Class 7 - Update / reset/ forget password, sending email

Limitations or challenges with Cookies (prev class's topic)

1. Size Limitations
    a. Storage Capacity: Each cookie is limited in size (about 4KB). This restricts the amount of data that can be stored in a single cookie.
    b. Browser Limit: Browsers limit the number of cookies that can be stored per domain. This number varies across different browsers.
2. Security Concerns
    a. Vulnerability to Theft: Cookies, especially those containing sensitive data like session tokens, can be intercepted and stolen, particularly if they are not transmitted over secure channels (HTTPS).
    b. Cross-Site Scripting (XSS): Cookies can be vulnerable to XSS attacks if the httpOnly flag is not set. This flag prevents client-side scripts from accessing cookie data.
    c. Cross-Site Request Forgery (CSRF): Cookies are automatically sent with every request to their originating domain, making them vulnerable to CSRF attacks.
3. Privacy Issues
    a. Tracking user behavior

Introduction to SMTP Servers

1. In the digital world, SMTP servers are like the post office for emails. SMTP stands for Simple Mail Transfer Protocol, and it's the standard protocol used across the internet for sending emails.

How SMTP Servers Work - A Simple Analogy

1. Writing the Email (Composing a Letter):
    a. Just as you compose a letter, you write an email. This is done through an email client or a service, which could be anything from your Gmail interface to a custom-built application.
2. Sending to SMTP Server (Dropping at the Post Office):
    a. When you hit 'send', your email client hands over the email to an SMTP server. Think of this as dropping your letter off at the post office.
    b. The SMTP server is responsible for processing your email and directing it towards its destination.
3. Routing the Email (Sorting and Delivery):
4. Recipient's Email Server (Destination Post Office):
5. Email Delivery (Mailbox Delivery):
    a. Finally, the recipient's email server delivers the email to the recipient's inbox

Using SMTP with NodeMailer and SendGrid:

NodeMailer, a module for Node.js, to send emails. NodeMailer communicates with an SMTP server to send our emails. You can configure NodeMailer to use any SMTP server

SendGrid: SendGrid offers an SMTP service but also provides a powerful API

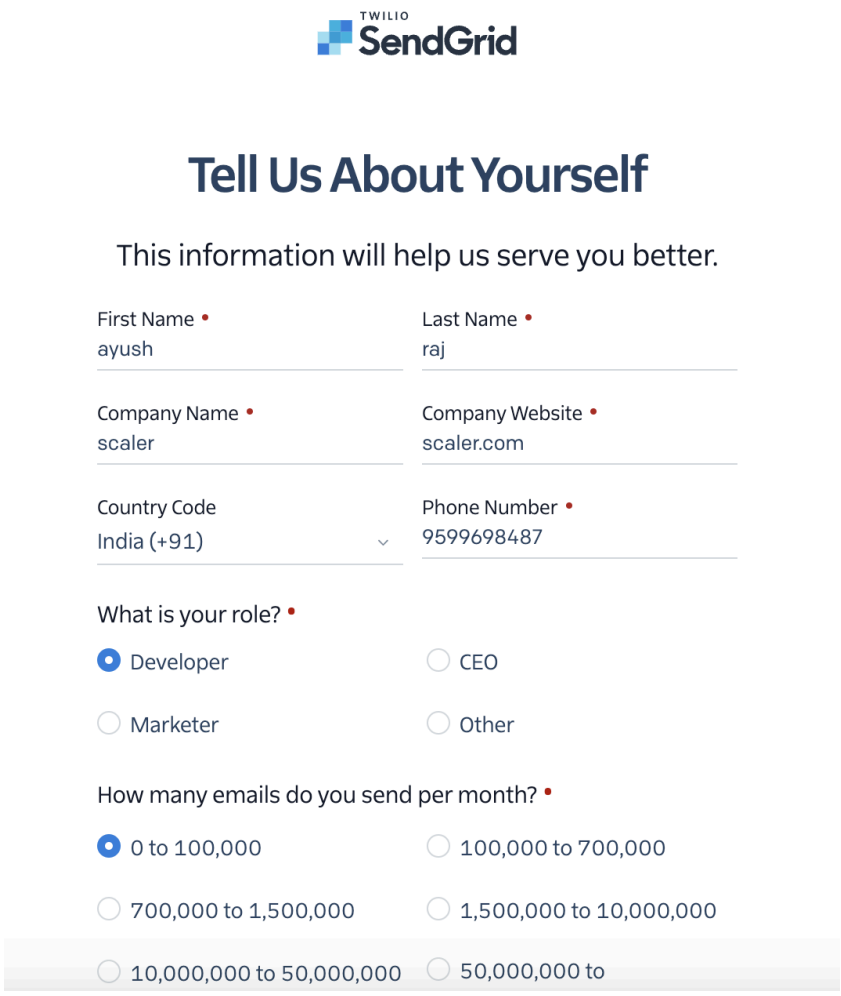
Limitations of SMTP

1. Inconsistent Rendering Across Email Clients
    a. An HTML email that looks fine in one email client might appear broken or differently in another
2. Limited HTML and CSS Support
    a. Email clients do not support the full range of HTML and CSS features that modern web browsers do.
    b. Generally tables are used for layouting

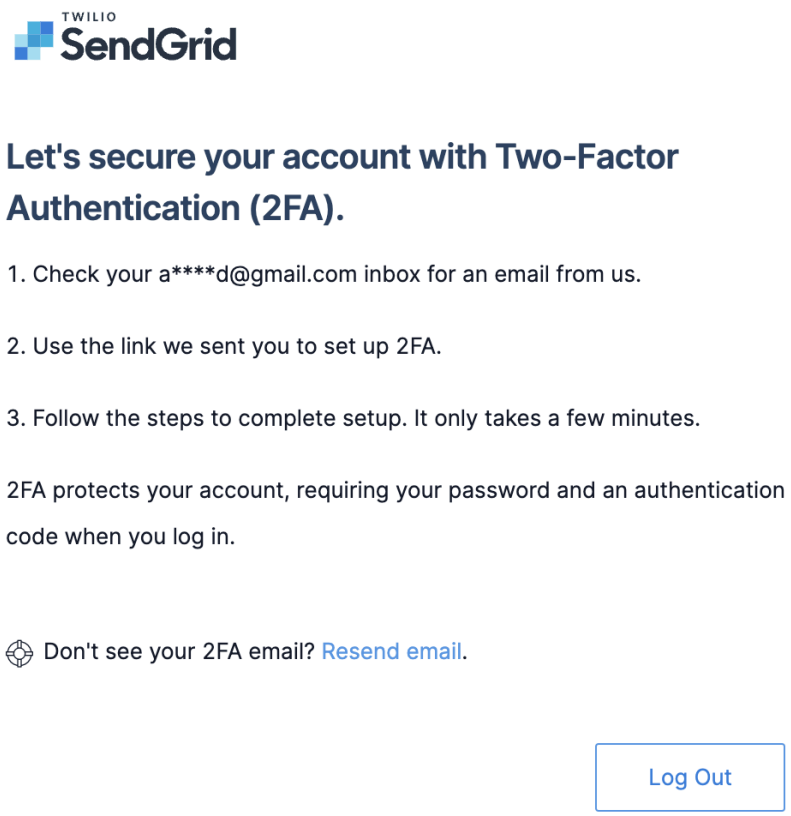
Sendgrid - SMTP server as a service

1. gmail , hotmail also provide such service
2. We will need to get one api key
    a. Why do we need api keys ?
    b. Identification, authentication
    c. Track our usage

3. Signup


TWILIO
SendGrid
Tell Us About Yourself
This information will help us serve you better.
First Name •
ayush
Last Name •
raj
Company Name •
scaler
Company Website •
scaler.com
Country Code
India (+91)
Phone Number •
9599698487
What is your role? •
Developer
CEO
Marketer
Other
How many emails do you send per month? •
0 to 100,000
100,000 to 700,000
700,000 to 1,500,000
1,500,000 to 10,000,000
10,000,000 to 50,000,000
50,000,000 to

4.



5.

6. Code - create sendGridEmail.js

```
require("dotenv").config();
const sgMail = require('@sendgrid/mail');
sgMail.setApiKey(process.env.SENDGRID_API);
const msg = {
   to: 'senders email',
   from: 'receivers email',
   subject: 'Sending with SendGrid is Fun',
   text: 'and easy to do anywhere, even with Node.js',
   html: '<strong>and easy to do anywhere, even with
Node.js</strong>',
};
sgMail
```

```
    .send(msg)
    .then(() => {
    console.log('Email sent');
})
    .catch((error) => {
    console.error(error);
});
```

5. SendGrid's Web API to send emails, not directly using an SMTP server. While SendGrid itself is an email delivery service that operates over SMTP at its core, the specific method used in this code is via their Web API, which is a different approach
6. SendGrid's Web API allows applications to send email using HTTP requests.
7. In the sendgrid’s dashboard , you will be able to see the number of emails sent, how many are opened, bounce rate, etc.

Nodemailer

1. Nodemailer is a popular Node.js module used for sending emails from within Node.js applications. It's known for its simplicity and versatility, allowing developers to send emails with just a few lines of code. Nodemailer can be used with SMTP (Simple Mail Transfer Protocol), which is the standard protocol for sending emails, or with various email sending services like SendGrid, Mailgun, etc.

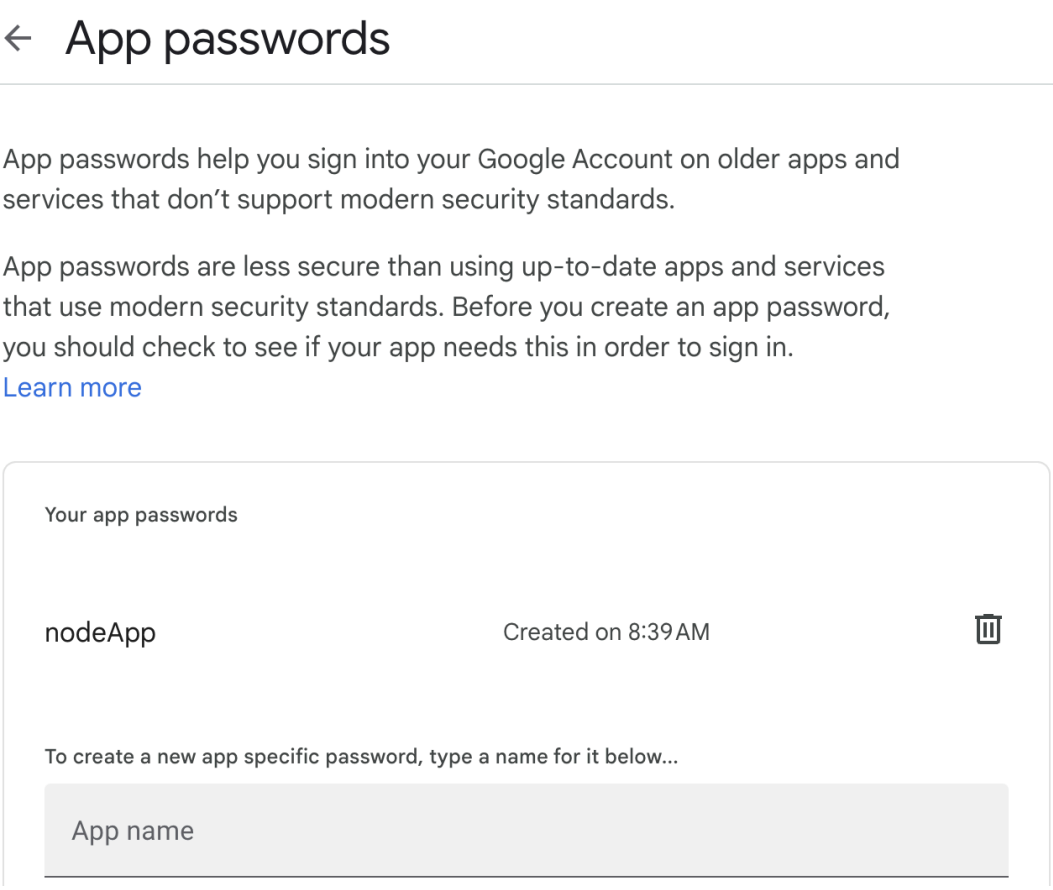
2. Create a nodemailer.js file
3. Install the package

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

4. Create a transporter object
5. This has the details about what is the service name, smtp server details, port 587 for http (less secure connections)

```
const transporter = nodemailer.createTransport({
 service: "gmail",
 host: "smtp.gmail.com",
 port: 587,
 secure: false,
 auth: {
   user: "AYUSH.RAJ.SD@GMAIL.COM",
   pass: process.env.PASSWORD,
 },
});
```

   Auth part is interesting where you enterr your gmail username and an app password
6. Now direct saving of password is not accepted
7. You need to enable two factor authentication
8. Once that is done, you will need to enable app passwords
9. This option also we will get under two factor authentication once it has been enabled

10. 

11. This will generate a 16 digit password that you can use
12. There is another way to use OAuth using google cloud project -> enable gmail api -> generate credentials but for our project case, we can do that .
13. You can also use sendgrid's documentation. That is much simpler

Creating a reusable send email function

```
/** create reusable sendmail function
@params {object} options - mail options (to, subject, text,
html)
```

```
@params {function} callback - callback function to handle
response
*/
const SENDMAIL = async (mailDetails, callback) => {
 try {
   const info = await transporter.sendMail(mailDetails);
   callback(info);
 } catch (error) {
   console.log(error);
 }
};
```

1. This accepts a parameter mailDetails which has details about who is sending, what is the subject, body of the email, etc.
2. Callback we will pass when invoking the SENDEMAIL

Email template

1. Let us look at a basic email template .
2. We use simple html , css for better compatibility
3. This is not needed

```
const HTML_TEMPLATE = (text) => {
   return `
     <!DOCTYPE html>
     <html>
       <head>
         <meta charset="utf-8">
         <title>NodeMailer Email Template</title>
         <style>
           .container {
```

```
      width: 100%;
      height: 100%;
      padding: 20px;
      background-color: #f4f4f4;
    }
    .email {
      width: 80%;
      margin: 0 auto;
      background-color: #fff;
      padding: 20px;
    }
    .email-header {
      background-color: #333;
      color: #fff;
      padding: 20px;
      text-align: center;
    }
    .email-body {
      padding: 20px;
    }
    .email-footer {
      background-color: #333;
      color: #fff;
      padding: 20px;
      text-align: center;
    }
  </style>
</head>
<body>
  <div class="container">
    <div class="email">
      <div class="email-header">
        <h1>EMAIL HEADER</h1>
      </div>
      <div class="email-body">
```

```
              <p>${text}</p>
            </div>
            <div class="email-footer">
              <p>EMAIL FOOTER</p>
            </div>
          </div>
        </div>
      </body>
    </html>
  `;
}
```

Generating the email details that we pass to SENDEMAIL

```
const message = "Hi there, you were emailed me through nodemailer"
const options = {
   from: "ayush.raj.sd@gmail.com", // sender address
   to: "raj.ayush@scaler.com", // receiver email
   subject: "Send email in Node.JS with Nodemailer using Gmail
account", // Subject line
   text: message,
   html: HTML_TEMPLATE(message),
}
```

Invoking the SENDEMAIL

```
SENDMAIL(options, (info) => {
   console.log("Email sent successfully");
   console.log("MESSAGE ID: ", info.messageId);
})
```

You can choose to export for reusability

```
module.exports = {
   SENDMAIL,
   HTML_TEMPLATE,
}
```

Reference article -
https://miracleio.me/snippets/use-gmail-with-nodemailer/

Sumamrising Sending Emails

1. We are using the SMTP protocol to send the email via Gmail's SMTP server. SMTP (Simple Mail Transfer Protocol) is the standard protocol for sending emails across the internet, and our setup with Nodemailer is configured to use Gmail's SMTP service.
2. Understanding SMTP in our Code
    a. SMTP Configuration:
        i. The createTransport function in Nodemailer is used to create a transporter object, which is configured to send emails using SMTP.
        ii. we've specified Gmail's SMTP server details (smtp.gmail.com with port 587) and authentication credentials (user email and App Password).
    b. Secure Connection:
        i. The secure flag is set to false because we're using port 587, which starts with an insecure connection

and then upgrades to a secure (TLS) connection. If we were using port 465, would set secure to true.

   c. App Password:
   d. Using an App Password (as opposed to regular Gmail password) is a more secure way to access Gmail's SMTP server.

3. What is SMTP?:
   a. SMTP is the standard protocol for sending emails and it's similar to postal mail services but in the digital world.
4. Nodemailer:
   a. Nodemailer as a module in Node.js simplifies sending emails through SMTP.
5. App Passwords and Security:
   a. understand the importance of using App Passwords for security and the risks associated with less secure methods.
6. HTML Email Templates:
   a. our code includes an HTML template for the email body. This allows for more styling and formatting compared to plain text.
7. Error Handling:
   a. try-catch block in our SENDMAIL function is used to handle errors that might occur while sending the email.
8. Environment Variables: importance of using environment variables to keep sensitive information like email addresses and passwords secure.

9. Gmail's SMTP Limits: Gmail SMTP has limits on the number of emails you can send per day, which is important for understanding the scope of its use (more suited for individual and small-scale applications).

OTP Reset Email

1. Will make some tweaks to send email with OTP
2. Change template a bit to receive message with otp

```
const HTML_TEMPLATE = (text) => {
 return `
    <!DOCTYPE html>
    <html>
      <head>
        <meta charset="utf-8">
        <title>Reset Password</title>
        <style>
          .container {
            width: 100%;
            height: 100%;
            padding: 20px;
            background-color: #f4f4f4;
          }
          .email {
            width: 80%;
            margin: 0 auto;
            background-color: #fff;
            padding: 20px;
          }
          .email-header {
```

```
            background-color: #333;
            color: #fff;
            padding: 20px;
            text-align: center;
          }
          .email-body {
            padding: 20px;
          }
          .email-footer {
            background-color: #333;
            color: #fff;
            padding: 20px;
            text-align: center;
          }
        </style>
      </head>
      <body>
        <div class="container">
          <div class="email">
            <div class="email-header">
              <h1>OTP for reset</h1>
            </div>
            <div class="email-body">
              <p>${text}</p>
            </div>
            <div class="email-footer">
              <p>EMAIL FOOTER</p>
            </div>
          </div>
        </div>
      </body>
    </html>
  `;
};
```

2. Next we can build a reusable email builder function that takes parameters of to, from ,subject , etc.

```
async function emailBuilder(to, subject, text) {
 try {
   const options = {
     from: "ayush.raj.sd@gmail.com", // sender address
     to: to, // receiver email
     subject: subject, // Subject line
     text: text,
     html: HTML_TEMPLATE(text),
   };
   SENDMAIL(options, (info) => {
     console.log("Email sent successfully");
     console.log("MESSAGE ID: ", info.messageId);
   });
 } catch (err) {
   console.log(err);
 }
}
```

3. Text is the fallback in case the html is not rendered correctly
4. Can move the from email to env or some configuration file

```
```

5. YOu can have html in a different file and then read the content using fs module. Then you can create placeholders and simply do a replace as we do on strings
6. There are email service providers like SendGrid, Mailgun which provide tracking details like opened, bounced, etc. To get these details we generally implement something called as webhooks

7. nstead of you constantly asking the email service if your email was delivered or opened (which would be inefficient), the email service promises to notify you when these events happen. This notification is the webhook.
8. We will implement webhooks for payment gateway

Forget and Reset Password

1. User clicks on forget password
2. Received an email with the token
3. Updates current password

4. Create routes in userRouter.js

```
userRouter.post("/forgetPassword", forgetPassword);
userRouter.patch("/resetPassword/:userId", resetPassword);
```

5. In the userController.js add the two handlers

```
const forgetPassword = (req, res) => {
   // user sends their email
   // verify that the email exists
   // generate a token
   // send the token to the user's email

 }

 const resetPassword = (req, res) => {
   // user sends their token and new password
   // verify that the token is valid
   // update the user's password
```

```
}
```

6. In the userModel , we add two fields for token and otpExpiry

```
a.    token:String,

b.    otpExpiry:Date
```

c.

7. We will be updating tokens when user clicks on forget Password
8. Updating the forgetPassword
9. We need a function to generate a mock otp for us

```
const otpGenerator = () => {
 return Math.floor(100000 + Math.random() * 900000);
};
```

10. Also we need to make the confirm password as not required field so that we can save the document without that field

```
const forgetPassword = async (req, res) => {
// user sends their email
// verify that the email exists
// generate a token
// send the token to the user's email
try {
  const { email } = req.body;
  const user = await User.findOne({ email });
  console.log("user", user)
  if (!user) {
    res.status(404).json({
      message: "error",
      data: "User not found",
    });
```

```
    } else {
      // generate a token
      const token = otpGenerator();
      user.token = token.toString();
      user.otpExpiry = Date.now() + 1000*60*5; // 5 minutes
      console.log("updated user", user)
      await user.save();
      sendEmailHelper(token, email)
        .then(() => {
          console.log("Email is sent");
        })
        .catch((err) => {
          console.log(err);
        });
        res.status(200).json({
            message: "success",
            data: "Email is sent",
        })

      // save the token in the user's document
      // send the token to the user's email
    }
  } catch (err) {
    res.status(500).json({
      message: "error",
      data: err.message,
    });
  }
};
```

11. Import the sendEmailHelper

```
const { sendEmailHelper } = require("../nodemailer");
```

ResetPassword

```
const resetPassword = async (req, res) => {
 // user sends their token and new password
 // req has user id in the params
 // verify that the token is valid
 // update the user's password
```

```
const resetPassword = async (req, res) => {
 // user sends their token and new password
 // req has user id in the params
 // verify that the token is valid
 // update the user's password
try{
  const { token, password, email } = req.body;
  const { userId } = req.params;
  // logic
  /**
   * 1. find the user with the id
   * 2. check if the token is valid
   * 3. check if the token is expired
   * 4. upate the password
   */
  console.log("email", email)
 //  const user = await User.findOne({email});
 // convert string id to ObjectId
 console.log("id", userId)
 const user = await User.findById(userId);
   console.log("user", user)
  if(!user){
```

```
    res.status(404).json({
      message: "error",
      data: "User not found",
    });
  }else {
     if(user.token !== token){
       res.status(400).json({
         message: "error",
         data: "Invalid token",
       });
     }else {
       if(user.otpExpiry < Date.now()){
         res.status(400).json({
           message: "error",
           data: "Token is expired",
         });
       }else {
         user.password = password;
         user.token = null;
         user.otpExpiry = null;
         await user.save();
         res.status(200).json({
           message: "success",
           data: "Password is updated",
         });
       }
     }
  }

}catch(err){
 res.status(500).json({
   message: "error",
```

```
    data: err.message,
  });
}



};
```