Building a Notification Service Application

Relevel
by Unacademy

Topics/Concepts to be covered in today's class:

- API for raising the notification request
- API to get the result of the notification request
- Scheduled job to regularly check for any new request and then send email notifications to every one listed

Intro to Notification service

We will add a Notification feature to our CRM app in this class. At a certain point of time, while using various applications, you may have faced some issue which was needed to be reported to the application provider; it could be an issue, suppose you are using a Banking app, and you are paying your credit card bill, but there is some issue with the transaction where the amount got debited from your account. Still, it's not getting updated in the app. In this case, you raise a transaction dispute with them through their portal where you describe the issue and maybe share transaction details as screenshots etc.; after the ticket is raised on your behalf, you will get an email with the ticketId, issue details status of your request etc. on your registered mail id. This is how different applications use the notification service for customer support, and we will be using it for similar purposes.

Problem Statement?

Create a Notification service, which can take the notification requests from it's customers and then asynchronously send the notification to all the recipients of that message.

Feature 1 :

Customer can request for sending the email notification to all the recipients

Customer needs to provide the following information:

- Subject
- Content of the email
- Recipient email Ids
- Requester
- ticketId (for which the notification has to be sent)

Solution :

- Exposing RESTful API for customers to raise this request
- Nature of the request should be asynchronous.
- User should be provided with the requestId, which can be used to retrieve the status of email request later

Sample REST Endpoint :

We will use the POST request method with the request body having all the above required fields which will return the response after successful hit to the API.

```
POST /notifiServ/api/v1/notifications
Headers :
 Content-Type:application/json
Sample request body :
{
    "subject" : "Ticket with id [ id ] has been created",
    "content" : "Ticket has been created. User is having some issues",
    "recepientEmails" : "xyz@gmail.com, pqr@linkeddin.com",
    "requester" : "Vishwa Mohan",
    "ticketId" : "qqwe2314355521"
}

Sample response body :
{
    "requestId": "qqwe2314355521",
    "status": "Accepted Request"
}
```

Feature 2 :

Customer can check the status of his last notifications request

Customer needs to provide the following information :

- requestId

Solution :

- Exposing RESTful API for the customers to know the status of his last request

Sample REST Endpoint :

```
GET /notifiServ/api/v1/notifications/621893416ff4a6435445935b
Headers :
 Content-Type:application/json


Sample response body :
{
    "requestId": "621893416ff4a6435445935b",
    "subject": "Ticket with id [ id ] has been created",
    "content": "Ticket has been created. User is having some issues",
    "recepientEmails": [
        "xyz@gmail.com, xyz@linkeddin.com"
    ],
    "sentStatus": "SENT"
}
```

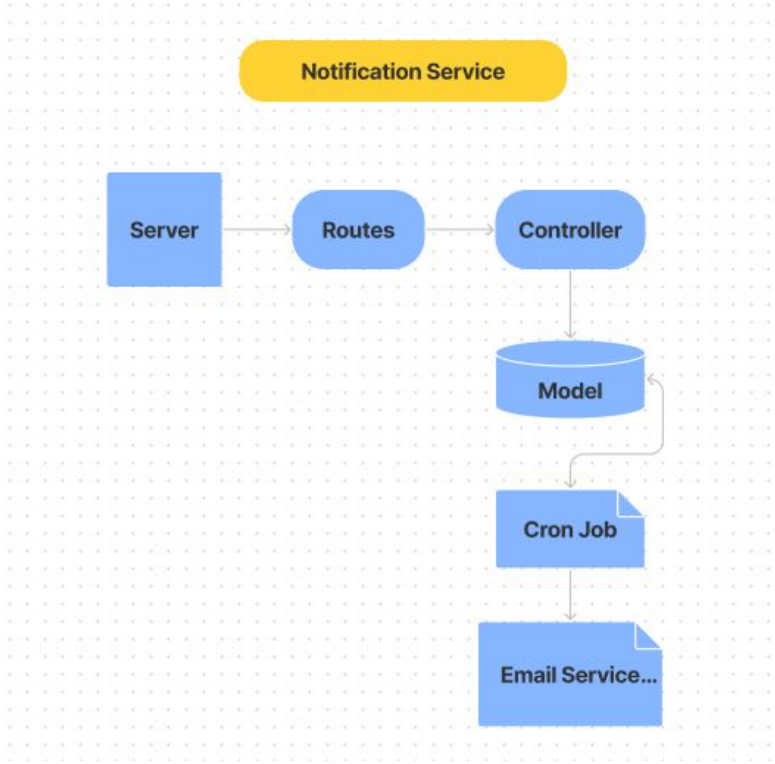
Feature 3 :

Run a continuously running background job, triggered every 30 seconds, check for any new request for notification and send the notification message

Solution :

- Write a schedule job to regularly check for any new request and send the notification accordingly

System Architecture



- **Server** - We will create a server.js file that will connect to our MongoDB database and set up a port for accessing the APIs via URLs.
- **Routes** - We will set up and map the routes with API methods inside route.js.
- **Controller** - We will write our logic for building the POST and GET APIs and describe the request/response. We will also use the model/database here to create the entries and find the entries from the database collection.
- **Model** - This will have the database schema implementation.
- **Cron Job** - As we want to schedule our email notifications, we will use node-cron module/library as a utility that will trigger the email notification after 30 seconds. Cron is basically a tool that helps us to run a scheduled job i.e. it will perform the function for us at desired interval/scheduled time, we will discuss this in detail while implementing the actual job.
- **Email Service** - Finally, we will use the nodemailer module for smtp setup and authentication of the email id through which we will send the emails.

Application Code:

server.js

```
require('./crons/cron');
const serverConfig = require('./configs/server.config');
const dbConfig = require('./configs/db.config');
const mongoose = require('mongoose');
const express = require('express');
const bodyParser = require("body-parser");
const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended :true}));
/**
 * DB Connection initialization
 */
```

```
mongoose.connect(dbConfig.DB_URL, () => {
    console.log("connected to Mongo DB ")

}, err => {
    console.log("Error :", err.mssage)
}
);
//Stiching the routes
require('./routes/ticketNotification.route')(app);

app.listen(serverConfig.PORT, () => {
    console.log(`Application started on the port num :
${serverConfig.PORT}`);
})
```

- In server.js, we will import all the required configurations and packages.
- Here we are importing server and database configurations from server.config.js and db.config.js that we will see below.
- We are also importing cron.js used for job scheduling and sending the notification, which we will build later.
- Apart from that, we are importing.
    - Express - Backend framework for NodeJS used to build APIs and Web Apps.
    - Mongoose - MongoDB utility for creation of schema and performing CRUD operations.
    - Body parser - A middleware used to parse the request and response for REST APIs.
- We are using mongoose. Connect method to connect to the database URL and print "connected to MongoDB " in case of successful connection or "Error :" incase of error.
- Then we upload the routes and listen on the configured port. Once we get a response, we print “Application started on the port num: 8080” as we have used it as our application start point.
-

Configs

- We will write all the configuration details under this folder for server and database.

1. server.config.js

```
if (process.env.NODE_ENV !== 'production') {
        require('dotenv').config();
}
module.exports = {
    PORT: process.env.PORT
}
```

- Here we check if NODE_ENV (i.e. node environment) is not set to production we load the .env file using dotenv package and export the PORT as 8080.

2. db.config.js

```
module.exports = {
        DB_NAME: "crm_db",
        DB_URL: "mongodb://localhost/notification_db"
}
```

- We simply set the database name and URL and export it.

ticketNotification.route.js

```
const notificationController = require('../controllers/ticketNotification.controller');

module.exports = function (app) {

    app.post("/notifiServ/api/v1/notifications", notificationController.acceptNotificationRequest);

    app.get("/notifiServ/api/v1/notifications/:id", notificationController.getNotificationStatus);}
```

- We are importing the controller.js as we need the APIs written in them to be set with our route URLs.
- We set the route paths for POST and GET methods and map it with notificationController.acceptNotificationRequest and notificationController.getNotificationStatus respectively.

ticketNotification.controller.js

- acceptNotificationRequest

```
Const TicketNotificationModel = require("../models/ticketNotification.model");

exports.acceptNotificationRequest = async (req, res) =>{

    const notificationObject = {

            subject : req.body.subject,

            content : req.body.content,

            recepientEmails : req.body.recepientEmails,

            reuester : req.body.reuester,

            ticketId : req.body.ticketId

    };
```

```
    try{
    const notification = await TicketNotificationModel.create(notificationObject);
    res.status(201).send({
        requestId : notification.ticketId,
        status : "Accepted Request"
    })
    }catch(err){
        console.log(`Error while accepting a notification request : ${err.message}`);
    }
}
```

- Here we import the ticketNotification.model.js as we are using it for creating an entry in the database and fetching/getting entries from the database.
- We create an acceptNotificationRequest() method that accepts requests and returns responses.
- A notificationObject is created that has all the required email details and is used with TicketNotificationModel.create() method to push it into the database collection.
- For this request on http://localhost:8080/notifiServ/api/v1/notifications

```
{
    "subject" : "Ticket with id [ id ] has been created",
    "content" : "Ticket has been created. User is having some issues",
    "recepientEmails" : "xyz@gmail.com, pqr@linkeddin.com",
    "requester" : "Vishwa Mohan",
    "ticketId" : "qqwe2314355525"
}
```

- we receive a response as below.

```
{
    "requestId": "qqwe2314355525",
    "status": "Accepted Request"
}
```

- getNotificationStatus

```
exports.getNotificationStatus = async (req, res) =>{
    const reqId = req.params.id;
    try{
    const notification = await TicketNotificationModel.findOne({
        ticketId : reqId
    })

    res.status(200).send({
        requestId : notification.ticketId,
        subject : notification.subject,
        content : notification.content,
        recepientEmails : notification.recepientEmails,
        sentStatus : notification.sentStatus
    })
    }catch(err){
        console.log(`Error while fetching a notification request : ${err.message}`);
    }
}
```

- Similarly we create a getNotificationStatus() method that accepts requests and returns responses.
- This basically finds the record in the database with the ticketId that we pass and returns the response as email contents and sentStatus as SENT or UN_SENT.
- For this request on http://localhost:8080/notifiServ/api/v1/notifications/qqwe2314355525
- We get below response


```
{
    "requestId": "qqwe2314355525",
    "subject": "Ticket with id [ id ] has been created",
    "content": "Ticket has been created. User is having some issues",
    "recepientEmails": [
        "xyz@gmail.com, pqr@linkeddin.com"
    ],
    "sentStatus": "SENT"
}
```

1. ticketNotification.model.js

```
const mongoose = require("mongoose");

const ticketNotificationSchema = new mongoose.Schema({
    subject: {
        type: String,
        required: true,
    },
    ticketId :{
        type : String,
        required :true
    },
    content: {
        type: String,
        required: true,
    },
    recepientEmails: {
        type: [String],
        required: true
    },
```

```
    sentStatus :{
        type: String,
        required: true,
        default: "UN_SENT"

    },
    requester :{
        type: String

    },
    createdAt: {
        // I want to set default new date
        type: Date,
        immutable: true,  // This will ensure the createdAt column is never updated but once
in the start
        default: () => {
            return Date.now();
        }
    },
    updatedAt: {
```

```
        type: Date,
        default: () => {
            return Date.now();
        }
    }
})

module.exports = mongoose.model("TicketNotification", ticketNotificationSchema);
```

- This is similar to what we did in the previous classes when we learnt about mongoose and database schema.
- We have imported mongoose and created a new schema along with all the required email fields and createdAt and updatedAt fields that will keep a track of when the document is created and updated.

1. **cron.js**

   Before understanding the code for cron job scheduler let us see how it works.

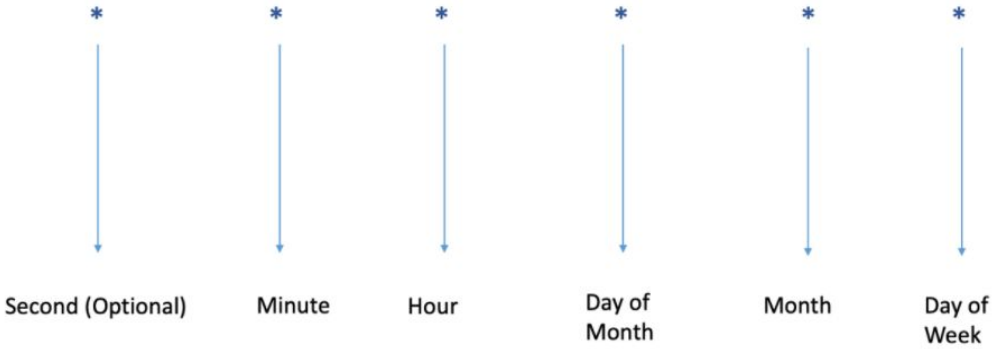
- Cron is basically a tool that helps us to run a scheduled job i.e. it will perform the function for us at desired interval/scheduled time.
- Let’s install the cron package using below

  npm i node-cron

- Let us understand the cron method now

```
cron.schedule('* * * * * *', () => {
        console.log("Runs after every second")
});
```

- As we can see there are 6 stars in the cron schedule function.
- In the below image we can see the meaning of each star.



- The second argument of the function is a callback function where you can write business logic for your job.
- As we want to schedule it for every 30 seconds we write it as
    - `cron.schedule('*/30  * * * * *', async ()=>{ console.log("Runs after every 30 seconds")});`
- This is the syntax “*/number of seconds or minutes etc”
- Now let’s get back to our actual code.

```
Const cron = require('node-cron');
const TicketNotificationModel = require("../models/ticketNotification.model");
const EmailTransporter = require("../notifier/emailService");

cron.schedule('*/30  * * * * *', async ()=>{
    /**
     * Logic inside this to search the db every 30 seconds and send the emails for any new request
     */
    const notifications = await TicketNotificationModel.find({
        sentStatus : "UN_SENT"
    });
    console.log(notifications.length);
```

```
notifications.forEach(notification => {

        const mailData = {
            from: 'crm-notification-service@gmail.com',
            to: notification.recepientEmails,
            subject: notification.subject,
            text: notification.content
        };
        console.log(mailData);
        EmailTransporter.sendMail(mailData, async function (err, info) {
            if (err)
                console.log(err.message);
            else
                console.log(info);
                //Update the DB
                const savedNotification  = await TicketNotificationModel.findOne({ _id : notification._id});
                savedNotification.sentStatus = "SENT";
                await savedNotification.save();
        });
    });
})
```

- We import the node-cron package, ticketNotification.model.js and emailService.js that is our service for sending the email which we will see after implementing the cron function.
- Now we schedule the cron job for 30 seconds as seen above using cron.schedule() and the second argument will have the logic to search the database for the scheduled job i.e. every 30 seconds and send the emails for any new request.
- We check in the database for entries with sentStatus : "UN_SENT".
- Then we display the number of such entries.
- Then for each of these entries we construct an object mailData with the email details and call sendMail() on the EmailTransporter object.
- Finally we update the status for this as sent.

1. emailService.js

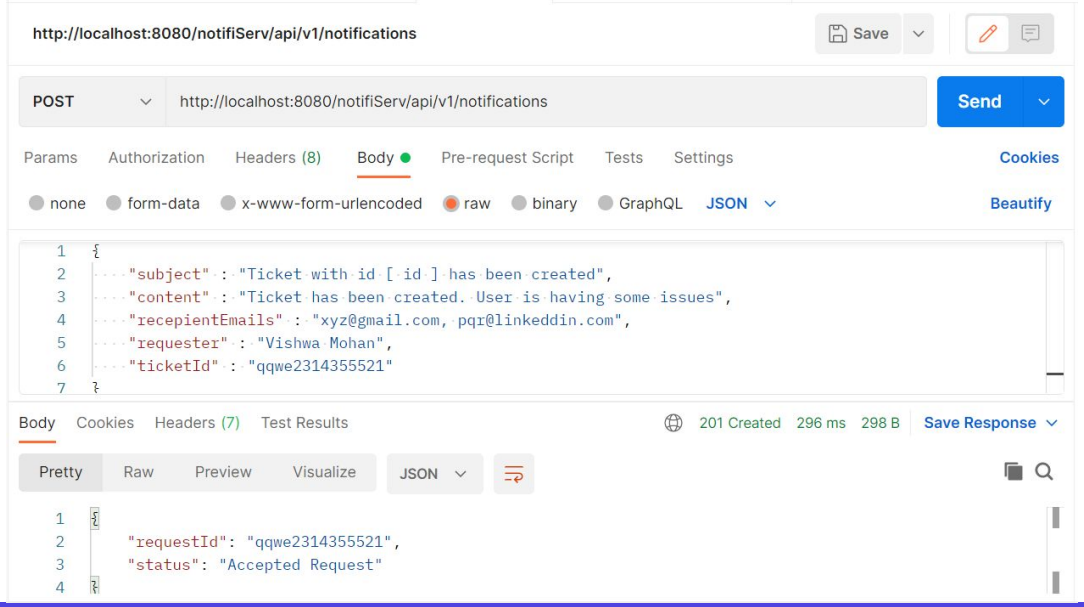
```
const nodemailer = require('nodemailer');

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,
});
```

- This is the emailService.js where we import the nodemailer package for authenticating the email id and sending the mail.
- We install nodemailer using **npm i nodemailer**
- Now we need to create a connection for sending the email, so we use Simple Mail Transfer Protocol (SMTP) which is a standard internet communication protocol for electronic mail transmission as a protocol with host which could be localhost or an email service provider like gmail, outlook etc.
- So we use createTransport() method which is in fact creating a tunnel for making a connection to the desired email service provider.
- Now we call createTransport() which will have a
    - port: is the port to connect to (which is set to 587 as default if is secure is false or 465 if true)
    - host: is the hostname or IP address to connect to (defaults to ‘localhost’) or mail id service provider as gmail or outlook etc.
    - auth: defines authentication data which checks the email credentials.
- On successful authentication an email is triggered using the sendmail method on an object of emailService which we have done in cron.js.

Request and response for APIs in Postman

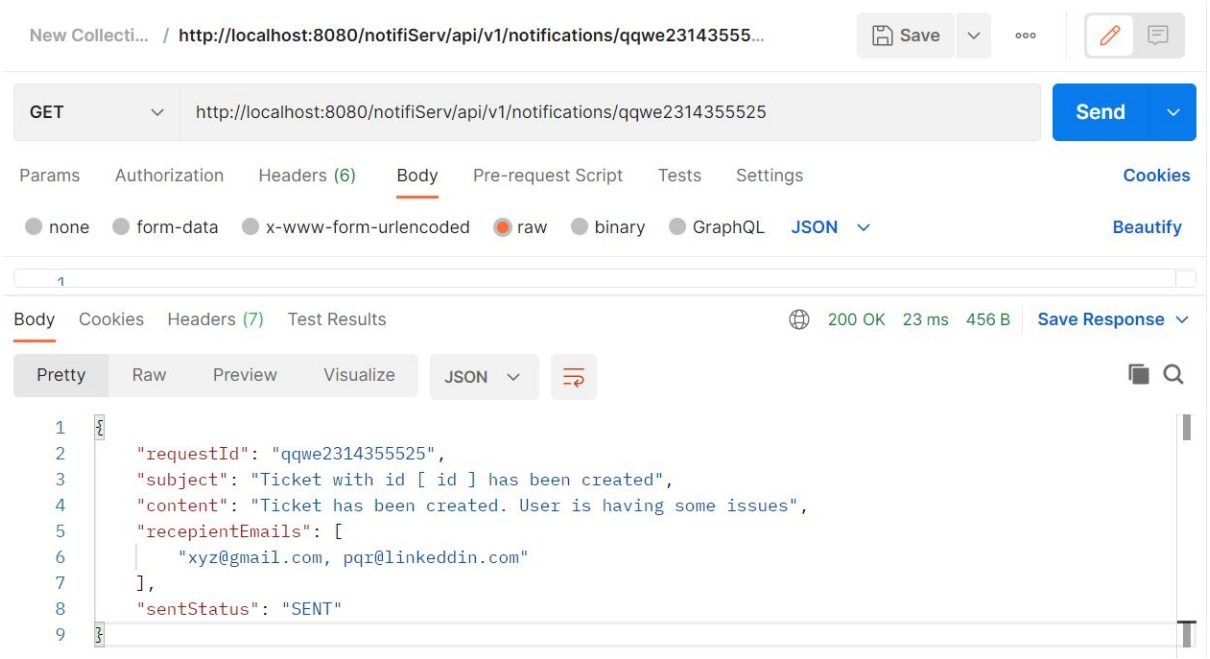
acceptNotificationRequest

http://localhost:8080/notifiServ/api/v1/notifications



getNotificationStatus:

http://localhost:8080/notifiServ/api/v1/notifications/qqwe2314355525



MCQ Questions

1) **Which REST method should be used for fetching the response with email sentStatus based on requestID in postman?**

   A) GET [Correct Answer]

   B) POST

   C) PUT

   D) None

2) What should be included as maildata for sendMail method of nodemailer module?

A) from

B) to

C) subject

D) All of the above [Correct Answer]

3) Which functionality must be included as part for notification service?

A.) Scheduling the Notification job

B.) Creating route for APIs

C.) Authenticate User email credentials

D.) All of the above [Correct Answer]

4) What is the syntax to schedule a cron job for every 15 mins

A.) cron.schedule('*/15 * * * * *', () => {});

B.) cron.schedule('0 */15 * * * *', () => {}); [Correct Answer]

C.) cron.schedule('* * */15 * * *', () => {});

D.) None

5) Which is the correct method for parsing request and response of APIs?

A.) JSON.parse()

B.) JSON.stringify()

C.) bodyParser.json() [Correct Answer]

D.) None

Practice/HW

1. Implement a CRON job to send an email notification 15 mins before the start of a meeting at 8 pm every day, i.e. the notification should be sent at 7.45 pm.

1. Write a controller for GET API to fetch the response of email details for the above cron job.

THANK YOU