Simulating a Twitter Clone Application using MEAN stack

Introduction:

Giving the impression of Twitter clone to the users who loves to stay connected 24*7 on social platforms like Facebook, Twitter, Instagram etc. In this project I have tried to mimic the same behavior and come up with the SAAS using Platform as a service popularly known as Heroku.

Initial Thoughts:

1) Needed a MVC framework to develop client side View of the Application.

   Options Available: Angular, React, Ember, Backbone

   Framework chosen: Angular

   Reasons:

   - Open source, maintained by Google
   - Client side MVC framework
   - Excellent data bindings between Model and the view
   - Easy to test and Extremely lightweight
   - Most importantly due to being very lightweight Page load times is very minimal relative to the other framework available.

2) Needed a server side framework or scripting language

   Options Available: Node, Golang, Php

   Framework chosen: Node

   Reasons:

   - Non-blocking, I/O model that makes it lightweight and faster.
   - NPM, is the largest ecosystem of open source libraries in the world
   - Express module inside NPM is extremely useful
   - Express abstracts away a lot of low level logic
   - Takes up heavy Lifting of Creating our routes and our Restful API

3) Structure of the Data and Database chosen

   - User Posts:

     1) Created by 2) Created at 3) Post Content

   - User Details:

     1) Username 2) Password 3) Created at

   - Options available: RDBMS or NoSQL Database

- Chosen: NoSQL DB (Auto-sharding, Agile sprints, quick schema iteration, and schema-free) – MongoDB over Couch Base because Retrieving documents in mongo is more like how you write SQL query relative to creating view every time for a query in Couch Base. Mongo provides operators for most Boolean matches, and pattern matching and full text search as well. You can define indexes to help speed up your results.

4) Require a Platform to deploy the final application
   Options Available: Heroku, Azure, Amazon EC2
   Framework chosen: Heroku

Overall Summary:

1. Create a dynamic front end by using the Angular.js front end framework.
2. Run a server-side application with authentication services on Node.js Model.
3. Store data in MongoDB.
4. Build an API using Express.
5. Tying up the front end and server side.
6. Testing API routes using chrome tools POSTMAN or Advanced Rest Client.
7. Deploying it all on Heroku.

Steps and Methodology:

Module 1: Setting up the front end with Angular.js:

1. Creating an Angular Module and Controller which is going to control all the functionality for application front end.
2. Setting up a template main.html that would contain a form for post creation as well as a place to display the feed of posts.
3. Connecting Model and View using scope variable inside controllers and other directives like ng-model in View which supports two-way data bindings.
4. Creating Register and Login page:
   - Define an Authentication controller
   - Authentication templates sharing the same controller
5. Creating a Single Page Application by creating partials of three views main.html, register.html, login.html that will dynamically load into the view, index.html file using **ng-view** directive of angular as when required.
6. Routing between views is done using **ng-Route** which doesn't come as part of npm, it needs to be referenced in script tags using a CDN link in index.html file.

7. Application routes in Angular are declared using the **$routeProvider**, which will wire up templates and controllers depending on the browser's location.

Module 2: Setting up the server side of the Application using Node.js and Express:

1. Generating a blank express application and install all the dependencies manifested inside **package. json** file which are required to jumpstart the application.
2. In the routes folder delete the user.js file, and create two new JavaScript files, authenticate.js and api.js.
3. App.js is right now the boiler plate we got from express that is used to run the application and we need to import our API routing code and register them with Express using **require** keyword.
4. Now we're done with setting up authenticate.js and api.js as application routers. Now we need to implement it.
5. Implementing the Restful API:
    - RESTful APIs follow a convention which present resources to the client.
    - Here Post is a resource so its required to create a /posts api for which we create a router in api.js.
    - Every router begins with a require to express and, using the express Router class.
    - At the end of the router implementation its required to export the module as Router is to be consumed by the code written in main App.js file.
    - To add /posts api handlers we use express to register the handler to specific HTTP method, such as GET, PUT, POST or DELETE.
    - Test the api router with routes /api/posts with some general text for GET and POST using POSTMAN or ADVANCE REST CLIENT.

6. Adding the Authentication API:
    - Passport module from Node is used to implement authentication api. Need passport module, express-session and bcrypt which will store the password as hashes.
    - Require session middleware called as Express-session and then change the middleware section to use session module with a secret key.
    - Need to bootstrap passport and then add passport as an application level middleware at the bottom of the middleware chain.
    - In order to properly initialize passport we need to initialize it with its own middleware which will tell passport how to persist users in our data store.
    - Create passport-init. js file and paste boiler plate passport initialization code.
    - Passport needs to be able to serialize and deserialize users to support persistent login sessions.
    - Finally, we have to initialize passport with the authentication strategies we've defined in passport-init.js. Add this segment of code after the middleware section in app.js

   - This will call the passport initialization and allow those functions to be called whenever a user needs to authenticate.

7. Implementing the Auth routes in authenticate.js file and integrating passport module:
   - **Authenticate.js** will be written as a router, similar to **api.js** except it will expose a function that will take the passport module and return the router.
   - Here /login, /signup and /signout routes are implemented. Passport provides the successRedirect and failureRedirect fields to redirect the client to the correct endpoint after an attempted sign-in.

8. Protecting the API with Authentication using Middleware:
   - For this application we want anyone to read the post but only we want that only our registered users should be able to create new posts much like twitter does.
   - Need to add a middleware in api.js that checks if the user is authenticated to access HTTP methods like POST, PUT. In case User is not authenticated this middleware will protect our other router handlers and will only responds to the GET request.

9. Test the auth api routes /auth/login, /auth/signup, /auth/signout with POSTMAN to check everything is working as expected.

Module 3: Integrating the application using MongoDB

1. Use npm to install mongoose which works as an object data mapper that would specify a schema for Post and User objects.
2. Create two schemas in a model.js file inside models folder after importing mongoose.
3. It maintains 1:1 relationship between a Schema and a MongoDB Collection.
4. Finally use the schemas to register a User and Post model with mongoose.
5. In app.js import the mongoose model and establish a connection to local db.
6. Implementing the Authentication API to use MongoDB.
7. Implementing the Posts APIs with MongoDB.

Module 4: Tying things with Angular to show our server responses on the View

1. Need an authentication variable to store our authentication state that all of our controllers can access.
2. Each controller only has access to its own $scope, but they all have access to the $rootScope of the module.
3. Created two auth variables attached to $rootscope namely authenticated and current_user.
4. Handling authentication responses:

    - Upon login, call to /auth/login endpoint is necessary and if it’s successful set the authentication variable and current_user accordingly and redirect the user to main file showing all the posts.
    - Same goes with /auth/register as well as /auth/signout.

5. Create authentication sensitive elements in index.html file using directive ng-click to call the signout function if logout is clicked and also use ng-show or ng-hide to check out the authenticated status and show or hide the login and registration links when the user is logged in.
6. Create a simple **postService** factory with a getAll function that calls out to our API to get all the the posts and also don’t forget to inject the postService in maincontroller as a dependency just like done for $http service.
7. Since API is a fully RESTful one, Angular has a service that we can use instead of having to manually call out to our endpoint with each type of request, called **ngResource**. It doesn’t come in node modules so its explicitly declared in the project.
8. Simply use the $resource in postService factory, pass endpoint to $resource, and start performing CRUD operations. Now use the query method to GET all of our posts instead.

Module 5: Deploy the final application on Heroku and setup monglab connection in app.js

1. Install the herokubelt to use it from CLI.
2. Create an app on Heroku, which prepares Heroku to receive your source code.
3. Change the app.js to support the application run both in development environment as well as on the cloud using the heroku and mongolab connection.
4. Also create a ProcFile in your project right in where Package.json is and write the script to run the node.js server on heroku.
5. Git push heroku master and then you can run heroku open in the shell to go to the link genearated by heroku where your application has been deployed.

Deployed link: https://murmuring-garden-28663.herokuapp.com/#/

Github link: https://github.com/rgupta8493/TwitterAPP.git