Final Report

Project: Online Messenger Prototype

Introduction

This project is to create an online messenger prototype which users can login, logout and post messages on a whiteboard. Users can delete and edit their own posts and read others' posts. Registration is required before posting any new message and one must to login before editing or deleting messages. Edit and Delete button only shows when user is logged in. Error message will display if any error is returned from backend REST APIs.

Design and Implementation

The Web Application is implemented using Angular2 and NodeJS frameworks. Frontend and backend design decisions and implementation are made separately.

Backend Implementation and Design

main features are create, receive, update, delete messages and creation, validation of user and authorization of access to REST APIs. Only 2 models exist in this project: user and message.
Message Model:

```
 {
    content: {type: String, required: true},  // field to store the content of a message
    user: {type: Schema.Types.ObjectId, ref: 'User'  // An Mongoose ObjectId points to the author.
}
```

User Model:

```
{
    firstName: {type: String, required: true},
    lastName: {type: String, required: true},
    password: {type: String, required: true},
    email: {type: String, required: true, unique: true}, // unique keyword to ensure no duplicates on email
    messages: [{type: Schema.Types.ObjectId, ref: 'Message' // array of message references.
}
```

To Resolve various request from frontend, 3 services is built: application, user, message and application.
Im application routing file, a get api is implemented to render the main page of the application.

```
router.get('/', function (req, res, next){ // render the main page when application is being loaded.
})
```

User routing file is mainly responsible for user validation when login the app and create new user account.

```
router.post('/', function(req, res, next){ // Create a new User and save it to the Collection User.
                                           //Return error or success object.
})
router.post('/signin', function(req, res, next){ // Find the user in user collection. Use bscypt middleware to
verify if the password is valid. Give client a token upon successfully verified with jsonwebtoken plugin.
})
```

Message routing file deals with the request of deleting, posting, editing and receiving a message.

```
router.get('/', function(req, res, next) { // Find all messages in Message Collection.
```

```
                    //Populate the firstName field in user attribute to display on frontend.
})
router.use('/', function(req, res, next) { //Use json web token plugin to verify if the token sent from client is valid. Prevent willful operation.
})
router.post('/', function (req, res, next){  //Create a new entry in Message Collection.
        // Firstly verify if the user can be found in the user collection and decode token with json web token plugin. Save the newly created message after verification.
})
router.patch('/:id', function(req, res, next) { //Update a message saved by client. Step1: Verify if the user is who posted the message by comparing message found in database by id attached the the url and decoded user token.
// Step 2:  Reset the content of the message to the new content and save the updated one back.
})
router.delete('/:id', function(req, res, next) { // Delete message.
                //Step1: Verify if the user is valid. Step2: delete the message with id attached to the url.
})
```

Outside of routing folder, application.js file is the setting of all middlewares and plugin and linkage to mongDB.

Frontend Implementation and Design

In angular2, all code are organized into different components. The whole application is a component which is made up by different subcomponents. Components are defined by features to be implemented. In this project, the Application is generally consisted by 4 components: The App component, Messages component, Authentication component and Error component.
Each component can be compounded with other components. For an undividable component, it is usually made by an HTML file(view), a js file written in typescript (controller), a js file stores the user defined datatypes(model), a module file set up dependencies and services to be used in the other files and a service file dealing with useful functionalities and communication provided within the component and a routing file if it includes multiple pages.

Component structure of the frontend:

⇒ : being consisted by

Application ⇒ authentication ⇒ login, signup, signin, authentication services,
routing logics, user frontend model
⇒ messages ⇒ message-input, message-list, messages, message, message services,
routing logics, message frontend model
⇒ errors ⇒ error, error frontend model, error services
⇒ header

To implement the above structure, simply write codes for each file: components.html file controls content layout and you may use two-way data binding to create response user experience. In component.ts file import component from angular2 core and write class for display logic of the view. You can define new CSS style or usage of services with component decorator. In service.ts file, you are going to implement frontend and backend communication using Http, Response and Header from Angular Http core. In module.ts file,

export an empty class and define all modules, components, services and routing that will be used in the current component. In routing file, you mainly write routing logic using Routes and RouterModule from Angular2 http module.

Key Design Decision

Angular2 vs Angular1

Though Angular1 was taught in this specification track, I have encountered lots of unexpected errors or strange behavior when developing using Angular1. Angular2 provides a much clean and concise solution to organize frontend codes into different components instead of attempting to reuse controller code like in Angular1. No need to care about $scope or dependency injection and strange syntax in Angular1. Thus for this project I use Angular2 as frontend framework.

Session Storage vs Local Storage of Token

After a user sign in, a token for the user will be generated using jsonwebtoken plugin. The token can be stored in a session like other variables in frontend code and also can be stored in local storage so that users do not to login again whenever he/she leaves app page. The drawback of local storage is the security issue. A token encodes users information such as id, email address and password which is danger to expose them at frontend. The drawback of session storage is decreasing user experience. User need to sign in each time he/she close the web browser.
Here I choose to use local storage rather than session storage because this is not a project but a prototype for demonstration. We do not need to take care too much about security which is very important in product in contrast. Also it makes logout feature easy to implement. You only need clear out token in the local storage of browser and user is automatically logged off.

Loading all ahead vs Lazy Loading afterwards

Usually every components used in routing logic will be loaded when launching application, but angular2 provides an option that browser do not need to load all of the page at the beginning but load whatever is need for displaying initially and load other components if user trying to browse it. In light of the increasement of user experience by decreasing launching time, I choose to delay the loading of user authentication pages and load authentication components after user trying to access it.

Frontend User Validation vs Backend User Validation

To verify if a user has authorization to edit or delete a message, it can be implement using frontend validation plugins and modules or backend validation techniques. I implemented both for the sake of security. A backend implementation must be done to prevent user access other users' data and third party access to private REST API on server. Also a prescreen on frontend is also useful, for example to check if email address is valid, if user has entered something etc.

Error Handling and Displaying
Errors may comes from different places: invalid email address, database access errors, not finding users in mongoDB or unauthorized operation on messages. The main error displaying is separated into 2 forms. For form validation, it is displayed by changing color of the input field and unenable the submit button. For

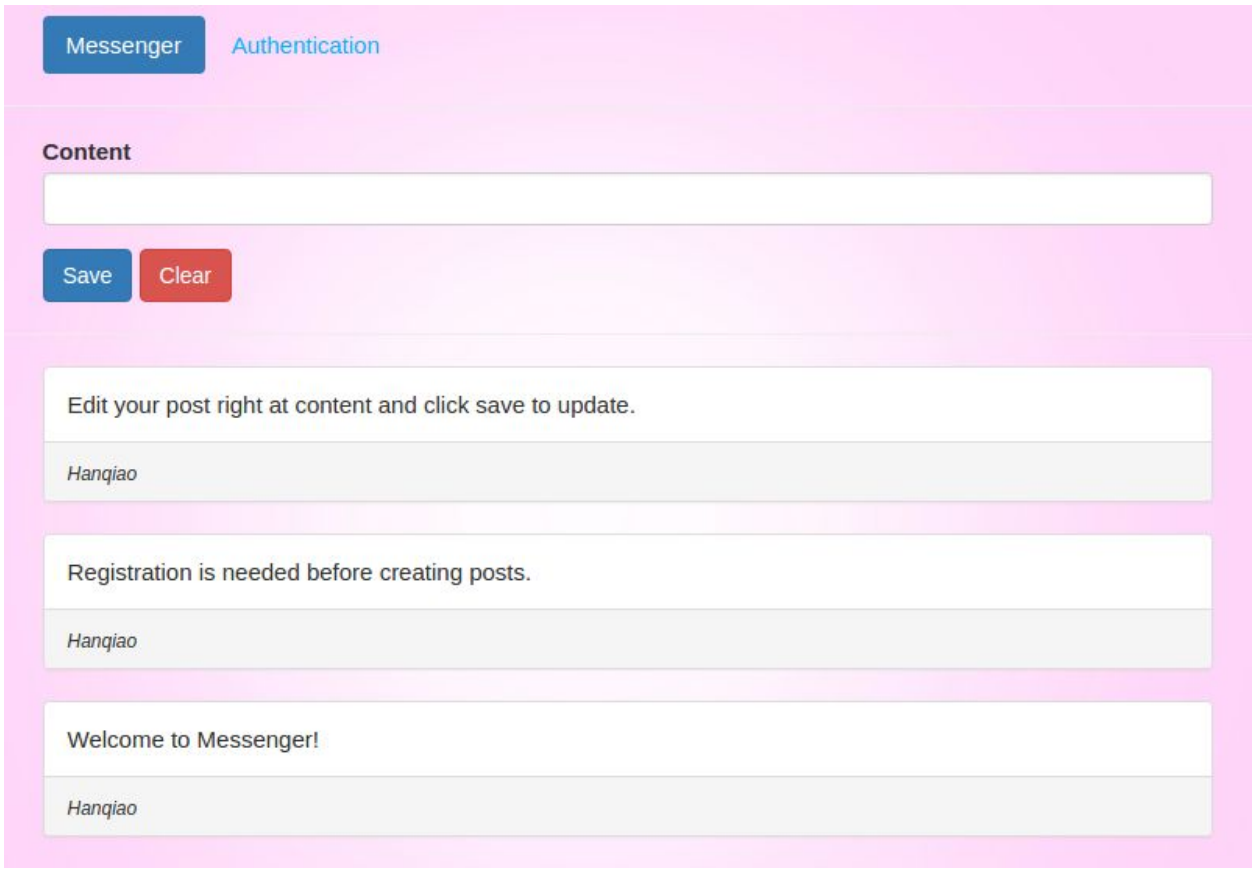
backend database errors or user validation errors, it will give a popover in frontend and display the error messages and title.

Third Party Libraries and Modules

For encrypting and decoding password, I choose to use bcrypt node module. It is a lightweight and simple to use encryption libraries and you can define the level of strength of encryption.
For user token generation, I use jsonwebtoken. Easy to use and you can verify if a token is valid just by calling verify method of it.
For a much friendly connection to mongoDB, I use mongoose module. It is much easier to save, find or create new collection with mongoose rather than using the default client module of mongoDB.

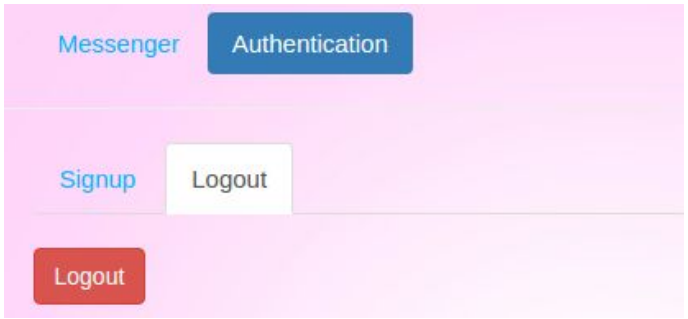
Screenshot of Online Messenger

Main Page:



User Authentication Page:

After Signin Showing Sign out Button



After signin, user can create, delete, edit the posts by himself:

Conclusion

An online messenger prototype is successfully developed including features of user authentication, user sign up, sign in and sign out, posting, editing, deleting messages and display of user firstname automatically. The project is implemented with Angular2 for frontend and nodejs for backend. Whole application is compiled using webpack and deployed to AWS elastic beanstalk and mongoLab.
More features is expected to be developed in the future, such as animation of message posting, deleting and editing and selectively displaying messages that is set to be viewed by current user.
An improvement can be made to display new messages at front of stack instead of the last.