.post('/login', user).success(function(data){
    auth.saveToken(data.token);
  });
};
```

Create a logout function that removes the user's token from localStorage, logging the user out.

```
auth.logOut = function(){
  $window.localStorage.removeItem('flapper-news-token');
};
```

Creating the Login and Register pages

Now that our authentication factory is complete we can start using it in our application. Let's create a controller for our login and register page.

Create an authentication controller with the following code:

```
.controller('AuthCtrl', [
'$scope',
'$state',
'auth',
function($scope, $state, auth){
  $scope.user = {};

  $scope.register = function(){
    auth.register($scope.user).error(function(error){
      $scope.error = error;
    }).then(function(){
      $state.go('home');
    });
  };

  $scope.logIn = function(){
    auth.logIn($scope.user).error(function(error){
      $scope.error = error;
    }).then(function(){
      $state.go('home');
    });
  };
}])
```

We need to initialize a user on `$scope` for our form. Then, we can create a `register` and `logIn()` method on `$scope` to call the respective methods on the auth factory. We can then handle any errors and set `$scope.error` for displaying error messages later. Finally, if no errors occur, we can send the user back to the `home` state using a promise. Now we can go ahead and create our login and registration templates

Create the following registration template in views/index.ejs:

```
<script type="text/ng-template" id="/register.html">
  <div class="page-header">
    <h1>Flapper News</h1>
  </div>

  <div ng-show="error" class="alert alert-danger row">
    <span>{{ error.message }}</span>
  </div>

  <form ng-submit="register()"
    style="margin-top:30px;">
    <h3>Register</h3>

    <div class="form-group">
      <input type="text"
      class="form-control"
      placeholder="Username"
      ng-model="user.username"></input>
    </div>
    <div class="form-group">
      <input type="password"
      class="form-control"
      placeholder="Password"
      ng-model="user.password"></input>
    </div>
    <button type="submit" class="btn btn-primary">Register</button>
  </form>
</script>
```

Create the following login template in views/index.ejs:

```
<script type="text/ng-template" id="/login.html">
  <div class="page-header">
    <h1>Flapper News</h1>
  </div>

  <div ng-show="error" class="alert alert-danger row">
    <span>{{ error.message }}</span>
  </div>

  <form ng-submit="logIn()"
    style="margin-top:30px;">
    <h3>Log In</h3>

    <div class="form-group">
      <input type="text"
      class="form-control"
      placeholder="Username"
      ng-model="user.username"></input>
    </div>
    <div class="form-group">
      <input type="password"
      class="form-control"
      placeholder="Password"
      ng-model="user.password"></input>
    </div>
    <button type="submit" class="btn btn-primary">Log In</button>
  </form>
  </script>
```

Finally, let's add two new states that make our login and register pages accessible:

Create two new states for the login and register page:

```
.state('login', {
  url: '/login',
```

```
  templateUrl: '/login.html',
  controller: 'AuthCtrl',
  onEnter: ['$state', 'auth', function($state, auth){
    if(auth.isLoggedIn()){
      $state.go('home');
    }
  }]
})
.state('register', {
  url: '/register',
  templateUrl: '/register.html',
  controller: 'AuthCtrl',
  onEnter: ['$state', 'auth', function($state, auth){
    if(auth.isLoggedIn()){
      $state.go('home');
    }
  }]
});
```

Here we're specifying an `onEnter` function to our states. This gives us the ability to detect if the user is authenticated before entering the state, which allows us to redirect them back to the `home` state if they're already logged in.

Adding Navigation

Let's add a navbar to our application so we can easily tell if the user is logged in or not.

Start by making a simple controller for our navbar that exposes the `isLoggedIn`, `currentUser`, and `logOut` methods from our `auth`factory.

```
.controller('NavCtrl', [
'$scope',
'auth',
function($scope, auth){
```

```
  $scope.isLoggedIn = auth.isLoggedIn;
  $scope.currentUser = auth.currentUser;
  $scope.logOut = auth.logOut;
}]);
```

Now we can add the markup for our navbar to our application:

```
<body ng-app="flapperNews">
  <nav class="navbar navbar-default pull-right" ng-controller="NavCtrl">
    <ul class="nav navbar-nav">
      <li ng-show="isLoggedIn()"><a>{{ currentUser() }}</a></li>
      <li ng-show="isLoggedIn()"><a href="" ng-click="logOut()">Log Out</a></li>
      <li ng-hide="isLoggedIn()"><a href="/#/login">Log In</a></li>
      <li ng-hide="isLoggedIn()"><a href="/#/register">Register</a></li>
    </ul>
  </nav>
```

We're using `ng-show` and `ng-hide` along with our `isLoggedIn()` function to determine if the user is authenticated. If they are, their username is displayed, along with a log out link which calls our `logOut()` function. If they aren't, a Log In and a Register link is shown instead.

Making the Rest of Our Application User-Aware

Since our routes that write to the database now require authentication, let's update our `posts` service to send the JWT token to the server on authenticated requests.

First, we'll need to inject the `auth` service to our `posts` service

```
.factory('posts', ['$http', 'auth', function($http, auth){
```

Next, we'll need to send up our JWT token as an `Authorization` header. The format for this header should be `Authorization: Bearer TOKEN.GOES.HERE`.

We'll need to pass the following object as the last argument for our `$http` calls for the `create`, `upvote`, `addComment`, and `upvoteComment` methods in our `posts` service:

```
o.create = function(post) {
  return $http.post('/posts', post, {
    headers: {Authorization: 'Bearer '+auth.getToken()}
  }).success(function(data){
    o.posts.push(data);
  });
};

o.upvote = function(post) {
  return $http.put('/posts/' + post._id + '/upvote', null, {
    headers: {Authorization: 'Bearer '+auth.getToken()}
  }).success(function(data){
    post.upvotes += 1;
  });
};

o.addComment = function(id, comment) {
  return $http.post('/posts/' + id + '/comments', comment, {
    headers: {Authorization: 'Bearer '+auth.getToken()}
  });
};

o.upvoteComment = function(post, comment) {
  return $http.put('/posts/' + post._id + '/comments/'+ comment._id + '/upvote', null, {
    headers: {Authorization: 'Bearer '+auth.getToken()}
  }).success(function(data){
    comment.upvotes += 1;
  });
};
```

We can now display the post authors in the `/home.html` template:

```
<span ng-show="post.author">
  posted by <a>{{post.author}}</a> |
</span>
```

We only want to show the add post and add comment forms if the user is logged in. To do this, our controllers need to be aware of the authentication state.

We'll want to inject `auth` into *both* `MainCtrl` and `PostsCtrl`, and expose the `isLoggedIn` method to `$scope`

```
$scope.isLoggedIn = auth.isLoggedIn;
```

Now we're able to only show the post form to authenticated users:

```
<form ng-submit="addPost()"
  ng-show="isLoggedIn()"
  style="margin-top:30px;">
```

And display a message to users who aren't signed in

```
<div ng-hide="isLoggedIn()">
  <h3>You need to <a href="/#/login">Log In</a> or <a href="/#/register">Register</a> before
you can add a post.</h3>
```

```
</div>
```

Finally, hide the comment form for unauthenticated users:

```
<form ng-submit="addComment()"
  ng-show="isLoggedIn()"
  style="margin-top:30px;">
```

And prompt them to log in or sign up instead

```
<div ng-hide="isLoggedIn()">
  <h3>You need to <a href="/#/login">Log In</a> or <a href="/#/register">Register</a> before you can comment.</h3>
</div>
```

We now have an application with authentication!

Where To Go Next

Throughout this tutorial we've seen what a basic MEAN app might look like by creating a basic Angular application with a persistent Node+Express backend. By now, you should hopefully have an understanding of how these technologies interact as well as the ability to make modifications and add new features. If you're looking to sharpen your skills through practice, here are some suggestions on modifications you can add to Flapper News:

- feature downvote: Implement a 'downvoting' feature
- feature vote once: Only allow authenticated users to vote once.

- feature number of comments: Display the number of comments next to each post on the main page
- feature hide new comments box: use `ng-hide` to hide the 'new comment' and 'new post' input box until a user clicks a button to see the field
- feature specify name when commenting: Create an 'Author' field so people can specify their name when commenting