Integrate Notification System – Email

Relevel
by Unacademy

Class Agenda:

- Notification system at all levels
- Admins will receive email when clients make any changes
- Users will get mails on booking
- Clients will receive emails if admin makes changes to their theatres

Features we implemented so far:

- CRUD operation APIs on Theatre and Movie models and APIs relating movies and theatres.
- Authentication and Authorization, using JWT for token validation.
- Several middleware, from model data verification to user access level verification.
- Model for booking and transaction, like create booking, cancel booking and set timeout for a booking.

List of Concepts Evolved:

- Associate Client with Theatre
- Create Clients on server start
- What is notification and how we are going to implement it in our app
- Create Notification Utility file
- Notify user on doing payment
- Notify clients when theatre is created, updated or deleted
- Create a middleware to distinguish between an admin and a client

Associate Client with Theatre:

- The CLIENT level user in our application as supposed to be the owner of the theatres in our application.
- So, we need to have an association between a Theatre model and User model to implement this feature in our application.
- Below show is how we will link a user to a theatre i.e., by adding an attribute named “ownerId” in theatre model that will be of type “ObjectId” of User model.

```
ownerId: {
    type: mongoose.SchemaTypes.ObjectId,
    required: true,
    ref : "User"

}
```

Code link:
https://github.com/Vishwa07dev/mba_backend/blob/session8/models/theatre.model.js

Create Clients on server start:

- Now, when we have added a relation between Theatre and User models, let us have some initial data added in our server.js file.
- First, we will create three CLIENT type users as shown below and assign them to a variable to use while creating theatre model data.

Here, we are creating some initials CLIENT level users

```
let client1, client2, client3 ;
try {

    client1 = await User.create({
        name: "Client1",
        userId: "client01", // It should be atleat 16, else will throw error
        email: "Kankvish1@gmail.com",  // If we don't pass this, it will throw the error
        userType: "CLIENT",
        password :bcrypt.hashSync("Welcome1", 8) //this field should be hidden from the end user

    });
    client2 = await User.create({
        name: "Client2",
        userId: "client02", // It should be atleat 16, else will throw error
        email: "Kankvish2@gmail.com",  // If we don't pass this, it will throw the error
        userType: "CLIENT",
        password :bcrypt.hashSync("Welcome1", 8) //this field should be hidden from the end user

    });
    client3 = await User.create({
        name: "Client3",
        userId: "client03", // It should be atleat 16, else will throw error
        email: "Kankvish3@gmail.com",  // If we don't pass this, it will throw the error
        userType: "CLIENT",
        password :bcrypt.hashSync("Welcome1", 8) //this field should be hidden from the end user

    });
    console.log("Clients created");

} catch (e) {
    console.log(e.message);
}
```

As shown below, we have now added “ownerId” attribute in all the theatre objects and used the client user id which we used above.

```
//Creating few intial sets of Theatres
await Theatre.collection.drop();
await Theatre.create({
    name: "FunCinemas",
    city: "Bangalore",
    description: "Top class theatre",
    pinCode: 560052,
    movies : [movie1._id, movie2._id, movie3._id],
    ownerId : client1._id

});
await Theatre.create({
    name: "PVR Cinemas - Kormangala",
    city: "Bangalore",
    description: "PVR franchise theatre",
    pinCode: 560095,
    movies : [movie1._id, movie2._id, movie4._id],
    ownerId : client1._id

});
await Theatre.create({
    name: "IMax",
    city: "Bangalore",
    description: "IMax franchise theatre",
    pinCode: 560095,
    movies : [movie1._id, movie4._id],
    ownerId : client2._id

});
```

```
await Theatre.create({
    name: "Vaibhav Theatre",
    city: "Bangalore",
    description: "Economical theatre",
    pinCode: 560094,
    movies : [movie5._id, movie4._id],
    ownerId : client2._id

});

await Theatre.create({
    name: "Inox",
    city: "Pune",
    description: "Top class theatre",
    pinCode: 411001,
    movies : [movie5._id, movie2._id],
    ownerId : client3._id

});
await Theatre.create({
    name: "Sonmarg Theatre",
    city: "Pune",
    description: "Economical theatre",
    pinCode: 411042,
    movies : [movie3._id, movie2._id],
    ownerId : client3._id

});
```

What is notification and How to implement in our app:

Let us now talk about notification and how are we going to implement it in our application.

What is notification:

- Notification is basically another asynchronous mode of information confirmation after doing any type of registration or transaction on an application.
- Apart from confirmation on the application itself user gets an Email or SMS stating the same information as well.
- Now, since the application confirmation that a user is successfully registered or payment is successfully done, this another piece of communication is to provide a permanent form of message.
- Which the user can use to keep a record or as an evidence for any future requirements.

Notification Service App:

- To integrate notification via email into our application we will be using another backend service named NotificationService, available in below given code link.

- Code link: https://github.com/Vishwa07dev/NotificationService

- This application is also built **using expressJs, mongoose, node-cron and nodemailer**.

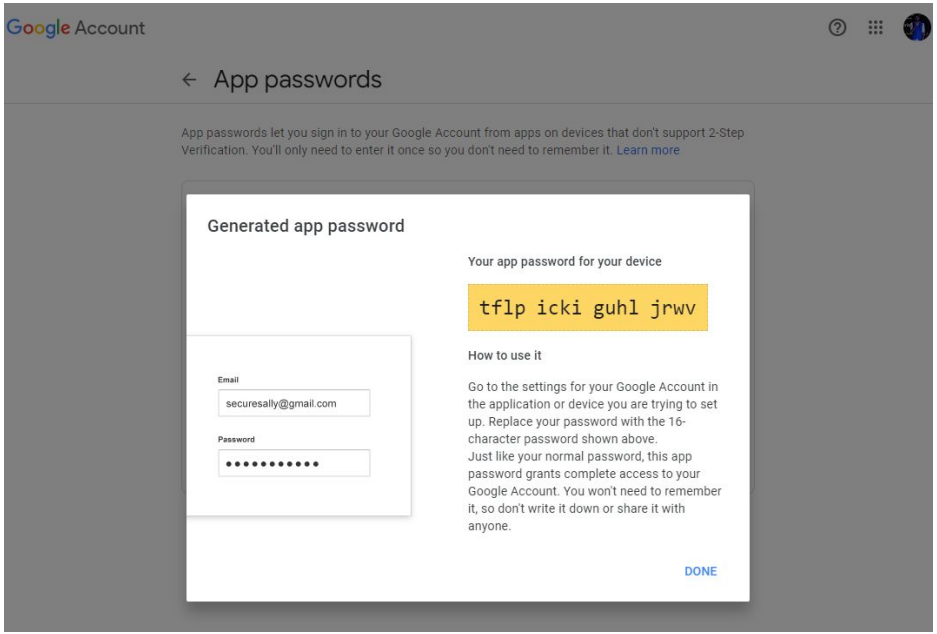
- So, this application exposes two APIs as given below:
  - POST /notifiServ/api/v1/notifications- submit a new notification object.
  - GET /notifiServ/api/v1/notifications/:id- get a notification detail by their id.

- This service uses node-cron package to create a scheduler which runs in every 30 seconds.
- For every 30 sends it runs a query to check for notification objects with “UN_SENT” status from the database.
- Once it has all the objects it runs a loop around each items and the uses SMTP host and nodemailer’s mailTransporter function to send the email.
- Once the emails are sent for each items the corresponding status of the notification object is set to “SENT”.

Steps to setup you own SMTP server using Google account:

1. Login or Signup for a google account.
2. Go to your account **Security** page: https://myaccount.google.com/security
3. In the **Signing in to Google section**, you will find **2-Step Verification**, which we need to enable. Click on it and follow the steps.

4. Next, again go to the Security page and Signing in to Google section and select App passwords this will be used to connect to our google account from an application running on a server. Click and setup your app password here and save the password created.



5. Now, you can use your credentials here, https://github.com/Vishwa07dev/NotificationService/blob/main/notifier/emailService.js

All you need to changes is your user and pass to make the notification service send an email successfully.

```
module.exports = nodemailer.createTransport({
    port: 465,               // true for 465, false for other ports
    host: "smtp.gmail.com",
    auth: {
        user: 'vish007dev@gmail.com',
        pass: 'Welcome@07',
    },
    secure: true,
});
```

Steps to run this service:

- git clone https://github.com/Vishwa07dev/NotificationService.git
- cd NotificationService
- npm install
- npm run devStart.

So basically, we will be consuming a REST API from our express based application which we can do by installing a package called “node-rest-client” to send request to another REST API and accept the response.

Code link:
https://github.com/Vishwa07dev/mba_backend/blob/session8/package.json

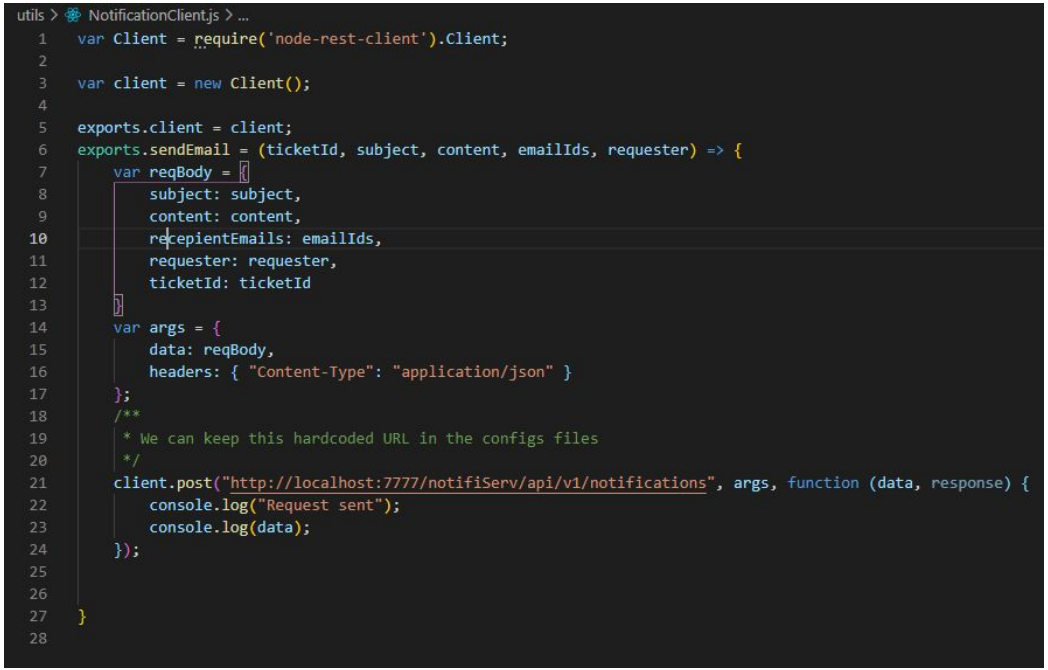
package.json > {} dependencies

```
{
  "name": "mba",
  "version": "1.0.0",
  "description": "Code base for the Movie Booking Application",
  "main": "server.js",
  Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "devStart": "nodemon server.js"
  },
  "author": "kankvish@gmail.com <Vishwa Mohan>",
  "license": "ISC",
  "dependencies": {
    "bcryptjs": "^2.4.3",
    "body-parser": "^1.19.2",
    "dotenv": "^16.0.0",
    "express": "^4.17.3",
    "jsonwebtoken": "^8.5.1",
    "mongoose": "^6.2.9",
    "node-rest-client": "^3.1.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.15"
  }
}
```

Create Notification Utility file:

- Now, we have the required service up and running with all the package we need.
- Let’s create a utility file which we can use the form the request body, send the request and accept the response.
- Shown below is the “sendMail” created in NotificationClient.js file in utils package.
- The method will accept a unique id for the notification object, subject of the mail, mail body, receiver’s mail id and requester mail which in this case is “mba-no-reply@mba.com”.

Code link:
https://github.com/Vishwa07dev/mba_backend/blob/session8/utils/NotificationClient.js



Notify user on doing payment:

It’s time to see our code in action. All we need to do is call the “sendMail” method from our “createPayment” controller when a payment is successful.

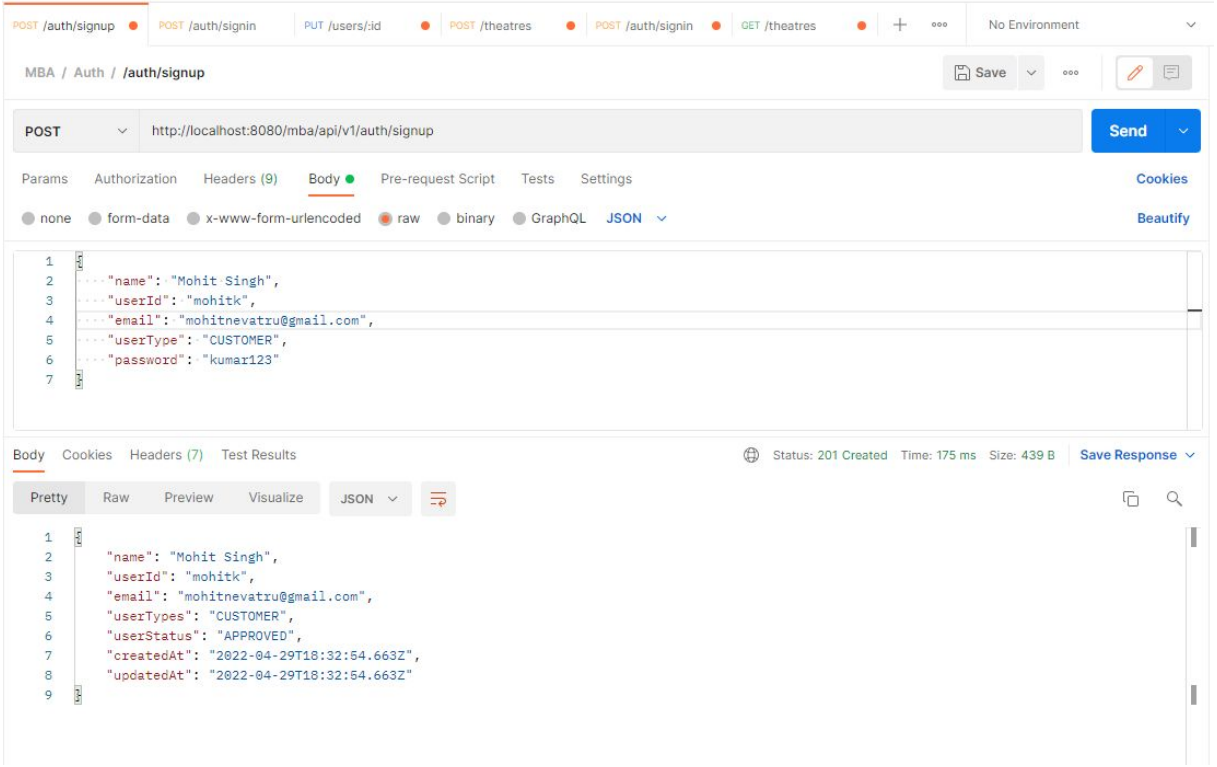
```
try {
    const payment = await Payment.create(paymentObject);
    /**
     * Update the booking status
     */
    booking.status = constants.bookingStatus.completed;
    await booking.save();

    const user = await User.findOne({"userId" : req.userId});
    sendEmail(payment._id, "Payment successfull for the booking id : " + req.body.bookingId, JSON.stringify(booking), user.email, "mba-no-reply@mba.
    com")
    /**
     * Send the confirmation email
     */

    return res.status(201).send(payment);
```

Code link:
https://github.com/Vishwa07dev/mba_backend/blob/session8/controllers/payment.controller.js

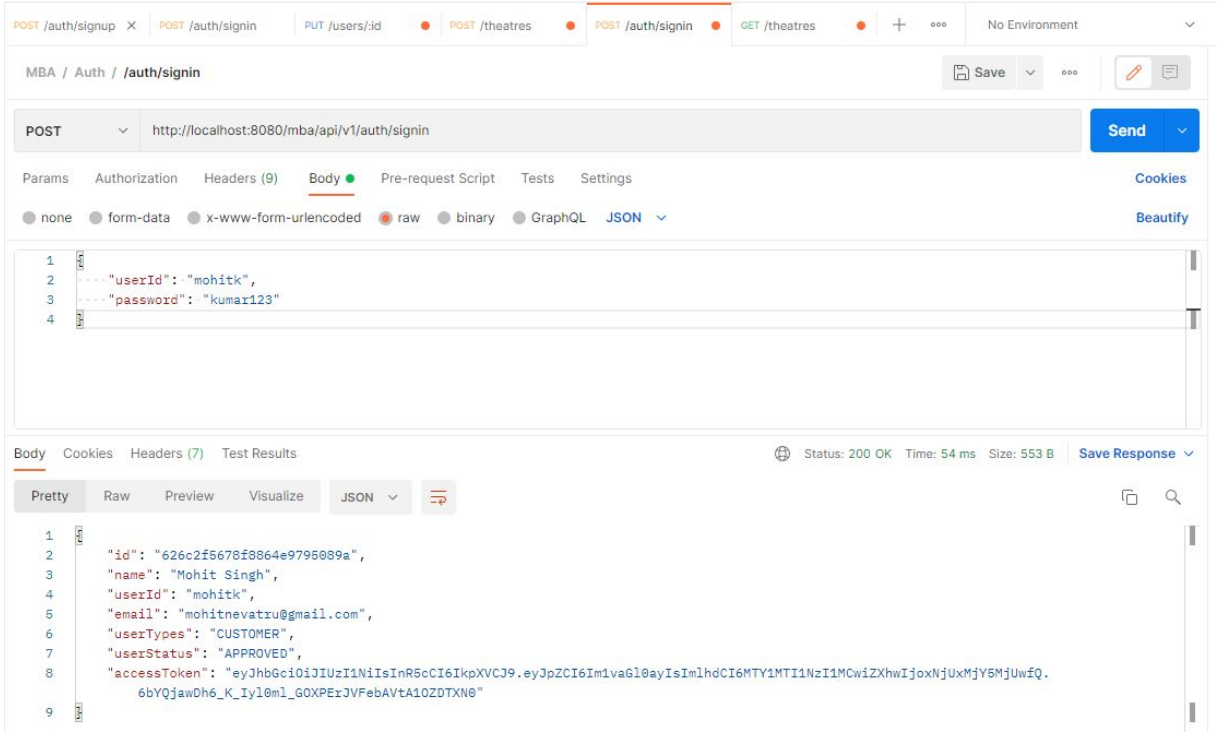
Testing the code:

Let us now test the code changes and verify that on successful payment the notification via mail is received.

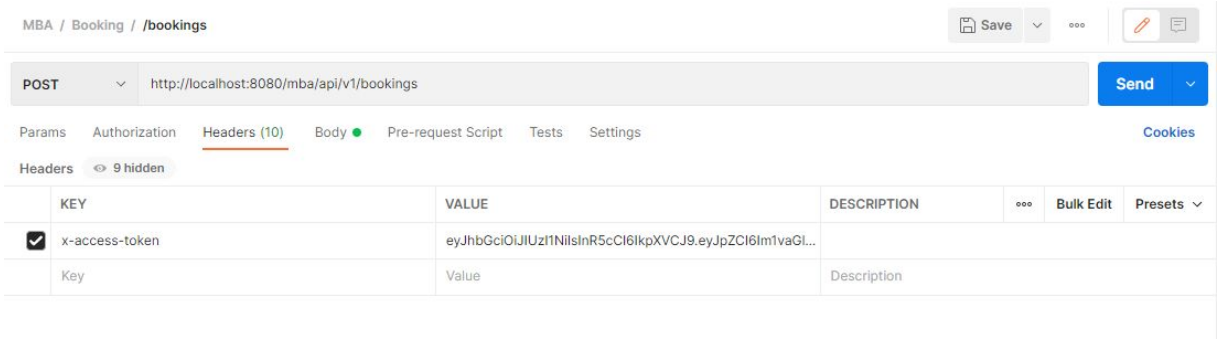
Signup a customer:



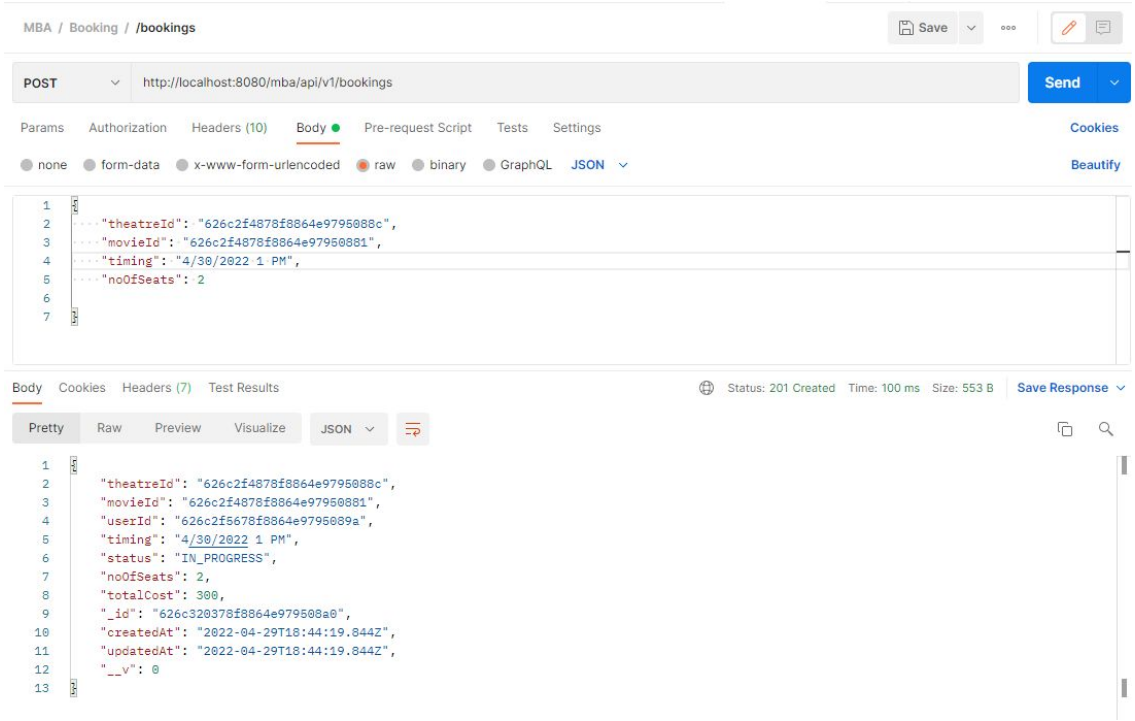
Login as the new registered customer:



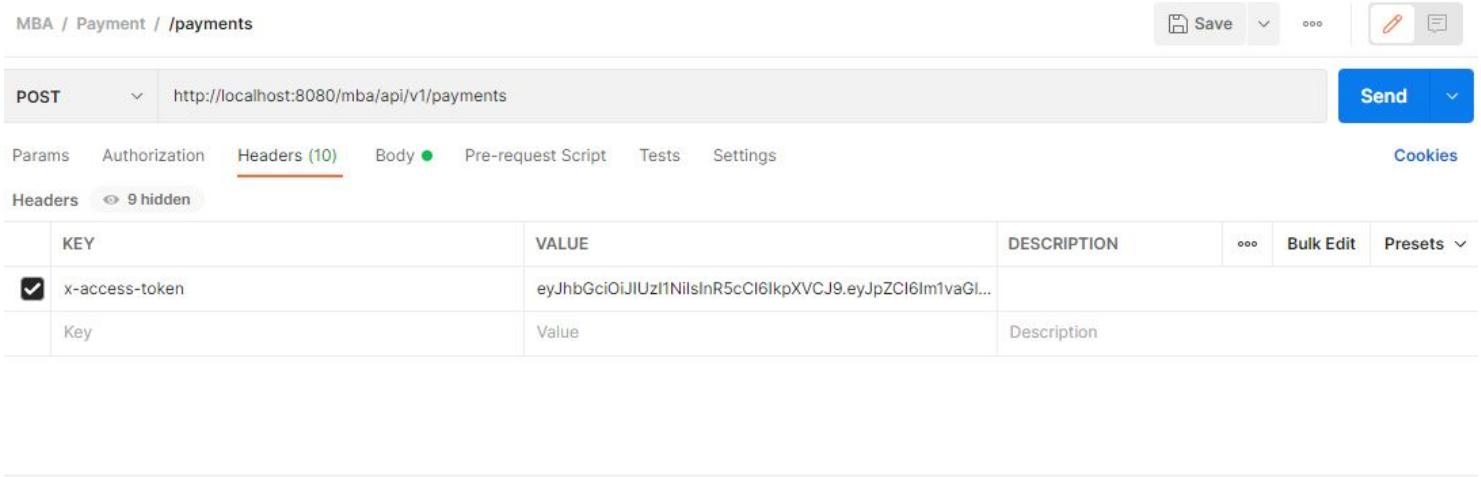
Create a booking id using the new customer token:



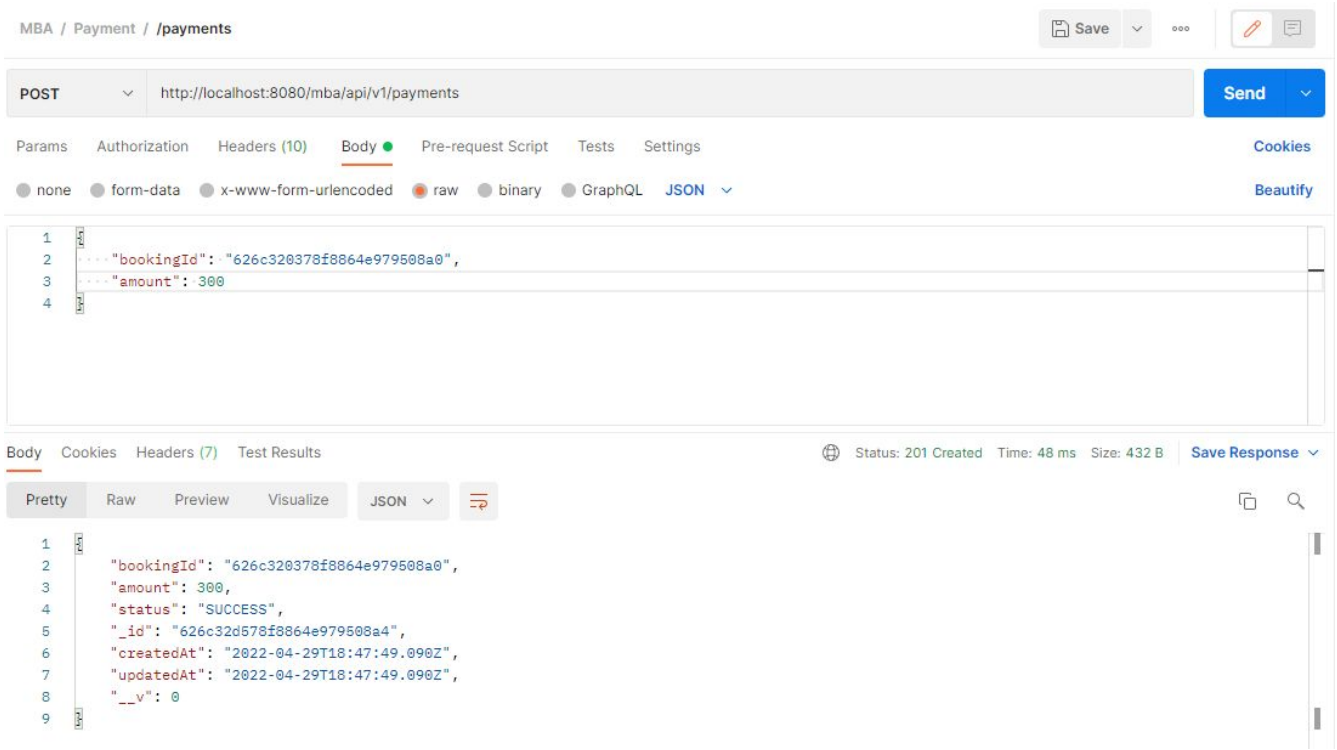
And the required information in request body, such as theatre id, movie id , timing and no. of seats.



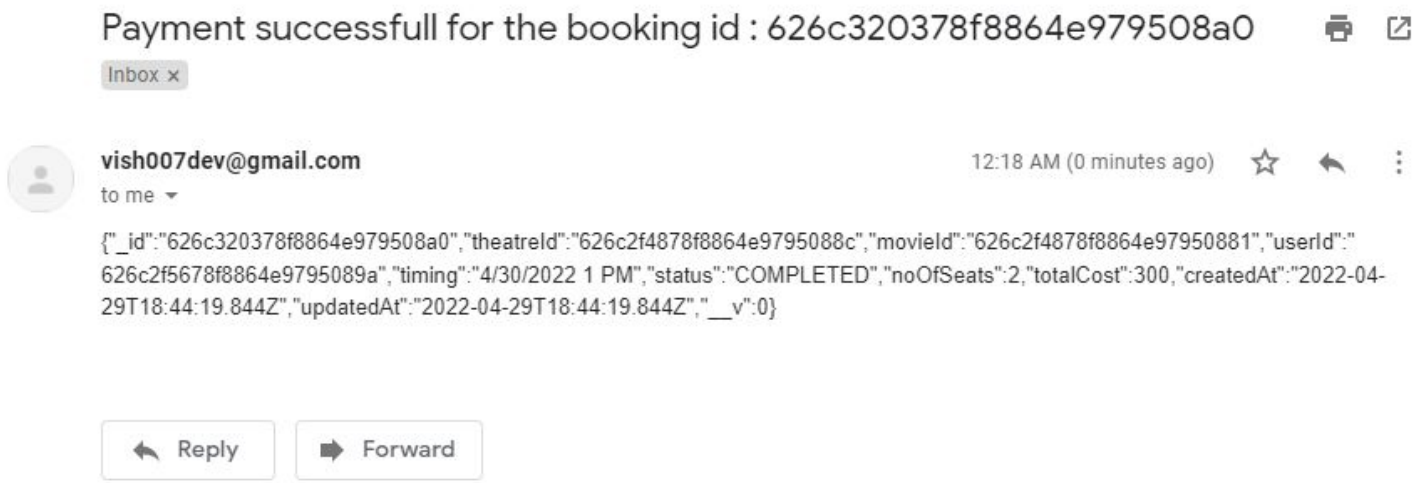
Now, using above booking id and amount we request for new payment creation, by passing the required header.



And the required request body.



Once, payment is successful we will get the email confirmation as well, which is shown below.



Create a middleware to distinguish between an admin and a client:

- So far, we have made notification for customer for payment confirmation.
- Since we also have clients in our application, who should be notified about update on their theatre like, adding a new theatre or updating theatres movie list.
- Amid various validation on some of the routes we need to distinguish between and ADMIN and CLIENT and verify the CLIENT owns the theatre we should have a middleware named “isAdminOrClient” which is shown below.

Code link:

https://github.com/Vishwa07dev/mba_backend/blob/session8/middlewares/authjwt.js

```
isAdminOrClient = async (req, res, next) => {

    const user = await User.findOne({
        userId: req.userId
    })
    if (user && ( (user.userType == constants.userTypes.admin) || user.userType == constants.userTypes.client) ) {


        if(user.userType == constants.userTypes.client){
            //check if the client is the owner of the theatre or not
            const savedTheatre = await Theatre.findOne({ _id: req.params.id });
            if(savedTheatre.ownerId != user._id){
                return res.status(403).send({
                    message: "Client requesting to update the theatre is not the owner!"
                });
            }else{
                next();
            }
        }
        next();

    } else {
        return res.status(403).send({
            message: "Require Admin Role or Client role!"
        });

    }


};
```

Now, let’s add this middleware into required routes, which are updating a theatre and updating movies inside a theatre.

```
const theatreController = require("../controllers/theatre.controller");
const { authJwt, verifyTheatreReqBody } = require("../middlewares");


/**
 * Routes for the movie resource
 */

module.exports = function (app) {
    app.get("/mba/api/v1/theatres", [authJwt.verifyToken], theatreController.getAllTheatres);
    app.get("/mba/api/v1/theatres/:id", [authJwt.verifyToken], theatreController.getTheatre);
    app.post("/mba/api/v1/theatres", [authJwt.verifyToken, authJwt.isAdmin, verifyTheatreReqBody.validateTheatreRequestBody], theatreController.
    createTheatre);
    app.put("/mba/api/v1/theatres/:id", [authJwt.verifyToken, authJwt.isAdminOrClient, verifyTheatreReqBody.validateTheatreRequestBody], theatreController.
    updateTheatre);
    app.delete("/mba/api/v1/theatres/:id", [authJwt.verifyToken, authJwt.isAdmin], theatreController.deleteTheatre);
    app.put("/mba/api/v1/theatres/:id/movies", [authJwt.verifyToken, authJwt.isAdminOrClient], theatreController.addMoviesToATheater);
    app.get("/mba/api/v1/theatres/:theatreId/movies/:movieId", [authJwt.verifyToken], theatreController.checkMovieInsideATheatre);
}
```

Code link:

https://github.com/Vishwa07dev/mba_backend/blob/session8/routes/theatre.routes.js

Notify clients when theatre is created, updated or deleted:

Now, let us again use our mail utility and use it inform CLIENT on various transactions associated with the theatre they own.

Create Theatre method

Here, we have passed theatre id, subject line with theatre id, content of mail as the theatre object, to the theatre owner mail id and the default requester mail id.

```
/**
 * Create  a new Theatre
 */
exports.createTheatre = async (req, res) => {
  try {
    const theatreObject = {
      name: req.body.name,
      city: req.body.city,
      description: req.body.description,
      pinCode: req.body.pinCode,
      ownerId: req.body.ownerId,
    };

    const theatre = await Theatre.create(theatreObject);

    /**
     * Sending email to the owner of the theatre
     */
    const theatreOwner = await User.findOne({ _id: theatre.ownerId });
    sendEmail(
      theatre._id,
      "New theatre created with the theatre id : " + theatre._id,
      JSON.stringify(theatre),
      theatreOwner.email,
      "mba-no-reply@mba.com"
    );

    res.status(201).send(theatre);
  } catch (err) {
    console.error("Some error while saving the user in db", err.message);
    res.status(500).send({
      message: "Some internal error while creating the theatre",
    });
  }
};
```

Update Theatre method

Here, we have passed theatre id, subject line with theatre id, content of mail as the theatre object, to the theatre owner mail id and the default requester mail id.

```
/**
 * Add a movie inside a theatre
 */
exports.addMoviesToATheater = async (req, res) => {

    //validation of tha savedTheatre will be done in the later section as middleware
    const savedTheatre = await Theatre.findOne({ _id: req.params.id });

    //Validation of these movie ids will be done in the later section
    movieIds = req.body.movieIds;

    //Add movieIds to the theatres
    if (req.body.insert) {
        movieIds.forEach(movieId => {
            savedTheatre.movies.push(movieId);
        });
    } else {
        //remove these movies from the theatres
        savedMovieIds = savedTheatre.movies;

        movieIds.forEach(movieId => {
            savedMovieIds = savedMovieIds.filter(smi => smi != movieId);
        });
        savedTheatre.movies = savedMovieIds;
    }


    await savedTheatre.save(); //save in the database

    /**
     * Sending email to the owner of the theatre
     */
     const theatreOwner = await User.findOne({_id : savedTheatre.ownerId});
     sendEmail(savedTheatre._id, "Theatre deleted with the theatre id : " + savedTheatre._id, "Theatre deleted", theatreOwner.email, "mba-no-reply@mba.
     com");

    res.status(200).send(savedTheatre);

}
```

Delete Theatre method

Here, we have passed theatre id, subject line with theatre id, content of mail as the theatre object, to the theatre owner mail id and the default requestor mail id.

```
/**
 * Delete a theatres
 */
exports.deleteTheatre = async (req, res) => {

    const savedTheatre = await Theatre.findOne({_id: req.params.id});
    await Theatre.deleteOne({
        _id: req.params.id
    });

    /**
     * Sending email to the owner of the theatre
     */
     const theatreOwner = await User.findOne({_id : savedTheatre.ownerId});
     sendEmail(savedTheatre._id, "Theatre deleted with the theatre id : " + savedTheatre._id, "Theatre deleted", theatreOwner.email, "mba-no-reply@mba.
     com");



    res.status(200).send({
        message: "Successfully deleted theatre with id [ " + req.params.id + " ]"
    });


}
```

Add Movies To A Theatre method

Here, we have passed theatre id, subject line with theatre id, content of mail as the theatre object, to the theatre owner mail id and the default requester mail id.

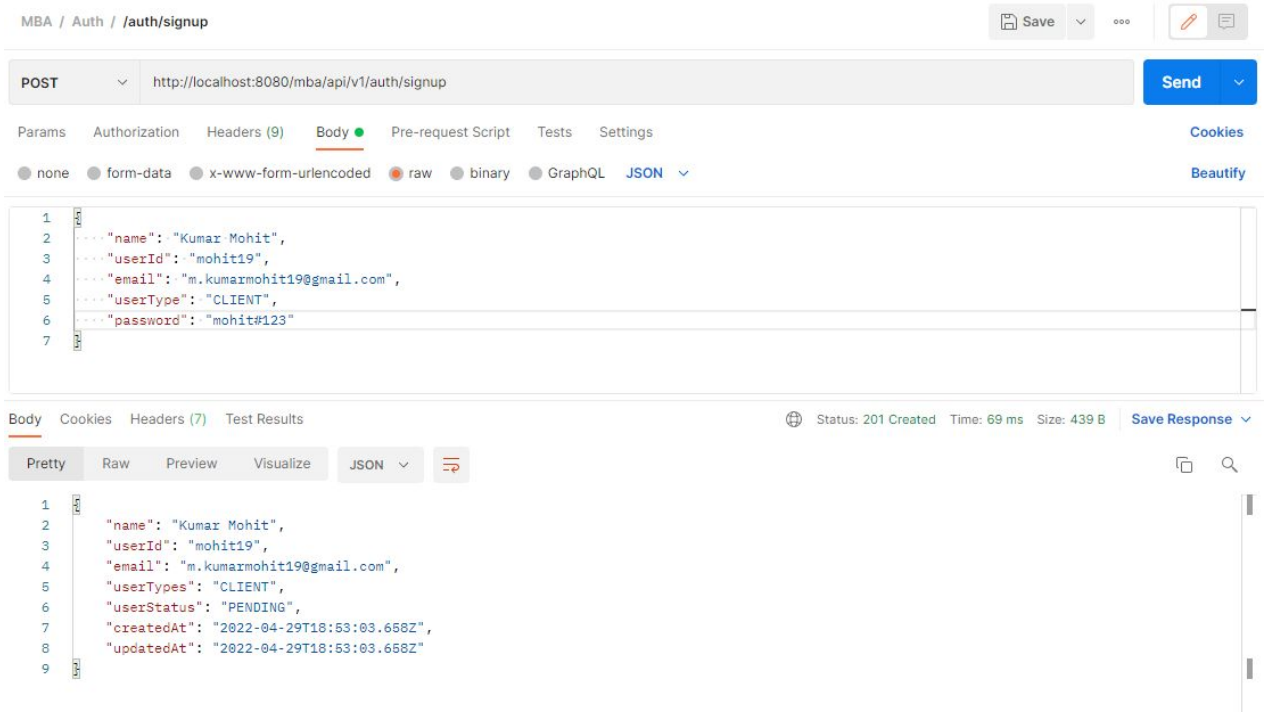
```
/**
 * Add a movie inside a theatre
 */
exports.addMoviesToATheater = async (req, res) => {

    //validation of tha savedTheatre will be done in the later section as middleware
    const savedTheatre = await Theatre.findOne({ _id: req.params.id });

    //Validation of these movie ids will be done in the later section
    movieIds = req.body.movieIds;

    //Add movieIds to the theatres
    if (req.body.insert) {
        movieIds.forEach(movieId => {
            savedTheatre.movies.push(movieId);
        });
    } else {
        //remove these movies from the theatres
        savedMovieIds = savedTheatre.movies;

        movieIds.forEach(movieId => {
            savedMovieIds = savedMovieIds.filter(smi => smi != movieId);
        });
        savedTheatre.movies = savedMovieIds;
    }


    await savedTheatre.save(); //save in the database

    /**
     * Sending email to the owner of the theatre
     */
     const theatreOwner = await User.findOne({_id : savedTheatre.ownerId});
     sendEmail(savedTheatre._id, "Theatre deleted with the theatre id : " + savedTheatre._id, "Theatre deleted", theatreOwner.email, "mba-no-reply@mba.
     com");

    res.status(200).send(savedTheatre);

}
```

Code link:
https://github.com/Vishwa07dev/mba_backend/blob/session8/controll
ers/theatre.controller.js

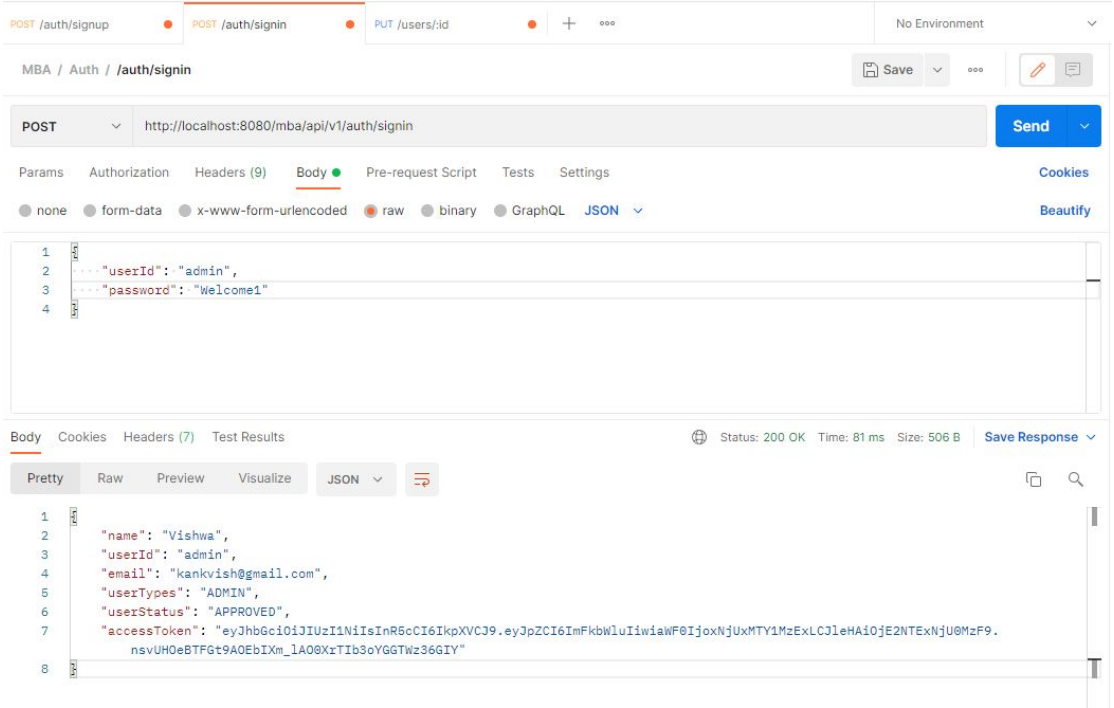
Testing the code:

Let us now test and code change and verify if the notification is being sent or not.

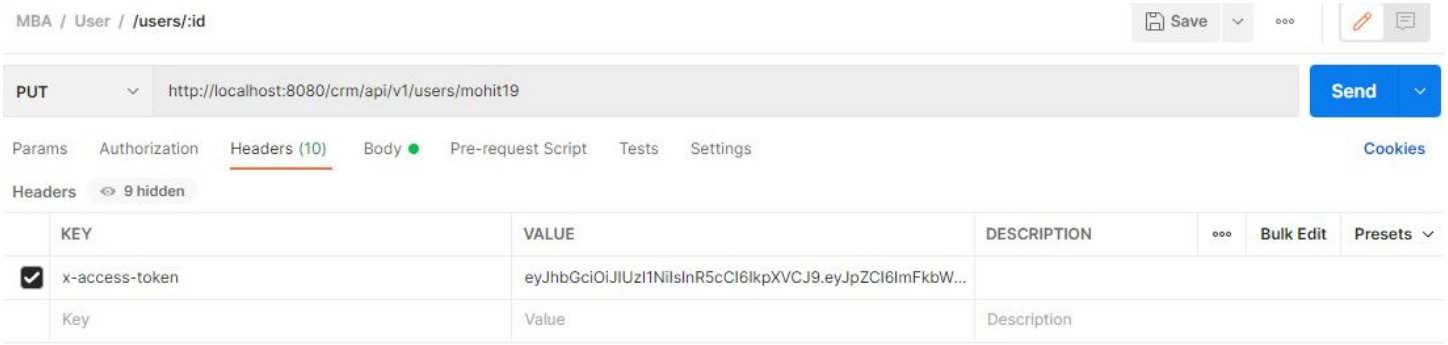
We will continue the testing after approving a new CLIENT which is already discussed before.



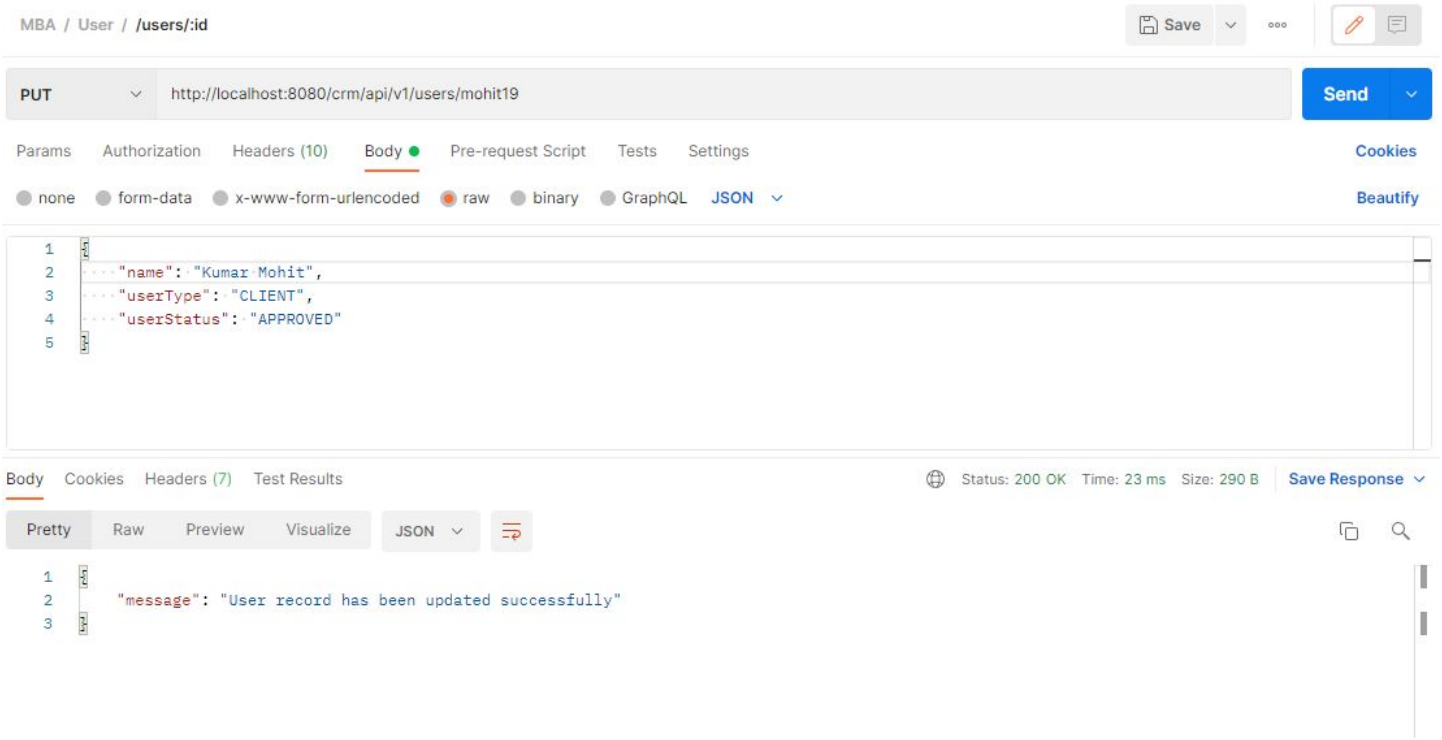
Then we will login with ADMIN credentials to approve the new CLIENT.



We will use the ADMIN level token and create a request to /users/:userId to approve the new CLIENT registration.

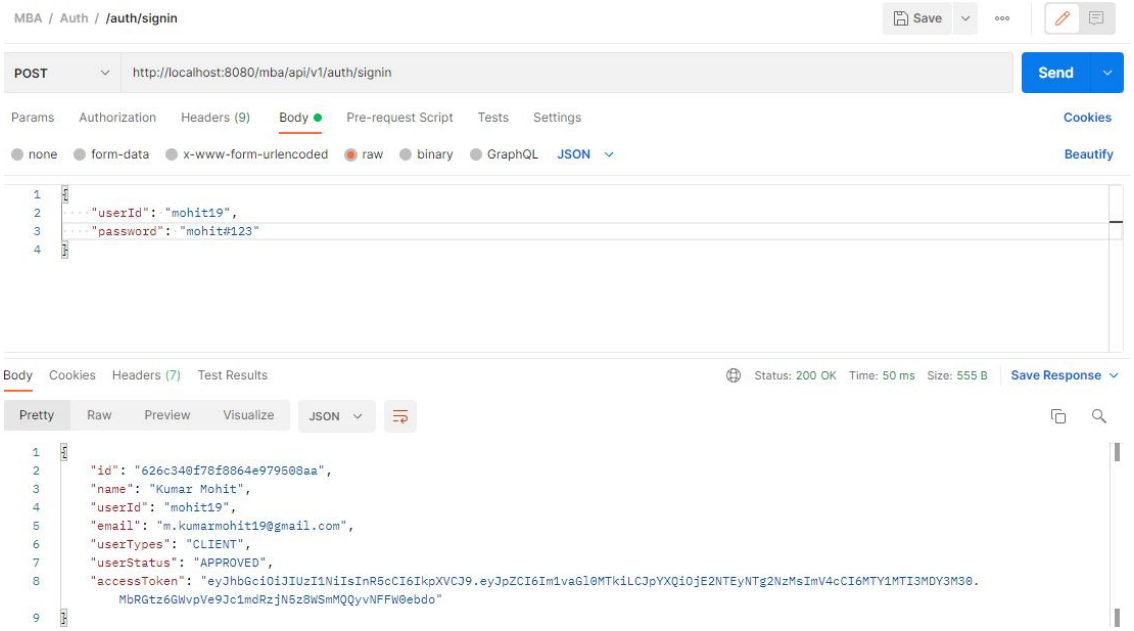


With the proper request body as shown below:

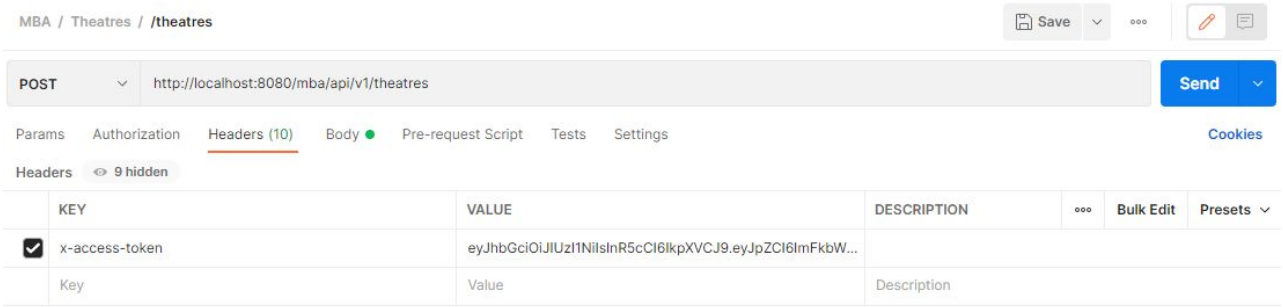


Now, let us create a theatre and use this user id in the owner attribute.

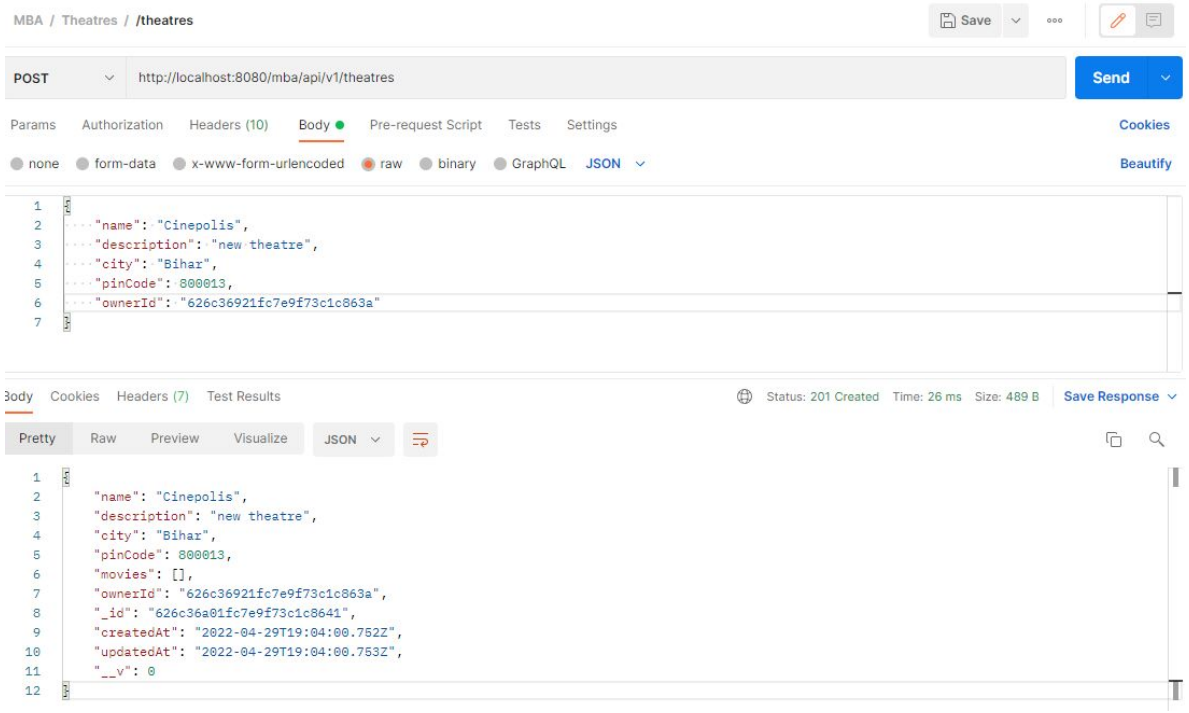
First, we will get the owner id by singing in.



Next, will call POST request for /theatres with ADMIN level token.

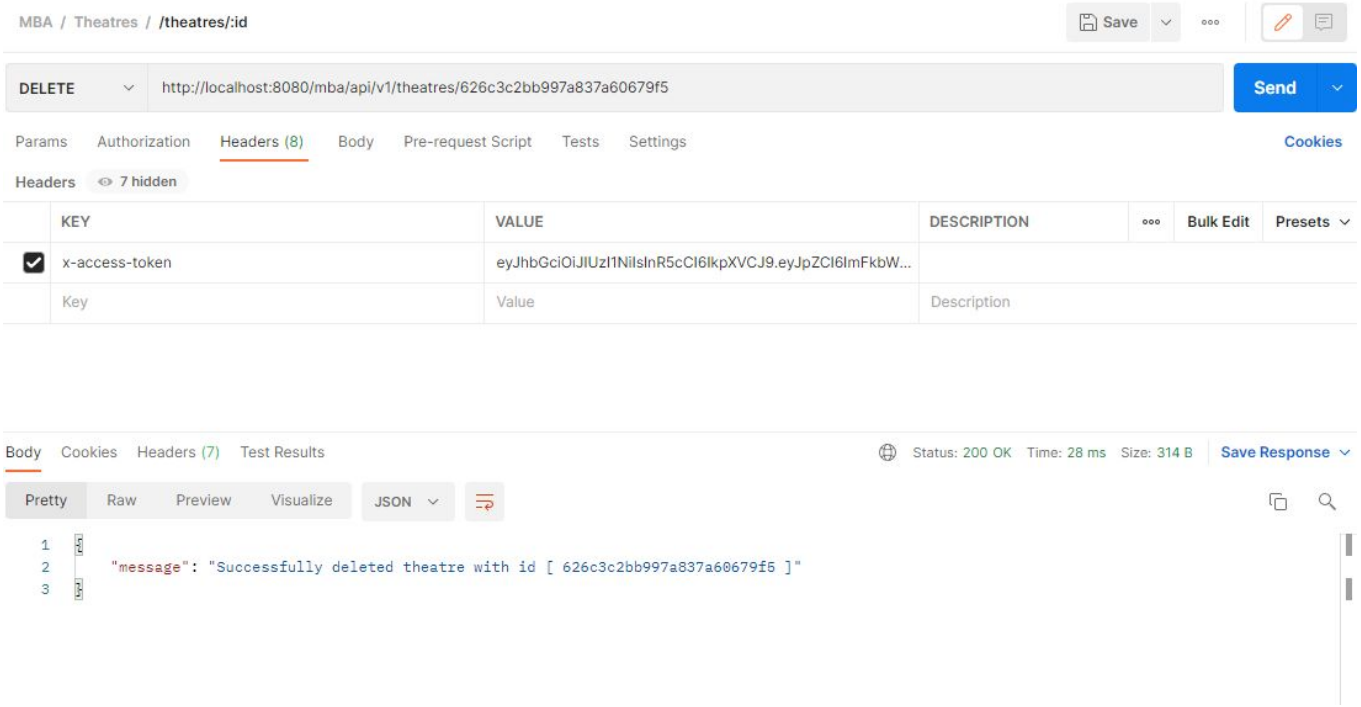


In the request body we will pass the new CLIENT user id as the owner id and other details of the theatre.

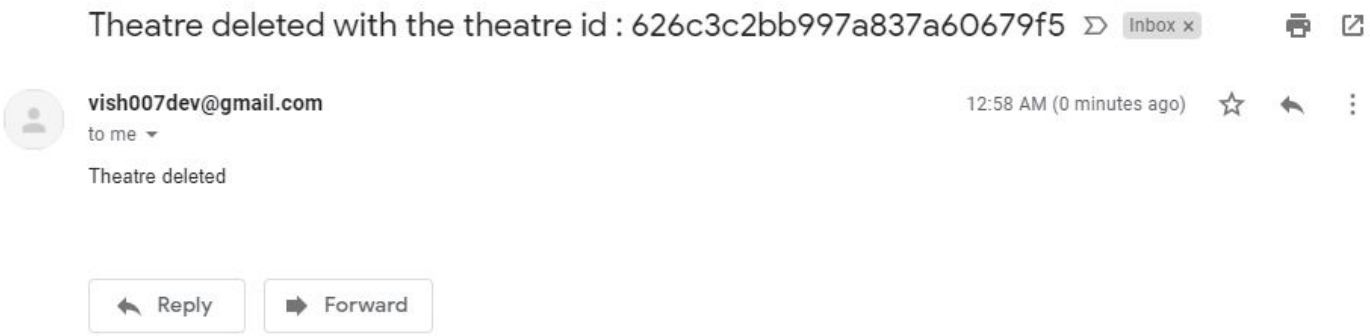


Once the request is successful the theatre owner will receive the email notification as show below.

Now let us test the DELETE /theatre/:id route for email notification using ADMIN token.



Once, the request is successful the theatre owner will again receive a notification about the same as shown below.



MCQ's:

1. Which package is used consume another REST API from our application?

A. mongoose
B. node-rest-client
C. express
D. Dotenv

2. In which scenarios a customer level user receives notification email?

A. When new theatre is added
B. When a movie is removed from a theatre
C. When payment is successful
D. None of the above

3. How many ways can we use notification service?

A. SMS notification
B. Email Notification
C. Push Notification
D. All of the above

4. Which package is used to send mails from our application?

A. node-rest-client
B. node-sms-send
C. nodemailer
D. None of the above

5. Which package is used to schedule a job in a node application?

A. nodemailer
B. node-cron
C. node-rest-client
D. express

Assignment Questions:

1. Send mail to the user when they sign up into the application for the first time.
2. Update the notification mail body from JSON to string format with proper alignment and design wherever required.

In the Upcoming Class:

- In brief about Git and GitHub
- What is Heroku
- Create a production DB on MongoDB Atlas
- Steps to deploy our app on Heroku

Thank You!