Parallel Jobs + Mailer

Introduction::Mailers

- We will be studying how to send email notifications to the user whenever specific events happen { someone likes your post, someone replied to your post, etc }.
- We will be also looking at the optimization of those emails.
- Suppose there is a user who has performed some actions or makes some requests. Let's suppose the website has a provision that when a user makes a new comment, it should send an email to the user on whose post the comment was made.
- The server has some logic written on it. So it requests the MAIL server like { GMAIL, Yahoo, SendGrid, Zoho} to send an email. It will send that email to the remote user to whom we want to send it, in return, it will give a response whether the request of sending mail is successful or not.
- MAIL servers including { GMAIL, Yahoo, SendGrid, Zoho} are some of the dedicated services that help in sending emails in mass or bulk.
- In this lecture, we will be using GMAIL.
- We request the server to send mails using the SMTP {Simple Mail Transfer Protocol } protocol.
- It will define how both the mail server and other servers will communicate.
- Since we will be utilizing an API of the mailing server, we will need to establish some sort of identity.

Setting Up Nodemailer

- We will be installing the module **Nodemailer** and setting up the config for it.
- We will be setting up the file that will have the function to send mail. It will have predefined content.
- We want to send mails with some HTML content that will have different formatting, we will be sending those emails using templates.
- To install nodemailer use the command { **npm install nodemaile**r } in the terminal.
- Nodemailer has two things- transporters and template { transporter defines the configuration using which we will be sending emails }.
- We will create a separate folder in **views** wherein we define the mail template to be used.
- Inside the **config** folder, we need to create a new file { **config.js** }.
- We need to import nodemailer inside the { **config.js** } file.
- We need to define the transporter that will be attached to the nodemailer.
- For the rendering of the template we need to require EJS in the file

{ **config.js** }

```
const nodemailer = require("nodemailer");
const ejs = require('ejs');
const path = require('path')


let transporter = nodemailer.createTransport({
    service: 'gmail',
    host: 'smtp.gmail.com',
    port: 587,
    secure: false,
    auth: {
        user: 'alchemy.cn18',
        pass: 'codingninjas'
    }
});


let renderTemplate = (data, relativePath) => {
    let mailHTML;
    ejs.renderFile(
        path.join(__dirname, '../views/mailers', relativePath),
        data,
        function(err, template){
         if (err){console.log('error in rendering template'); return}

         mailHTML = template;
        }
    )

    return mailHTML;
}

module.exports = {
    transporter: transporter,
    renderTemplate: renderTemplate
}
```

EXTRA:
You can check out the link below to understand more about nodemailer -
https://nodemailer.com/about/

Sending Our First Email via SMTP

- We need a file that will help in sending the emails repeatedly to different users with the same text. We need a template for the same.

- The triggering point of mail is whenever one user comments on someone's posts, let the mail come to the user itself who is commenting in the initial stage. Later, send that mail to the user who has posted the post.
- We will create a new folder **{ mailers }**, and inside it a new file **{ comments_mailer.js }.**
- We need to import nodemailer inside the **{ comments_mailer.js }** file.
- We need to create a function that will create the mail.

{ comments_mailer.js }

```
const nodeMailer = require('../config/nodemailer');


// this is another way of exporting a method
exports.newComment = (comment) => {
    console.log('inside newComment mailer', comment);

    nodeMailer.transporter.sendMail({
       from: 'arpan@codingninjas.in',
       to: comment.user.email,
       subject: "New Comment Published!",
       html: '<h1>Yup, your comment is now published!</h1>'
    }, (err, info) => {
        if (err){
            console.log('Error in sending mail', err);
            return;
        }

        console.log('Message sent', info);
        return;
    });
}
```

- We need to call this mailer that we have created inside the **{ comments-controller.js } ** file.

{ comments-controller.js }

```
module.exports.destroy = async function(req, res){

    try{
        let comment = await Comment.findById(req.params.id);

        if (comment.user == req.user.id){

            let postId = comment.post;

            comment.remove();

            let post = Post.findByIdAndUpdate(postId, { $pull: {comments: req.params.id}});

            // send the comment id which was deleted back to the views
            if (req.xhr){
                return res.status(200).json({
                    data: {
                        comment_id: req.params.id
                    },
                    message: "Post deleted"
                });
            }
            req.flash('success', 'Comment deleted!');

            return res.redirect('back');
        }else{
            req.flash('error', 'Unauthorized');
            return res.redirect('back');
        }
    }catch(err){
        req.flash('error', err);
        return;
    }

}
```

Send HTML Template Emails

- We need to set up the template so that we can send HTML emails.
- We will create a mail file that will be used for sending mail to the person.
- Inside the views/mailers, we need to create a folder for **comments.**
- Inside the comments folder we need to create a file { new_commnents.ejs}.
- We will be mentioning inside the { **comments.mailer.js }** that we are going to use { **new_commnents.ejs}** file inside the comments folder as the template.

{ comments.mailer.js }

```
// this is another way of exporting a method
exports.newComment = (comment) => {
    let htmlString = nodeMailer.renderTemplate({comment: comment}, '/comments/new_comment.ejs');

    nodeMailer.transporter.sendMail({
       from: 'arpan@codingninjas.in',
       to: comment.user.email,
       subject: "New Comment Published!",
       html: htmlString
    }, (err, info) => {
        if (err){
            console.log('Error in sending mail', err);
            return;
        }

        console.log('Message sent', info);
        return;
    });
}
```

{ new_commnents.ejs}

```
<div>

    <p>Hi <%= comment.user.name%>!</p>
    <br>
    <br>
    <p>Your comment <strong><%= comment.content %></strong> just got pubished on codeial.</p>
    <br>
    <br>
    <p>Thanks!</p>
</div>
```

Introduction to Delayed Jobs

- We can decide and divide the task or the actions that are executed into two parts -
    - First, the one that needs an immediate response being sent back to the user.
    - Second, that does not need to be executed immediately.
- The email of the comment that we sent does not need to execute immediately.
- All the less important tasks can be pushed into a queue and be executed one by one.
- All the tasks that are important need to be sent to the CPU immediately so that they can be executed.

- We will be putting the emails into the queue using the **KUE** library.

Installing and Understand KUE

- Install the library **KUE** using the command { **npm install KUE** } in the terminal.
- KUE workers are codes that execute the tasks from the KUE.
- KUE workers keep on watching the queues. We have one worker at least for each queue and the worker executes tasks from the KUE.
- Workers keep on pinging the KUE asking for the new jobs available.
- These queues are maintained in JSON format in **Redis**.
- **Redis** simplifies your code by enabling you to write fewer lines of code to store, access, and **use** data in the applications
- We need to install **Redis**.
- **Redis** can be run using two ways -
    - Redis can be run in the background, or
    - We can manually run Redis.

EXTRA:
You can check out the link below to understand more about KUE and Redis -
https://www.npmjs.com/package/kue

https://redis.io/documentation

Setting KUE and Using It

- We need to configure KUE.
- We need to create a new file inside the config folder { **Kue.js** }.
- We need to require the KUE library.

  { **Kue.js** }.

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

const queue = kue.createQueue();

module.exports = queue;
```

- We need to create workers for the KUE.
- We need to create a new folder worker and inside that create a new file **{ comment_email_worker.js }.**
- **//The code for this is missing { workers/comment_email_worker.js}**

{ comments_controller.js }

```
const Comment = require('../models/comment');
const Post = require('../models/post');
const commentsMailer = require('../mailers/comments_mailer');
const queue = require('../config/kue');
const commentEmailWorker = require('../workers/comment_email_worker');
module.exports.create = async function(req, res){
    try{
        let post = await Post.findById(req.body.post);
        if (post){
            let comment = await Comment.create({
                content: req.body.content,
                post: req.body.post,
                user: req.user._id});
            post.comments.push(comment);
            post.save();
            comment = await comment.populate('user', 'name email').execPopulate();
            // commentsMailer.newComment(comment);
            let job = queue.create('emails', comment).save(function(err){
                if (err){
                    console.log('Error in sending to the queue', err);
                    return;}
                console.log('job enqueued', job.id);})
            if (req.xhr){
                return res.status(200).json({
                    data: {
                        comment: comment},
                    message: "Post created!"
                });}
            req.flash('success', 'Comment published!');

            res.redirect('/');
    }catch(err){
        req.flash('error', err);
        return;}  }
```

Forgot Password - Mini Assignment

- We should be able to send the forgot password email when the user forgets his/her password.
- There should be a form that will contain buttons for reset passwords and confirm passwords.
- The action URL of the form will be POST.
- Once the form is submitted at the server-side, matching both the passwords.
    - If they don't match, take the user back to the sign-in page.
    - If they match, we will be changing the password.
- To remove the user from the already established access token, we will create a schema.

- The schema will contain one user that is the reference to the user schema, one access token, one boolean of **isValid name** along with updated at and created at timestamps.
- As soon as the password is reset, we will find the access token in the database and we will make the **isValid** boolean false.
- Now when we refresh or go to the link again for resetting the password, it would show that your token has expired.

Summarizing Mailers and Delayed Jobs

- We sent an email but you can set an attachment, account activation emails, etc.
- We have studied KUE so that we can maintain different queues using KUE to optimize which jobs are prioritized and which are not.