CSCC01 - Just on Time Documentation

Arya Sharma, Bhoomi Patel, Sanan Rao, Virthiya Uthayanan, Yuto Omachi

Table of Contents

Backend

server/routes/customerRoutes.js

This file combines all the customer routes together, now it is just handling the post and get requests for the customer i.e registering and logging in using a customer account.

Request

Mimetype	application/json
Method	POST
URL	/api/customer
Description	It takes in a req and res parameter, where req holds the data of the user trying to sign up. Signup then ensures that the user does not already exist and all the parameters are of proper format. If the data is of the proper format defined and the user does not already exist, the data will then be sent to MongoDB. If that succeeds, res returns 200 as its status, customer data as its data. Upon failure, res returns 400 as its status, and the corresponding error as its message.

Body Example	{
	"email": "email1223@gmail.com",
	"password": "my password",
	"firstName": "Yuto",
	"lastName": "Omachi",
	"suitNo": "123",
	"street": "someStreet",
	"city": "Toronto",
	"country": "Canada",
	"postalCode": "A1A2B2"
	}

Mimetype	application/json
Method	GET
URL	/api/customer
Description	It takes in a req and res parameter, where req holds the data of the user

	trying to log in. Login then ensures that the user all the parameters are of proper format and if the data is of the proper format defined and the user exists, the user password is compared . If that succeeds, res returns 200 as its status, customer data as its data. Upon failure, res returns 400 as its status, and the corresponding error as its message.

Body Example	{
	"email": "email1223@gmail.com",
	"password": "my password",
	}

server/routes/organizerRoutes.js

This file combines all the organizer routes together, now it is just handling the post and get requests for the organizer i.e registering and logging in using an organizer account.

Request

Mimetype	application/json
Method	POST
URL	/api/organizer
Description	It takes in a req and res parameter, where req holds the data of the user trying to sign up. Signup then ensures that the user does not already exist and all the parameters are of proper format. If the data is of the proper format defined and the user does not already exist, the data will then be sent to MongoDB. If that succeeds, res returns 200 as its status, customer data as its data. Upon failure, res returns 400 as its status, and the corresponding error as its message.
Body Example	{ "email": "email1223@gmail.com", "password": "my password", "firstName": "Yuto", "lastName": "Omachi", "suitNo": "123", "street": "someStreet", "city": "Toronto", "country": "Canada", "postalCode": "A1A2B2",

	“bankName”:”CIBC”, “branchNum”:”21”, “accountNum”:”someAccount” }

Mimetype	application/json
Method	GET
URL	/api/customer
Description	It takes in a req and res parameter, where req holds the data of the user trying to log in. Login then ensures that the user all the parameters are of proper format and if the data is of the proper format defined and the user exists, the user password is compared . If that succeeds, res returns 200 as its status, customer data as its data. Upon failure, res returns 400 as its status, and the corresponding error as its message.

Body Example	{
	"email": "email1223@gmail.com",
	"password": "my password",
	}

server/auth/strategies/customerRegisterStrategy.js

customerRegisterStrategy.js is a passport strategy responsible for authenticating requests, which is accomplished by implementing an authentication mechanism. This strategy validates that the request has all the required fields and has a unique email which is essential for every user. Also this strategy encrypts the password for the user so as to make the account and database info secure.

server/auth/strategies/organizerRegisterStrategy.js

organizerRegisterStrategy.js is a passport strategy responsible for authenticating requests, which is accomplished by implementing an authentication mechanism. This strategy validates that the request has all the required fields and has a unique email which is essential for every user. Also this strategy encrypts the password and all the bank info of the organizer for the user so as to make the account and database info secure.

server/auth/strategies/customerLoginStrategy.js

customerLoginStrategy.js is a passport strategy responsible for authenticating requests, which is accomplished by implementing an authentication mechanism. This strategy validates that the request has all the required fields and then it finds the customer in the database using the email and compares the password and returns success or failure. Also return failure when a customer doesn't exist.

server/auth/strategies/organizerLoginStrategy.js

organizerLoginStrategy.js is a passport strategy responsible for authenticating requests, which is accomplished by implementing an authentication mechanism. This strategy validates that the request has all the required fields and then it finds the organizer in the database using the email and compares the password and returns success or failure. Also return failure when a organizer doesn't exist.

server/auth/serialize.js

Serializing a user determines which data of the user object should be stored in the session, usually the user id. The serializeUser() function sets an id as the cookie in the user's browser, and the deserializeUser() function uses the id to look up the user in the database and retrieve the user object with data.

server/auth/index.js

index.js uses a function configPassportStratergy to combine all the passport strategies and the serializing together so that it can be easily accessed in the backend.

server/controllers/customerController.js

customerController.js is a file that is responsible for returning relevant error messages, status codes or success messages based on the error code or info returned by customerRegisterStratergy.js. For instance we get status code:400 and a relevant error message for errors like 'email is already in use' for non unique email, it returns status code 200 and user object when there are no errors.

server/controllers/organizerController.js

organizerController.js is a file that is responsible for returning relevant error messages, status codes or success messages based on the error code or info returned by organizerRegisterStratergy.js. For instance we get status code:400 and a relevant error message for errors like 'email is already in use' for non unique email, it returns status code 200 and user object when there are no errors.

Models

server/models/customerModel.js

customerModel.js defines the customer model with Mongoose Schema to present to the server and MongoDB the format of the customer. It imports Mongoose and each customer takes in an email, phoneNumber, firstName, lastName, street, suitNo, country, city, postalCode that are all of type String. An example of a customer model follows along with how it would be called and utilized:

```
const customer = new Customer(
{contact:{
        email,
        phoneNumber
        },
personalInfo: {
                firstName,
                lastName,
         address: {
                    suitNo,
                    stree,
                    city,
                    country,
                    postalCode,
                }
        },
password
}
```

server/models/eventOrganizerModel.js

eventOrganizerModel.js defines the eventOrganizer model with Mongoose Schema to present to the server and MongoDB the format of the eventOrganizer. It imports Mongoose and each event organizer takes in an email, phoneNumber, firstName, lastName, street, suitNo, country, city, postalCode, bankName, branchNum, accountNum that are all of type String. An example of a event organizer model follows along with how it would be called and utilized:

```
const organizer = new eventOrganizer(
{contact:{
        email,
        phoneNumber
        },
personalInfo: {
                firstName,
                lastName,
         address: {
```

```
                suitNo,
                stree,
                city,
                country,
                postalCode,
            }
        },
bankInfo:{
        bankName,
        branchNum,
        accountNum
        },
password
}
```

server/models/eventModel.js

eventModel.js defines the event model with Mongoose Schema to present to the server and MongoDB the format of the eventModel. It imports Mongoose and each event takes in an eventInfo, tags, bidHistory that are all of type String and an eventOrganizer object with the organizer id and name. An example of a event model follows along with how it would be called and utilized:

```
    const eventSchema = new Event({
        eventInfo: {
            type: eventInfoSchema,
            required: true
        },
        tags: {
            type: [String],
            required: true
        },
        bidHistory: {
            type: [bidInfoSchema],
            required: true,
            trim: true
        },
        organizerInfo: {
            id: {
                type: mongoose.Schema.Types.ObjectId, // refers to the eventOrganizer's id
                required: true
            },
            name: {
```

```
            type: String,
            required: true
        }
    }
});
```

server/models/schemas/address.schema.js

address.schema.js defines the address model with Mongoose Schema to present to the server and MongoDB the format of the address. It imports Mongoose and each address takes in a street, suitNo, country, city, postalCode that are all of type String.

server/models/schemas/contact.schema.js

contact.schema.js defines the contact model with Mongoose Schema to present to the server and MongoDB the format of the contact. It imports Mongoose and each address takes in an email and phoneNumber that are all of type String.

server/models/schemas/personalInfo.schema.js

personalInfo.schema.js defines the personalInfo model with Mongoose Schema to present to the server and MongoDB the format of the personal info. It imports Mongoose and each address takes in firstName, lastName that are all of type String and an object of type address.

server/models/schemas/organizer/bankInfo.schema.js

bankInfo.schema.js defines the bankInfo model with Mongoose Schema to present to the server and MongoDB the format of the bankInfo. It imports Mongoose and each bankInfo object takes in a bankName, branchNum, accountNum that are all of type String.

server/models/schemas/event/bidInfo.schema.js

bidInfo.schema.js defines the bidInfo model with Mongoose Schema to present to the server and MongoDB the format of the bid. It imports Mongoose and each bid takes in a bid price and date.

server/models/schemas/event/eventInfo.schema.js

eventInfo.schema.js defines the eventInfo model with Mongoose Schema to present to the server and MongoDB the format of the event. It imports Mongoose and each event takes in a name, description, time, address and status.

Frontend

client/src/components/header/Header.jsx

Header.jsx is a frontend page which displays a header with the justontime logo and a button which can be used to navigate between login and signup pages.

- Header
    - This function creates the navbar and adds the image and the user button on the header.

client/src/features/auth/authService.js

authService.js connects the frontend with the backend using axios.

client/src/pages/signup/signup.js

signup.js is the frontend page that allows a potential customer to become a registered user of Justontime. This page loads a display of the signup page. This page loads a form that takes 10 inputs: First name, Last name, email, phone number, street, city, country, postal code, password and confirm password. There is a ‘sign up’ button that allows the user to submit the data and a link to the login page if the user is already registered. All input on the page is required as such errors are sent to the user if any of the text fields are left empty. Errors are also sent to the signup page from the backend if the information provided is not accurate. For example if the provided email address does not exist. Methods in this file are as follows: handleSubmit, renderForm, renderback.

- handleSubmit
    - Ensures all the data fields are not empty and returns an error if there are any empty input fields
- renderForm
    - Creates the html code for the react component
- renderback
    - Creates the background for the signup page

client/src/components/login/login.js

login.js is the frontend page that allows a registered user of Justontime to use our website. This page loads a display of the login page. This page loads a form that takes 2 inputs: Username and password. There is a ‘login’ button that allows the user to submit the data and a link to the sign page if the user is not already registered. All input on the page is required as such errors are sent to the user if any of the text fields are left empty. Errors are also sent to the login page from the backend if the information provided is not accurate. For example if the provided

username does not exist. Methods in this file are as follows: handleSubmit, renderForm, renderback.

- handleSubmit
    - Ensures all the data fields are not empty and returns an error if there are any empty input fields
- renderForm
    - Creates the html code for the react component
- renderback
    - Creates the background for the login page