

THE UNIVERSITY OF BURDWAN

Of



Mankar College

**A Project submitted in partial fulfilment of
The requirement for the Honors Degree of
B.Sc. Computer Science**

ONLINE HOSPITAL MANAGEMENT SYSTEM

Paper Code: DSE-4

Year of Examination: 2023

SUBMITTED BY

SOURIN SAHA (200311700046)

SHIBAM MUKHERJEE (200311700039)

BIVA MONDAL (200311700010)

SANDEEP KUMAR (200311700035)

.....
GUIDED BY

PROF. ANURAG MUKHERJEE

Table of content

SL.NO	CONTENT	PAGE NO
1.	INTRODUCTION	3
2.	ACKNOWLEDGEMENT	5
3.	CERTIFICATE	6
4.	INTRO TO NODE JS AND MySQL	7
5.	WHY PEOPLE BUY OUR PROJECT?	8
6.	PROJECT DESCRIPTION AND DOCUMENTATION	10
7.	REFERENCE AND RESOURCES	37

INTRODUCTION

The Online Hospital Management System (OHMS) is a comprehensive web-based solution developed using Node.js, a popular server-side JavaScript runtime environment. This system aims to streamline and automate various administrative and operational tasks within a hospital, providing a centralized platform for managing patient records, appointments, billing, and other crucial aspects of healthcare management.

The healthcare industry faces numerous challenges in effectively managing patient information, scheduling appointments, coordinating between different departments, and ensuring efficient communication among healthcare professionals. The OHMS addresses these challenges by leveraging the power of Node.js to create a robust, scalable, and user-friendly online platform.

With the Online Hospital Management System, healthcare organizations can digitize their operations, significantly reducing paperwork, minimizing errors, and improving overall efficiency. The system allows authorized personnel, including doctors, nurses, administrators, and billing staff, to access and update patient records, manage appointments, and generate reports from any internet-connected device.

Key Features:

- 1. Patient Management:** The OHMS provides a comprehensive patient management module, enabling healthcare professionals to create and maintain electronic health records, track medical history, record diagnosis and treatment details, and manage patient demographics securely.
- 2. Appointment Scheduling:** The system offers an intuitive interface for scheduling and managing appointments. Patients can request appointments online, and staff members can efficiently assign and reschedule appointments based on availability, doctor preferences, and urgency.
- 3. Doctor and Staff Management:** The OHMS allows hospitals to manage their medical personnel efficiently. Doctors and staff members can maintain their profiles, update availability, and access patient records securely.
- 4. Attractive UI Interface :** The interface incorporates visually appealing elements such as a clean and modern design, pleasing color schemes, and well-designed icons. The use of appropriate fonts and spacing ensures readability and clarity throughout the system.

5. User Friendly Nature: incorporating user-friendly design principles and prioritizing ease of use, the OHMS aims to provide a seamless and enjoyable experience for all users. The system's user-friendly nature fosters efficiency, productivity, and user satisfaction, ultimately contributing to improved hospital management processes.

6. Reporting and Analytics: The system provides powerful reporting and analytics capabilities, allowing hospitals to generate insights from their data. Key performance indicators, patient statistics, and financial reports can be easily generated and analysed to support decision-making processes.

The Online Hospital Management System built on Node.js offers a scalable and secure solution for hospitals and healthcare institutions seeking to enhance operational efficiency, improve patient care, and optimize resource utilization. By harnessing the capabilities of modern technology, this system empowers healthcare professionals to focus on delivering quality care while minimizing administrative burdens.

Acknowledgement

I would like to express my sincere gratitude and appreciation to everyone who has contributed to the successful completion of Project on Online Hospital Management System for DSE – 4 .

First and foremost, I would like to extend my heartfelt thanks to my project supervisor, Professor Anurag Mukherjee, for their invaluable guidance, support, and expertise. Their insightful feedback and constant encouragement were instrumental in shaping the direction and quality of this project. I am truly grateful for their mentorship throughout the entire process.

I would also like to extend my thanks to the faculty members of Mankar College, whose teachings and insights have contributed significantly to my academic growth and development. Their dedication to excellence and their willingness to share their knowledge have been pivotal in shaping my understanding of the subject matter.

I am deeply indebted to my classmates and friends who provided me with encouragement, support, and valuable suggestions throughout the project. Their input and discussions have greatly enriched my understanding of the topic and enhanced the overall quality of my work.

I would like to acknowledge the invaluable assistance received from the staff and personnel of the college library and various research facilities. Their prompt and efficient services enabled me to access the resources necessary for conducting thorough research and gathering relevant information.

Lastly, I would like to express my heartfelt gratitude to my family and loved ones who have been a constant source of encouragement and support throughout my college journey. Their unwavering belief in my abilities has been instrumental in keeping me motivated and focused on achieving my goals.

To all those mentioned above and to anyone else who has contributed in any way to the completion of this project, I extend my sincerest appreciation. Your support, guidance, and encouragement have played a significant role in shaping my academic and personal growth.

Thank you all once again for your invaluable contributions.

Sincerely,

SOURIN SAHA

(200311700046)

Certificate

This is to certify that SOURIN SAHA of
MANKAR COLLEGE, UNIVERSITY ROLL
NO.: 200311700046, successfully
completed his project on ONLINE
HOSPITAL MANAGEMENT SYSTEM using
NODE JS, MYSQL, HTML, CSS AND
JAVASCRIPT under the guidance of PROF.
ANURAG MUKHERJEE

.....

PROF ANURAG MUKHERJEE

INTRO OF NODE JS AND MySQL

NODE JS:

Node.js is an open-source, cross-platform server environment that runs on various operating systems and executes JavaScript code outside a web browser. It allows developers to use JavaScript for server-side scripting and to generate dynamic web page content. Node.js has an event-driven architecture and is capable of asynchronous I/O, optimizing throughput and scalability in web applications. The Node.js project is now governed by the OpenJS Foundation, facilitated by the Linux Foundation's Collaborative Projects program. Many corporations use Node.js software, including GoDaddy, Groupon, IBM, LinkedIn, Microsoft, Netflix, PayPal, SAP, Walmart, Yahoo!, and Amazon Web Services.

History:

Node.js was initially written by Ryan Dahl in 2009, about thirteen years after the introduction of the first server-side JavaScript environment, Netscape's LiveWire Pro Web. The initial release supported only Linux and Mac OS X. Its development and maintenance was led by Dahl and later sponsored by Joyent. Dahl criticized the limited possibilities of the most popular web server in 2009, Apache HTTP Server, to handle a lot of concurrent connections and the most common way of creating code (sequential programming) .

Mysql:

MySQL is an open-source relational database management system (RDBMS). It is the world's most popular open-source database and ranks as the second-most-popular database, behind Oracle Database. MySQL powers many of the most accessed applications, including Facebook, Twitter, Netflix, Uber, Airbnb, Shopify, and Booking.com. MySQL is ideal for both small and large applications. It offers a range of advanced features, management tools, and technical support to achieve high levels of scalability, security, reliability, and uptime. Over 2000 ISVs, OEMs, and VARs rely on MySQL as their products' embedded database to make their applications more competitive.

History

MySQL was created by a Swedish company, **MySQL AB**, founded by Swedes David Axmark, Allan Larsson and Finland Swede Michael "Monty" Widenius . The project of MySQL began in 1979 in the form of **UNIREG**, an in-house database tool developed to manage databases . The first version of MySQL appeared on **23 May 1995**. The company was later acquired by **Sun Microsystems** in 2008 for about \$1 billion . In 2010, when Oracle acquired Sun, Widenius forked the open-source MySQL project to create **MariaDB** .

Why people buy our Project?

1. **Comprehensive Functionality:** Emphasize the wide range of features and functionalities your online hospital management system offers. Showcase how it covers critical aspects of hospital operations, such as patient management, appointment scheduling, billing, reporting, and more.
2. **User-Friendly Interface:** Highlight the intuitive and user-friendly interface of your system. Describe how it simplifies complex tasks, reduces training time for new users, and enhances overall usability, making it easy for healthcare professionals of all technical backgrounds to navigate and utilize.
3. **Customizability and Scalability:** Emphasize the flexibility and scalability of your project. Showcase how the system can be tailored to meet the specific needs of different healthcare organizations, accommodating variations in workflows, specialties, and organizational structures.
4. **Mobile Compatibility:** Highlight the cross-platform compatibility and availability of a mobile version of your system. Emphasize how healthcare professionals can access and manage patient information, appointments, and other functionalities on-the-go using their smartphones or tablets.
5. **Enhanced Efficiency and Productivity:** Showcase how your project streamlines hospital processes, reduces administrative burden, and improves overall efficiency. Discuss how it automates tasks, eliminates paperwork, minimizes errors, and enables healthcare professionals to focus more on patient care.
6. **Data Security and Compliance:** Highlight the robust security measures implemented in your project to protect sensitive patient data. Describe how the system adheres to industry standards and compliance regulations (such as HIPAA) to ensure the confidentiality, integrity, and privacy of patient information.
7. **Technical Support and Documentation:** Assure potential buyers that your project comes with comprehensive documentation, user guides, and technical support to facilitate smooth implementation and usage. Highlight any additional resources you provide, such as training materials or online forums, to support users during and after deployment.

8. **Cost-Effectiveness:** Demonstrate the cost-effectiveness of your project compared to developing a similar system from scratch. Explain how purchasing your project can save considerable development time, resources, and costs associated with custom software development.

9. **Integration Capabilities:** Highlight the system's ability to integrate with other healthcare systems, such as electronic medical record (EMR) systems, laboratory information systems, or billing systems. Showcase how this interoperability enhances data exchange and facilitates seamless workflows across different departments.

10. **Testimonials and Success Stories:** Share testimonials or case studies from healthcare organizations or professionals who have successfully implemented and benefited from your project. Highlight their positive experiences, improved operational efficiency, and patient outcomes as evidence of the value your system brings.

PROJECT DESCRIPTION AND DOCUMENTATION

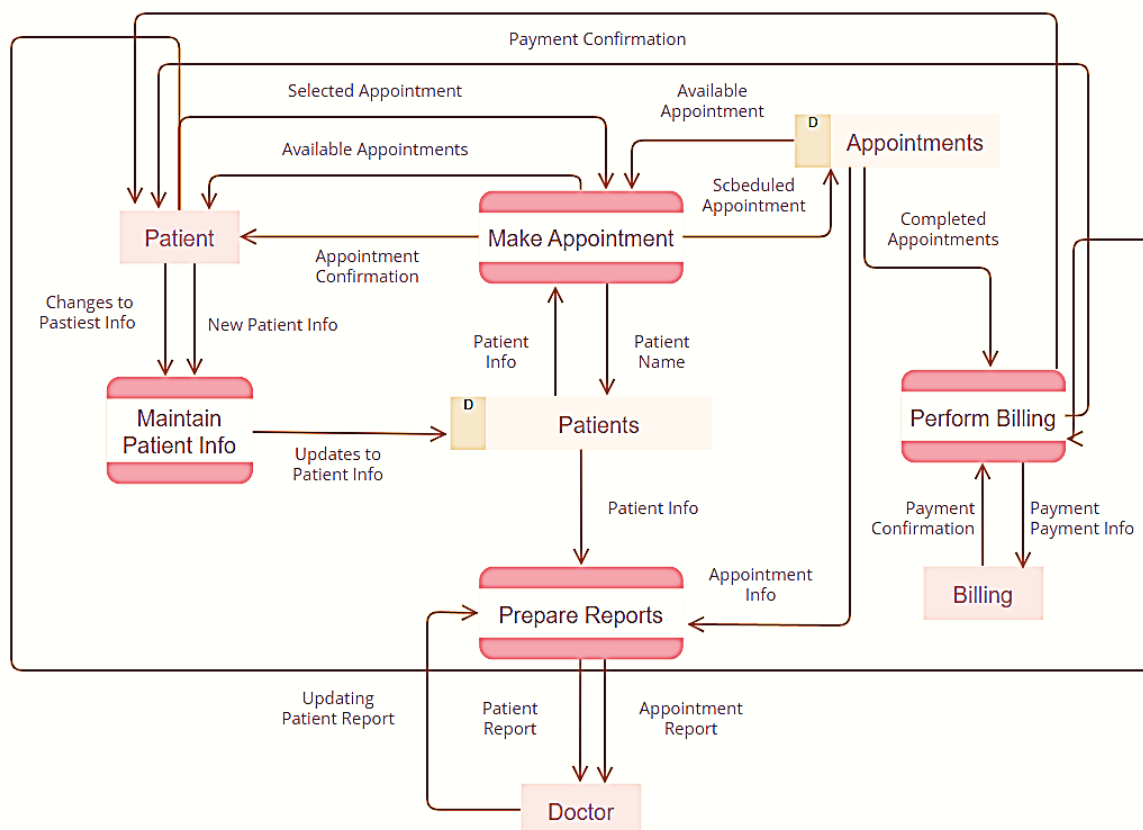
Dependencies

```
const express = require("express");
const app = express();
const port = process.env.PORT || 3000;
const database = require(__dirname + "/database");
const bodyParser = require("body-parser")
const sessions = require("express-session");
const router = express.Router();
const { name } = require("ejs");
const cookieParser = require("cookie-parser");
app.use( express.static(__dirname + "public" ) );
const nodemailer = require("nodemailer");
const crypto = require("crypto");
```

1. **express:** Express is a web application framework for Node.js that simplifies the process of building web applications. It provides a set of features and middleware for handling HTTP requests, routing, and rendering views.
2. **app:** It represents an instance of the Express application. The app object is used to define routes, middleware, and other application settings.
3. **port:** It specifies the port on which the server will listen for incoming requests. If the **process.env.PORT** environment variable is set, the server will use that port. Otherwise, it will default to port 3000.
4. **database:** It is a custom module or file that handles the database connection and provides methods for executing database queries.
5. **body-parser:** It is a middleware that parses the request body and makes it available on the **req.body** property. It is used to extract form data or JSON data from incoming requests.
6. **sessions:** It is a middleware that enables session management in Express. It provides a way to store user-specific data across multiple requests by creating and managing session objects.
7. **router:** It is an instance of the Express Router class. The router allows you to define routes and middleware separately and then mount them on the main app using **app.use()**.
8. **ejs:** EJS (Embedded JavaScript) is a templating language that allows you to embed JavaScript code into HTML templates. It is used for rendering dynamic HTML content on the server-side.
9. **cookie-parser:** It is a middleware that parses cookies attached to incoming requests and makes them available on the **req.cookies** property. It is used for handling cookies in Express applications.

10. **nodemailer:** Nodemailer is a module that allows you to send emails using Node.js. It provides an easy-to-use API for sending emails using various email service providers.
11. **crypto:** It is a built-in Node.js module that provides cryptographic functionality. It is used in this code snippet for generating cryptographic hashes, such as password hashes or unique tokens.

Data Flow Diagram



Database Connection

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

const conn = mysql.createConnection({
  host: "localhost",
  port: 3306,
  user: "root",
  password: "",
  database: "test",
});
```

This code snippet sets up a MySQL database connection using the `mysql` module and exports the connection object to be used in other parts of the application. Let's break down the code:

1. `const mysql = require("mysql");`: This line imports the `mysql` module, which is a popular Node.js library for interacting with MySQL databases.
2. `const conn = mysql.createConnection({ ... });`: This creates a MySQL database connection object named `conn`. It provides the necessary configuration details, such as the host, port, username, password, and database name, to connect to the MySQL database.
3. `conn.connect(function(error) { ... });`: This attempts to connect to the MySQL database using the configuration provided. It takes a callback function as an argument to handle the connection process.
4. Inside the callback function, it checks if there is an error while connecting to the database. If an error occurs, it logs the error to the console. If the connection is successful, it logs a success message to the console, indicating that the database connection has been established.
5. `module.exports = conn;`: This line exports the `conn` object so that other parts of the application can use this established database connection. By exporting the connection, other modules can require and use it to perform database operations.

Mailing service

```
const transporter = nodemailer.createTransport({
  service: "gmail",
  auth: {
    user: "testu0292@gmail.com",
    pass: "ljykkbtmyliprbfz"
  }
});

function mailing(toUser, subjectString, textString) {
  let mailoption = {
    from: 'admin<testu0292@gmail.com>',
    to: toUser,
    subject: subjectString,
    text: textString
  }

  transporter.sendMail(mailoption, function(err, info){
    if (err) {
      console.log(err);
    }
    console.log('email is send ... ' + info.response);
  });
}
```

This code snippet sets up a nodemailer transport configuration and defines a function for sending emails. Let's analyze the code:

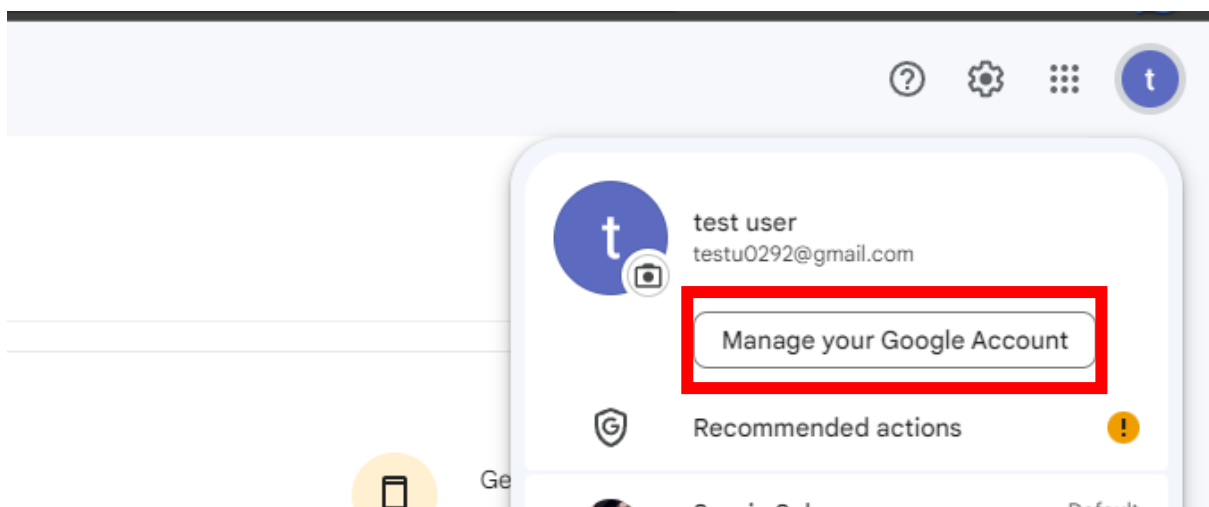
1. `const transporter = nodemailer.createTransport({ ... })`: This line creates a nodemailer transporter object using the `createTransport` method. The transporter is configured to use the Gmail service for sending emails. The `auth` object contains the Gmail account credentials (email and password) used to authenticate with the Gmail SMTP server.
2. `function mailing(toUser, subjectString, textString) { ... }`: This is a function named `mailing` that takes three parameters: `toUser` (the email address of the recipient), `subjectString` (the subject of the email), and `textString` (the body/content of the email).
3. `let mailoption = { ... }`: This creates an object `mailoption` that specifies the email options, including the sender, recipient, subject, and text content of the email.
4. `transporter.sendMail(mailoption, function(err, info) { ... })`: This line invokes the `sendMail` method of the transporter object to send the email. The `mailoption` object is passed as the first argument, and a callback function is provided as the second argument to handle any errors and log the response information.

Inside the callback function, it checks if there is an error while sending the email and logs the error if one occurs. If the email is sent successfully, it logs the response information.

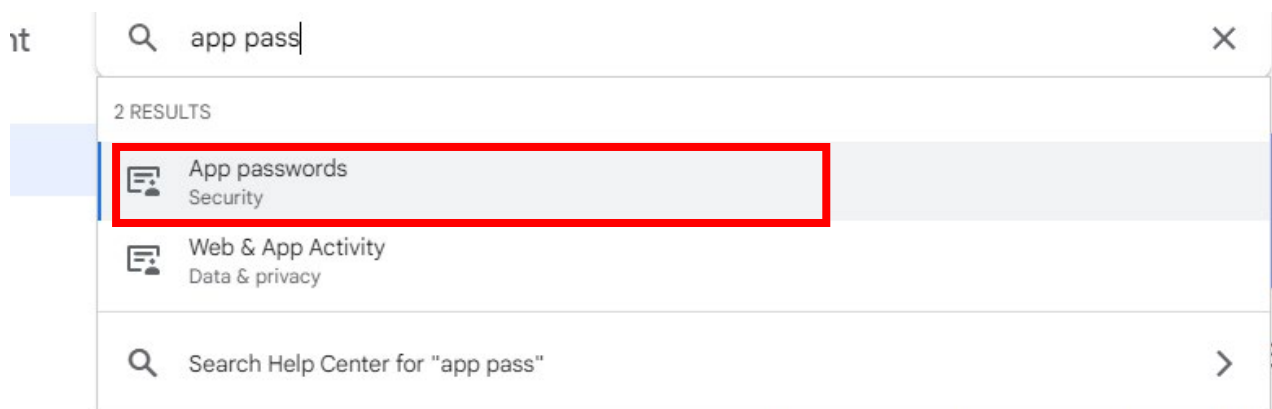
Overall, this code sets up nodemailer to use Gmail as the email service and defines a function for sending emails with the provided configuration. The `mailing` function can be used to send emails by passing the recipient's email address, subject, and content as parameters.

To get the local user password follow the following steps

Step – 1



Step 2:



Type 'app password' in search bar and hit enter.

Step 3 :

← App passwords

App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it. [Learn more](#)

You don't have any app passwords.

Select the app and device you want to generate the app password for.

nodemailer X

GENERATE

Click select apps -> others

Type "nodemailer" as our mailing provide is nodemailer

Hit GENARATE

You will be provided with a 16 digit code that is your local pass word for mailing service

Random string and integer generator

```
function generateRandomString(length) {
  const characters =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&';
  let randomString = "";

  for (let i = 0; i < length; i++) {
    const randomIndex = crypto.randomInt(0, characters.length);
    randomString += characters.charAt(randomIndex);
  }

  return randomString;
}

function generateRandomint(length) {
  const characters = '0123456789';
  let randomString = "";

  for (let i = 0; i < length; i++) {
    const randomIndex = crypto.randomInt(0, characters.length);
    randomString += characters.charAt(randomIndex);
  }

  return randomString;
}
```

These code snippets provide two functions for generating random strings and random integers of a specified length. Let's analyze them separately:

1. `generateRandomString(length)`: This function generates a random string of the specified `length`. It uses a set of characters that includes uppercase letters, lowercase letters, digits, and special characters. Here's how it works:

- It initializes an empty string called `randomString`.
- It loops `length` number of times to generate the desired length of the random string.
- In each iteration, it generates a random index using `crypto.randomInt(0, characters.length)`. This index is used to select a character from the `characters` string.
- The selected character is concatenated to `randomString`.
- Finally, the function returns the generated `randomString`.

2. `generateRandomint(length)`: This function generates a random integer of the specified `length`. It uses only digits (0-9) for generating the random integer. Here's how it works:

- It initializes an empty string called `randomString`.
- It loops `length` number of times to generate the desired length of the random integer.
- In each iteration, it generates a random index using `crypto.randomInt(0, characters.length)`. This index is used to select a digit from the `characters` string.
- The selected digit is concatenated to `randomString`.
- Finally, the function returns the generated `randomString`.

Both functions rely on the `crypto.randomInt(min, max)` function to generate random indices. However, the code you provided does not include the required import statement for the `crypto` module. Make sure to import the `crypto` module before using these functions.

Session declaration

```
const oneDay = 1000 * 60 * 60 * 24;
app.use(sessions({
  secret: "thisismysecrctekeyfhrgfgrfrty84fwir767",
  saveUninitialized:true,
  cookie: { maxAge: oneDay },
  resave: false
}));
```

1. `const oneDay = 1000 * 60 * 60 * 24;`: This line defines a constant variable `oneDay` and assigns it the value of the number of milliseconds in one day. This constant is used to set the maximum age of the session cookie.

2. `app.use(sessions({ ... }));`: This line registers the `sessions` middleware with the Express application. The middleware is configured with an object containing the following options:

- `secret`: A secret string used to sign the session ID cookie. It is recommended to use a long and secure secret to prevent session hijacking.

- `saveUninitialized`: A boolean value indicating whether to save uninitialized sessions to the store. In this case, it is set to `true`, which means a new session will be saved to the store even if it is not modified.

- `cookie`: An object specifying the options for the session cookie. In this case, the `maxAge` option is set to `oneDay`, which means the session cookie will expire after one day of inactivity.

- `resave`: A boolean value indicating whether to save the session back to the store, even if it was not modified during the request. In this case, it is set to `false`, which means the session will not be saved if it was not modified.

By configuring and using the `sessions` middleware with these options, the Express application can handle and manage user sessions, storing session data and using cookies to identify and track users.

Sign up page and logic.

MEDICAL CARE About Department Doctors Contact Us Search... Signup Login

REGISTRATION FROM

Your Name

First name Last name

Email

example@example.com

D.O.B

dd-mm-yyyy

Mobile No.

+91XXXXXXXXXX

Gender

SELECT GENDER

landmark:

City State Zip

Country

SELECT Country

Submit

```
app.get("/signup",function(req,res,next){
  if (req.session.loggedin == true) {
    res.redirect("/home");
  } else {
    res.render(__dirname+"/public/signup/index.ejs",{token: false});
  }
});
app.post("/signup", function(req,res){
  var F_name = String(req.body.F_name);
  var L_name = req.body.L_name;
  var Gender = req.body.Gender;
  var Country = req.body.Country;
  var dob = req.body.dob;
  var email= req.body.email;
  var mobile = req.body.Mobile;
  var H_no = req.body.H_no;
  var add1 = req.body.add1;
  var add2 = req.body.add2;
  var pin = req.body.pin;
  var Addrss = H_no+" "+add1+" State: "+add2+" P.I.N. : "+pin;
  var password = generateRandomString(15);
  var username = F_name+'@'+generateRandomint(4);
```

```

database.query("INSERT INTO `user` ( `F_name`, `L_name`, `email`, `Address`, `Country`,
`password`, `Gender`, `username`, `mobile_no`, `dob`) VALUES
(?,?,?,?,?,?,?,?,?);",[F_name,L_name,email,Addrsss,Country,password,Gender,username,mobile,dob], (err,results,fields)=>{
if (err) {
return console.log(err);
}
var SubjectString = 'user detail';
var textString = 'welcome user , \n This your user detail '+F_name+' '+L_name+' please put this
detail in login page to login out website \n Your Username is : '+username+' and Your Password
is :'+ password +' And Your user id you will find it in your profile page '+'\n thanks for visiting our
website' ;

mailing(email ,SubjectString , textString );
res.redirect('/home');
return console.log(results);
});
})

```

1. `app.get("/signup", function(req, res, next) {...})`: This route handler is for handling the HTTP GET request to the "/signup" path. It checks if the user is already logged in (`req.session.loggedin == true`). If the user is logged in, it redirects to the "/home" page. Otherwise, it renders the "index.ejs" file located in the "/public/signup" directory and passes a "token" variable with the value of false to the template engine for rendering.

2. `app.post("/signup", function(req, res) {...})`: This route handler is for handling the HTTP POST request to the "/signup" path. It processes the form data submitted by the user for signing up. Here's how it works:

- It retrieves the values of various form fields such as ``F_name``, ``L_name``, ``Gender``, ``Country``, ``dob``, ``email``, ``Mobile``, ``H_no``, ``add1``, ``add2``, and ``pin`` from the request body.

- It concatenates the values of ``H_no``, ``add1``, ``add2``, and ``pin`` to form an address string called ``Addrsss``.

- It generates a random password using the ``generateRandomString`` function with a length of 15 characters.

- It generates a random username by combining the value of ``F_name``, "@" symbol, and the result of the ``generateRandomint`` function with a length of 4 digits.

- It executes a database query to insert the user's information into the "user" table. The query includes placeholders (``?``) for the values that need to be inserted.

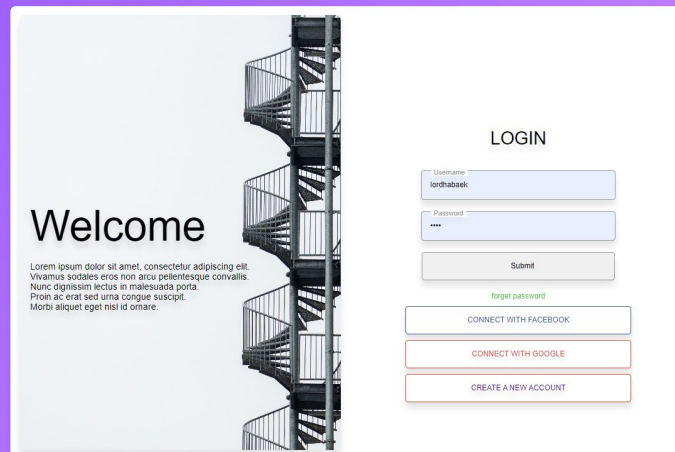
- Inside the query's callback function, it checks for any error in executing the query and logs the error if one occurs.

- It composes an email message to send to the user with their username and password details using the ``mailing`` function (assuming it exists).

- It redirects the user to the "/home" page after successful signup.

Overall, this code handles the signup process by retrieving the form data, inserting the user's information into the database, generating a random username and password, and sending an email with the signup details. After successful signup, the user is redirected to the home page.

Login page and logic



```
let username , password;
app.get("/signin",function(req,res,next){
  if (req.session.loggedin == true) {
    database.query("SELECT * FROM user WHERE username = ? AND password = ?",
[username, password], function(err,rows){
      res.render(__dirname+"/public/profile/index.ejs",{data : rows, token : true});
    });
  } else {
    res.render(__dirname+"/public/login/index.ejs");
  }
});

app.post('/auth', function(request, response) {
  // Capture the input fields
  username = request.body.username;
  password = request.body.password;
  // Ensure the input fields exists and are not empty
  if (username && password) {
    // Execute SQL query that'll select the account from the database based on the specified
    username and password
    database.query("SELECT * FROM user WHERE username = ? AND password = ?",
[username, password], function(error, results, fields) {
      // If there is an issue with the query, output the error
      if (error) throw error;
      // If the account exists
      if (results.length > 0) {
        // Authenticate the user

```

```

    request.session.loggedin = true;
    request.session.username = username;
    database.query("UPDATE `user` SET `isActive` = 1 WHERE username =
?',[username],function(error,res,next){
    if (error) {
        console.log(error);
    }
    });
    // Redirect to home page
    response.redirect('/profile');
} else {
    response.send("Incorrect Username and/or Password!");
}
response.end();
});
} else {
    response.send("Please enter Username and Password!");
    response.end();
}
});

```

This code snippet represents a server-side route handler for an HTTP POST request to authenticate a user. Let's break down the code and understand its functionality:

The route handler is defined for the path '/auth' using the `app.post` function. It listens for HTTP POST requests.

Inside the route handler function, the input fields "username" and "password" are captured from the request body.

The code checks if both the username and password fields are provided and not empty.

If the input fields are valid, a SQL query is executed to select the user account from a database table named "user" based on the specified username and password.

The database query is performed using the `database.query` function. It takes the SQL query string and an array of parameters to be replaced in the query for preventing SQL injection.

If there is an error in executing the database query, an error is thrown.

If the query execution is successful, the code checks if any results are returned from the query (i.e., if the account exists).

If the account exists, the user is authenticated by setting two properties in the session object: `loggedin` is set to `true`, and `username` is set to the provided username.

Another database query is executed to update the "isActive" field of the user in the database table to 1 (assuming it represents an active user). This query uses the `UPDATE` statement and replaces the username placeholder with the provided username.

If there is an error in executing the update query, it is logged to the console.

After successful authentication and updating the user's status, the response is redirected to the '/profile' page.

If the provided username and password do not match any account, a response is sent back with the message 'Incorrect Username and/or Password!'.

If the username or password fields are missing or empty, a response is sent back with the message 'Please enter Username and Password!'.

Forgot page and logic

```
app.get('/forgot_pass',(req,res)=> res.render(__dirname+'/public/common/forgot.ejs'));
app.post('/forgot_pass',function(req,res,next){
  var email = req.body.email;
  database.query('select * from user where email = ?', [email], function(error, result){
    if (error) {
      console.log(error);
    }
    let rows = Object.values(JSON.parse(JSON.stringify(result)));

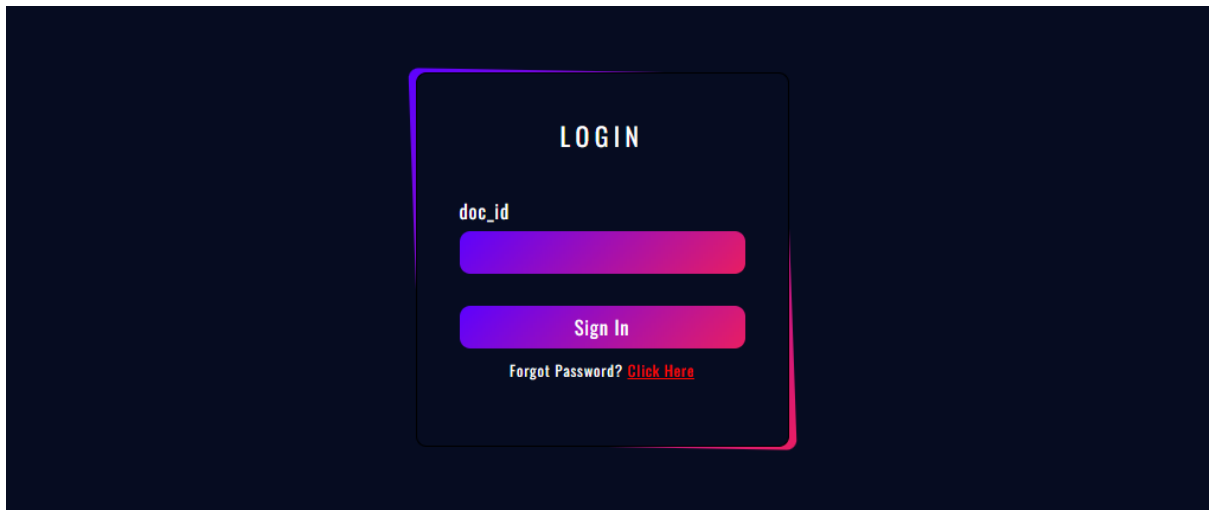
    var SubjectString = " your Password" ;
    var PasswordString = "your forgotten password is : " + rows[0].password + " Please Try to Remember it " ;
    console.log(rows[0].password)
    mailing(email , SubjectString, PasswordString );
    res.redirect('/signin');
  });
});
```

1. `app.get('/forgot_pass', (req, res) => res.render(__dirname+'/public/common/forgot.ejs'))`: This route handler is for handling the HTTP GET request to the `/forgot_pass` path. It renders the `forgot.ejs` file located in the `/public/common` directory using the template engine. This file is responsible for displaying a form to input the email address for password recovery.

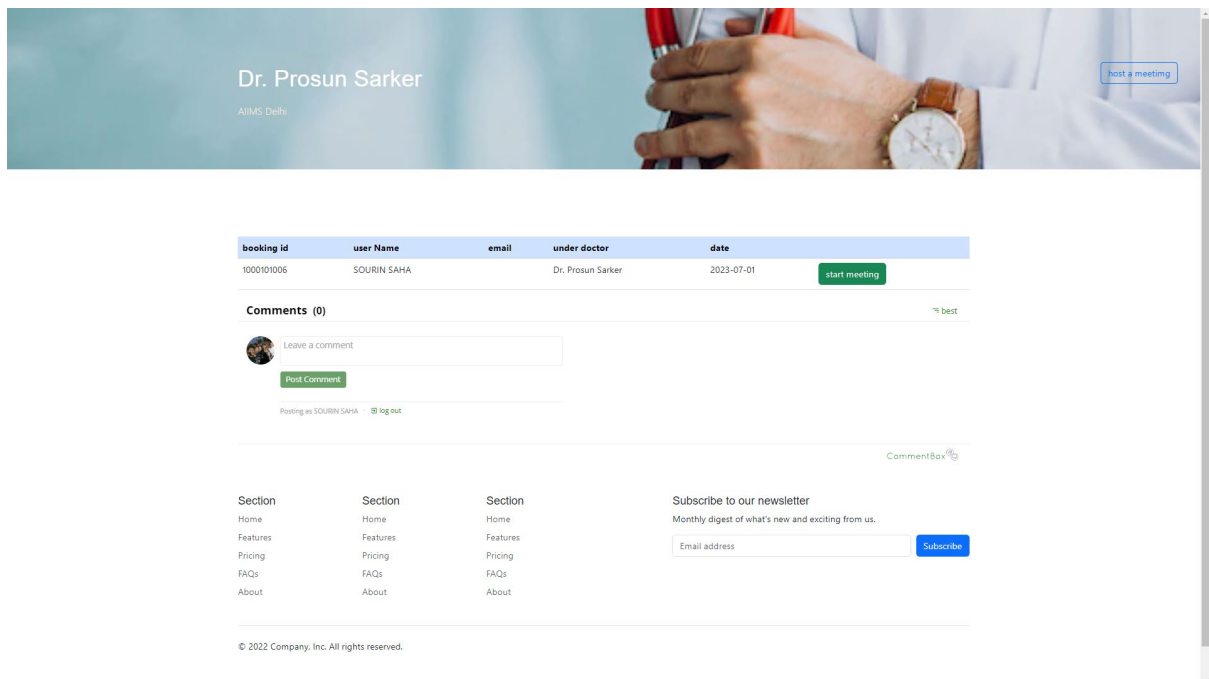
2. `app.post('/forgot_pass', function(req, res, next) {...})`: This route handler is for handling the HTTP POST request to the `/forgot_pass` path. It processes the form data submitted by the user to recover the forgotten password. Here's how it works:

- It retrieves the value of the 'email' field from the request body.
- It executes a database query to select the user's information from the 'user' table based on the provided email address. The query includes a placeholder ('?') for the email value.
- Inside the query's callback function, it checks for any error in executing the query and logs the error if one occurs.
- If the query is successful, it extracts the rows from the result and assigns them to the `rows` variable.
- It composes an email message with the subject "your Password" and the forgotten password retrieved from `rows[0].password`.
- It uses the `mailing` function (assuming it exists) to send the email to the user's email address.
- Finally, it redirects the user to the '/signin' page after completing the password recovery process.

Doctor login and Dashboard



Doctor login page



Dashboard for Doctor

```
app.get('/doctor_login',(req,res)=> res.render(__dirname+"/public/login/doctor.ejs"));
let docToken = false;
app.post('/doctor_login',function(req,res,next){
  var doc_id=req.body.doc_id;
  // var password = req.body.password;
  database.query('select * from doctor where doc_id = ?', [doc_id],function(err,rows,next){
    if (err) {
      console.log(err);
      res.send('<h1>Theres ome error </h1> ');
    }
  });
});
```

```

    }
    else
    {
        let data = Object.values(JSON.parse(JSON.stringify(rows)));
        // res.cookie('docDetail', data);

        // req.session.docToken = true;
        docToken = true;
        console.log(data[0]);
        res.redirect('/doc_dashbord?doc_id='+doc_id+'&doc_name='+data[0].doc_name);
    }
  });
});

app.get('/doc_dashbord',function(req,res,next){
  if(docToken = false){
    res.redirect('/doctor_login');
  }
  else{
    var doc_name = req.query.doc_name;
    var doc_id = req.query.doc_id;
    database.query('SELECT * FROM `doctor` WHERE doc_id = ?', [doc_id], function(err, result1){
      let rows = Object.values(JSON.parse(JSON.stringify(result1)));
      database.query('select * from booking where doc_name = 
?', [rows[0].doc_name], function(err, result2){
        res.render(__dirname + "/public/profile/doctorProfile.ejs", { data : result1, userData : result2
      });
    });
  });
});
});

```

1. `app.get('/doctor_login', (req, res) => res.render(__dirname+"/public/login/doctor.ejs"))`: This route handler is for handling the HTTP GET request to the '/doctor_login' path. It renders the 'doctor.ejs' file located in the '/public/login' directory using the template engine. This file is responsible for displaying a login form for doctors.

2. `let docToken = false;`: This line declares a variable `docToken` and initializes it with the value `false`. This variable is used to keep track of the doctor's login status.

3. `app.post('/doctor_login', function(req, res, next) {...})`: This route handler is for handling the HTTP POST request to the '/doctor_login' path. It processes the login form data submitted by the doctor. Here's how it works:

- It retrieves the value of the 'doc_id' field from the request body.

- It executes a database query to select the doctor's information from the 'doctor' table based on the provided 'doc_id'. The query includes a placeholder ('?') for the 'doc_id' value.
- Inside the query's callback function, it checks for any error in executing the query and logs the error if one occurs. If there is an error, it sends a response with an error message.
- If the query is successful, it extracts the rows from the result and assigns them to the `data` variable.
- It sets the `docToken` variable to `true`, indicating that the doctor is logged in.
- It redirects the doctor to the '/doc_dashbord' page, passing the 'doc_id' and 'doc_name' as query parameters.

4. `app.get('/doc_dashbord', function(req, res, next) {...})`: This route handler is for handling the HTTP GET request to the '/doc_dashbord' path. It displays the doctor's dashboard page. Here's how it works:

- It checks if the `docToken` variable is false. If so, it means that the doctor is not logged in, and it redirects them back to the '/doctor_login' page.
- If the `docToken` variable is true, it retrieves the values of 'doc_name' and 'doc_id' from the query parameters.
- It executes a database query to select the doctor's information from the 'doctor' table based on the 'doc_id'.
- Inside the query's callback function, it checks for any error in executing the query and logs the error if one occurs.
- If the query is successful, it extracts the rows from the result and assigns them to the `rows` variable.
- It executes another database query to select booking information for the doctor's profile based on the 'doc_name'.
- Finally, it renders the 'doctorProfile.ejs' file located in the '/public/profile' directory, passing the doctor's information and booking data as variables to the template engine.

Admin page

```
app.get('/admin', function(req, res, next) {
  if (adminToken !== true) {
    res.redirect('/admin_login');
  }
  else{
    database.query('Select count( `doc_id` ) as number FROM doctor UNION ALL SELECT
count( user_id ) FROM user UNION all SELECT COUNT( dept_id ) FROM department UNION
all SELECT COUNT( Admin_id ) from admin',function(err , result){
      if (err) {
        console.log(err);
      }
      database.query('SELECT * FROM user',function(err,rows){
        if(err){
          console.log('error', err);
        }

        res.render(__dirname+'/public/admin/index.ejs',{ amount : result , data:rows });
      });

    });}
});
```

`app.get('/admin', function(req, res, next) {...})`: This route handler is for handling the HTTP GET request to the '/admin' path. It renders the admin dashboard page. Here's how it works:

It checks if the `adminToken` variable is not true, indicating that the admin is not logged in. If the admin is not logged in, it redirects them to the '/admin_login' page.

If the `adminToken` variable is true, it proceeds to execute the code inside the 'else' block.

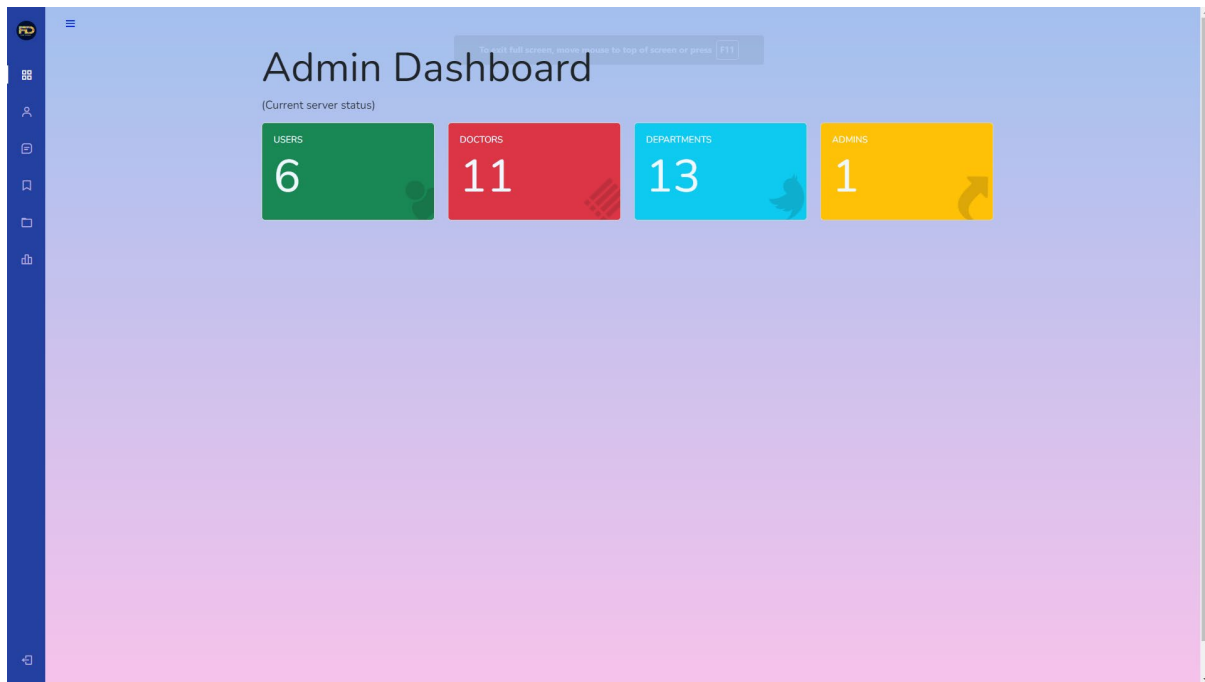
It executes a database query that retrieves the count of doctors, users, departments, and admins from their respective tables. The 'UNION ALL' operator combines the results of the four queries into a single result set.

Inside the callback function of the first query, it checks for any error in executing the query and logs the error if one occurs.

After retrieving the counts, it executes another database query to select all rows from the 'user' table.

Inside the callback function of the second query, it checks for any error in executing the query and logs the error if one occurs.

If both queries are successful, it renders the 'index.ejs' file located in the '/public/admin' directory, passing the count data and user data as variables to the template engine.



This is the base admin page
further it divided into 4 part that shows the tables of :

1. USER
2. DOCTOR
3. DEPARTMENT
4. ADMINS

User management page

```
app.get('/admin/user', function(req, res, next) {
  if (adminToken !== true) {
    res.redirect('/admin_login');
  }
  else{
    database.query('Select count( `doc_id` ) as number FROM doctor UNION ALL SELECT
count( user_id ) FROM user UNION all SELECT COUNT( dept_id ) FROM department UNION
all SELECT COUNT( Admin_id ) from admin',function(err , result){
      if (err) {
        console.log(err);
      }
      database.query('SELECT * FROM user',function(err,rows){
        if(err){
          console.log('error', err);
        }

        res.render(__dirname+'/public/admin/adminUser.ejs',{ amount : result , data:rows ,
title:"User Table"});
      });
    });
  }
});
```


This code snippet represents a server-side route handler for the '/admin/user' path, which is responsible for displaying the user table in the admin dashboard. Let's break down the code and understand its functionality:

`app.get('/admin/user', function(req, res, next) {...})`: This route handler is for handling the HTTP GET request to the '/admin/user' path. It renders the user table page in the admin dashboard. Here's how it works:

It checks if the `adminToken` variable is not true, indicating that the admin is not logged in. If the admin is not logged in, it redirects them to the '/admin_login' page.

If the `adminToken` variable is true, it proceeds to execute the code inside the 'else' block.

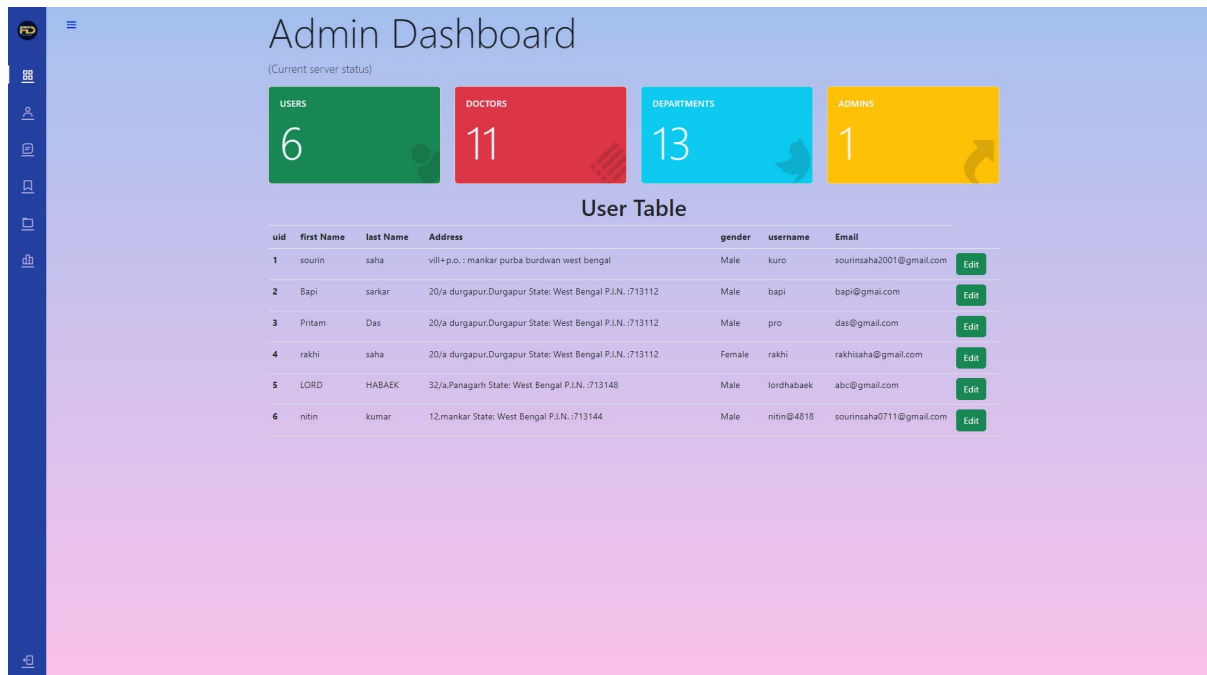
It executes a database query that retrieves the count of doctors, users, departments, and admins from their respective tables. The 'UNION ALL' operator combines the results of the four queries into a single result set.

Inside the callback function of the first query, it checks for any error in executing the query and logs the error if one occurs.

After retrieving the counts, it executes another database query to select all rows from the 'user' table.

Inside the callback function of the second query, it checks for any error in executing the query and logs the error if one occurs.

If both queries are successful, it renders the 'adminUser.ejs' file located in the '/public/admin' directory, passing the count data and user data as variables to the template engine. It also sets the title of the page to "User Table".



The screenshot shows the Admin Dashboard with a sidebar on the left containing icons for home, users, doctors, departments, and admins. The main content area has a header 'Admin Dashboard' with a sub-header '(Current server status)'. Below this are four colored boxes representing counts: Users (6), Doctors (11), Departments (13), and Admins (1). The 'User Table' is displayed below these boxes, showing a list of users with columns for uid, first Name, last Name, Address, gender, username, and Email. Each row has an 'Edit' button.

uid	first Name	last Name	Address	gender	username	Email
1	sourin	saha	vill+p.o.: mankar purba burdwan west bengal	Male	kuro	sourinsaha2001@gmail.com
2	Bapi	sarkar	20/a durgapur,Durgapur State: West Bengal P.I.N.:713112	Male	bapi	bapi@gmail.com
3	Pritam	Das	20/a durgapur,Durgapur State: West Bengal P.I.N.:713112	Male	pro	das@gmail.com
4	rakhi	saha	20/a durgapur,Durgapur State: West Bengal P.I.N.:713112	Female	rakhi	rakhisaha@gmail.com
5	LORD	HABAEK	32/a.Panagarh State: West Bengal P.I.N.:713148	Male	lordhabaek	abc@gmail.com
6	nitin	kumar	12.mankar State: West Bengal P.I.N.:713144	Male	nitin@4518	sourinsaha0711@gmail.com

Doctor management system

This code snippet represents server-side route handlers for managing doctors in the admin dashboard. Let's break down the code and understand its functionality:

1. ``app.get('/admin/doctor', function(req, res, next) {...})``: This route handler is responsible for rendering the doctor table page in the admin dashboard. Here's how it works:

- It first checks if the ``adminToken`` variable is not true, indicating that the admin is not logged in. If the admin is not logged in, it redirects them to the `'/admin_login'` page.
- If the ``adminToken`` variable is true, it proceeds to execute the code inside the `'else'` block.
- It executes a database query to retrieve the counts of doctors, users, departments, and admins from their respective tables, similar to the previous route handler.
- Inside the callback function of the first query, it checks for any error in executing the query and logs the error if one occurs.
- After retrieving the counts, it executes another database query to select all rows from the `'doctor'` table and the corresponding department information using a join operation.
- Inside the callback function of the second query, it checks for any error in executing the query and logs the error if one occurs.
- If both queries are successful, it renders the `'adminDoctor.ejs'` file located in the `'/public/admin'` directory, passing the count data and doctor data as variables to the template engine. It also sets the title of the page to "Doctor's Table".

2. ``app.get('/admin/addDoctors', function(req, res, next) {...})``: This route handler is responsible for rendering the page to add doctors in the admin dashboard. Here's how it works:

- It performs the same check for the ``adminToken`` variable as in the previous route handler to ensure that the admin is authenticated.
- It executes the same database query to retrieve the counts of various entities.
- Inside the callback function of the first query, it checks for any error in executing the query and logs the error if one occurs.
- After retrieving the counts, it executes another database query to select all rows from the `'doctor'` table and the corresponding department information using a join operation, similar to the previous route handler.
- Inside the callback function of the second query, it checks for any error in executing the query and logs the error if one occurs.
- If both queries are successful, it renders the `'adminAddDoctor.ejs'` file located in the `'/public/admin'` directory, passing the count data and doctor data as variables to the template engine. It also sets the title of the page to "Doctor's Table".

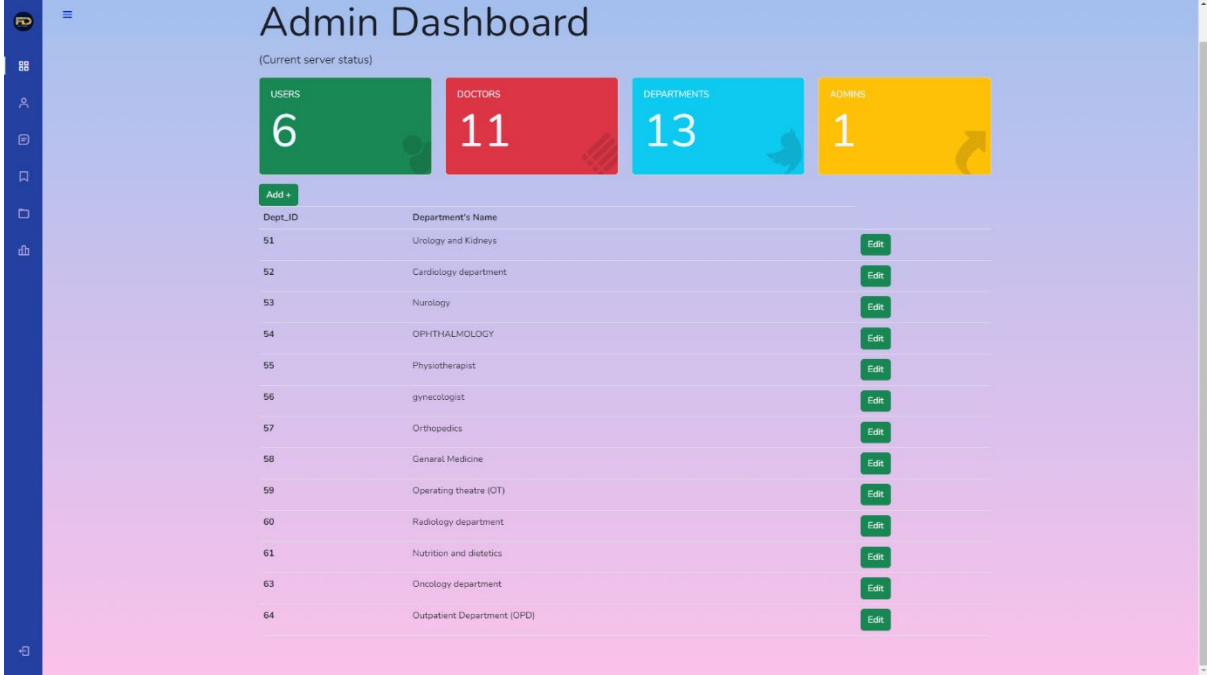
3. `app.post('/admin/addDoctors',function(req,res,next) {...})`: This route handler is responsible for handling the HTTP POST request to add a new doctor. Here's how it works:

- It retrieves the necessary data from the request body, such as the doctor's name, hospital, degree, department ID, and rating.
- It executes a database query to insert the new doctor's information into the 'doctor' table.
- Inside the callback function, it checks for any error in executing the query and logs the error if one occurs.
- If the query is successful, it redirects the admin back to the '/admin/addDoctors' page.

Overall, these route handlers provide functionality for viewing the doctor table, adding new doctors, and rendering the respective admin dashboard pages.

Department management system

```
app.get('/admin/department', function(req,res){
```



The screenshot displays the Admin Dashboard with the following components:

- Header:** "Admin Dashboard" with a subtitle "(Current server status)".
- Summary Cards:** Four colored cards showing counts: USERS (6), DOCTORS (11), DEPARTMENTS (13), and ADMINS (1).
- Table:** A table with columns "Dept_ID", "Department's Name", and "Edit". It lists 13 departments, each with an "Edit" button.

```

    if (adminToken != true) {
      res.redirect('/admin_login');
    }
    else{

```

```

        database.query('Select count( `doc_id` ) as number FROM doctor UNION ALL
SELECT count( user_id ) FROM user UNION all SELECT COUNT( dept_id ) FROM department
UNION all SELECT COUNT( Admin_id ) from admin',function(err , result){
    if (err) {
        console.log(err);
    }
    database.query('select * from department;',function(err,rows){
        if(err){
            console.log('error', err);
        }

        res.render(__dirname+'/public/admin/adminDetartment.ejs',{ amount : result ,
data:rows , title:"Department's Table"});
    });

    });}

});

app.get('/admin/addDepartment', function(req,res){
    if (adminToken != true) {
        res.redirect('/admin_login');
    }
    else{
        database.query('Select count( `doc_id` ) as number FROM doctor UNION ALL
SELECT count( user_id ) FROM user UNION all SELECT COUNT( dept_id ) FROM department
UNION all SELECT COUNT( Admin_id ) from admin',function(err , result){
            if (err) {
                console.log(err);
            }
            database.query('select * from department;',function(err,rows){
                if(err){
                    console.log('error', err);
                }

                res.render(__dirname+'/public/admin/adminAddDepartment.ejs',{ amount : result ,
data:rows , title:"Department's Table"});
            });

            });}

});

app.post('/admin/addDepartment',function(req,res,next){
    var dept_name = req.body.dept_name;
    var dept_about = req.body.dept_about;
    database.query('INSERT INTO `department` (`dept_name`, `dept_about`) VALUES
(?,?)',[dept_name,dept_about],function(err,rows){
        if (err) {
            console.log(err);

```

```

    }
    res.redirect('/admin/addDepartment');
  });
});

```

1. ``app.get('/admin/department', function(req, res) {...})``: This route handler is responsible for rendering the department table page in the admin dashboard. Here's how it works:

- It first checks if the ``adminToken`` variable is not true, indicating that the admin is not logged in. If the admin is not logged in, it redirects them to the `'/admin_login'` page.
- If the ``adminToken`` variable is true, it proceeds to execute the code inside the `'else'` block.
- It executes a database query to retrieve the counts of doctors, users, departments, and admins from their respective tables, similar to the previous route handlers.
- Inside the callback function of the first query, it checks for any error in executing the query and logs the error if one occurs.
- After retrieving the counts, it executes another database query to select all rows from the `'department'` table.
- Inside the callback function of the second query, it checks for any error in executing the query and logs the error if one occurs.
- If both queries are successful, it renders the `'adminDetartment.ejs'` file located in the `'/public/admin'` directory, passing the count data and department data as variables to the template engine. It also sets the title of the page to "Department's Table".

2. ``app.get('/admin/addDepartment', function(req, res) {...})``: This route handler is responsible for rendering the page to add departments in the admin dashboard. Here's how it works:

- It performs the same check for the ``adminToken`` variable as in the previous route handler to ensure that the admin is authenticated.
- It executes the same database query to retrieve the counts of various entities.
- Inside the callback function of the first query, it checks for any error in executing the query and logs the error if one occurs.
- After retrieving the counts, it executes another database query to select all rows from the `'department'` table, similar to the previous route handler.
- Inside the callback function of the second query, it checks for any error in executing the query and logs the error if one occurs.
- If both queries are successful, it renders the `'adminAddDepartment.ejs'` file located in the `'/public/admin'` directory, passing the count data and department data as variables to the template engine. It also sets the title of the page to "Department's Table".

3. `app.post('/admin/addDepartment',function(req,res,next) {...})`: This route handler is responsible for handling the HTTP POST request to add a new department. Here's how it works:

- It retrieves the necessary data from the request body, such as the department name and department about.
- It executes a database query to insert the new department's information into the 'department' table.
- Inside the callback function, it checks for any error in executing the query and logs the error if one occurs.
- If the query is successful, it redirects the admin back to the '/admin/addDepartment' page.

Overall, these route handlers provide functionality for viewing the department table, adding new departments, and rendering the respective admin dashboard pages.

Admin management system

```
app.get('/admin/ceo', function(req,res){
  if (adminToken != true) {
    res.redirect('/admin_login');
  }
  else{
    database.query('Select count( `doc_id` ) as number FROM doctor UNION ALL
SELECT count( user_id ) FROM user UNION all SELECT COUNT( dept_id ) FROM department
UNION all SELECT COUNT( Admin_id ) from admin',function(err , result){
      if (err) {
        console.log(err);
      }
      database.query('select * from admin;',function(err,rows){
        if(err){
          console.log('error', err);
        }

        res.render(__dirname+'/public/admin/adminceo.ejs',{ amount : result , data:rows ,
title:"Admin's Table"});
      });

    });
  }
});

app.get('/admin/addceo', function(req,res){
  if (adminToken != true) {
    res.redirect('/admin_login');
  }
}
```

```

else{
  database.query('Select count( `doc_id` ) as number FROM doctor UNION ALL
SELECT count( user_id ) FROM user UNION all SELECT COUNT( dept_id ) FROM department
UNION all SELECT COUNT( Admin_id ) from admin',function(err , result){
  if (err) {
    console.log(err);
  }
  database.query('select * from admin;',function(err,rows){
    if(err){
      console.log('error', err);
    }

    res.render(__dirname+'/public/admin/adminAddceo.ejs',{ amount : result , data:rows ,
title:"admin's Table"});
  });

  });}

});

app.post('/admin/addceo',function(req,res,next){
  var admin_name = req.body.admin_name;
  var role = req.body.role;
  var email = req.body.admin_email;
  var password = generateRandomint(8);
  database.query('INSERT INTO `admin` ( `Admin_name`, `Admin_pass`, `role`) VALUES (
?, ?, ?)',[admin_name,password,role],function(err,rows){
    if (err) {
      console.log(err);
    }
    var url = '<a href="http://localhost:3000/admin"> click here</a>'
    var msg ="hi "+admin_name+" ,\n \t your adminnistrial account has been created your
name is your login id i.e.: "+admin_name+" and your password is :"+password+" , \n don't
shearer this detail with anyone " + url;
    mailing(email , "hallo new user" , msg);
    res.redirect('/admin/addceo');
  });
});

```

1. `/admin/ceo` - GET route:

- Checks if the adminToken is true, otherwise redirects to the admin login page.
- Queries the database to retrieve the count of doctors, users, departments, and admins.
- Retrieves data from the `admin` table and renders the `adminceo.ejs` view, passing the count and data.

2. `/admin/addceo` - GET route:

- Checks if the adminToken is true, otherwise redirects to the admin login page.

- Queries the database to retrieve the count of doctors, users, departments, and admins.
- Retrieves data from the `admin` table and renders the `adminAddceo.ejs` view, passing the count and data.

3. `/admin/addceo` - POST route:

- Handles the form submission when adding a new CEO.
- Retrieves the values from the request body, including `admin_name`, `role`, and `admin_email`.
- Generates a random password using `generateRandomint` function.
- Inserts a new row into the `admin` table with the provided values.
- Sends an email to the provided admin_email address with the account details (name and password).
- Redirects to the `/admin/addceo` page.

Please note that there are some missing parts in the code you provided, such as the `generateRandomint` function and the implementation of the `mailing` function for sending emails. You would need to define and implement those functions separately.

Additionally, make sure to handle error cases and sanitize user input to prevent security vulnerabilities like SQL injections.

How to initiate the server in your machine

To start the server for your project, you can follow these steps:

1. Open the terminal or command prompt on your computer.
2. Navigate to the root directory of your project using the 'cd' command.
3. Once you are in the project directory, type the command 'npm test' and press Enter.
4. This command will trigger the test script defined in your package.json file.
5. The test script will execute the necessary setup and configuration to start the server.
6. It will run the automated tests included in your project to ensure that the server is functioning correctly.
7. If all the tests pass successfully, the server will start, and you will see the relevant output or logs in the terminal.
8. At this point, your server is up and running, ready to handle incoming requests and serve the application.

Note: Ensure that you have installed Node.js and the required dependencies for your project by writing `npm install` before running the 'npm test' command.

FUTURE PLANS

1. **Integration with Healthcare Systems:** Explore the possibility of integrating your online hospital management system with existing healthcare systems, such as electronic medical record (EMR) systems or laboratory management systems. This integration can streamline data exchange, improve efficiency, and provide a comprehensive solution for healthcare providers.

2. **Advanced Reporting and Analytics:** Enhance the reporting capabilities of your system by implementing advanced analytics and visualization features. This will enable users to generate insightful reports, track key performance indicators, and make data-driven decisions for better patient care and resource management.

3. **Telemedicine and Remote Patient Monitoring:** Investigate the integration of telemedicine capabilities into your system, allowing healthcare providers to offer virtual consultations and remote patient monitoring. This can expand access to healthcare services, especially in remote areas, and provide convenience for patients.

4. **AI-powered Decision Support:** Consider incorporating artificial intelligence (AI) algorithms and machine learning models into your system to provide decision support for healthcare professionals. This can include intelligent diagnosis assistance, predictive analytics for patient outcomes, or personalized treatment recommendations.

5. **Internet of Things (IoT) Integration:** Explore opportunities to leverage IoT devices for healthcare monitoring and data collection. Integrating IoT sensors, wearables, or smart medical devices with your system can provide real-time patient data, improve monitoring capabilities, and enhance preventive healthcare.

6. **Enhanced Patient Engagement:** Focus on improving patient engagement by implementing features like appointment reminders, online patient portals for accessing medical records, secure messaging between patients and healthcare providers, and educational resources for self-care and health management.

7. **Regulatory Compliance:** Stay up to date with healthcare regulations and standards to ensure compliance with data privacy laws (such as HIPAA), interoperability standards (such as HL7), and other relevant regulations. Continuously monitor and update your system to meet evolving regulatory requirements.

8. Cloud Migration and Scalability: Evaluate the feasibility of migrating your system to a cloud-based infrastructure to enhance scalability, flexibility, and availability. Cloud services can also provide robust data storage, disaster recovery, and easy access to resources for future expansion.

9. Continuous Improvement and Bug Fixing: Establish processes for ongoing maintenance, bug fixing, and performance optimization. Regularly analyze user feedback, monitor system performance, and prioritize enhancements to address user needs and ensure the system remains stable and reliable.

10. Collaboration and Partnerships: Seek opportunities for collaboration with healthcare organizations, medical professionals, or technology providers to further enhance your system's capabilities, gain insights into industry trends, and expand your user base.

REFERENCE AND RESOURCES

Stack Overflow (<https://stackoverflow.com>): An online community of developers where you can find solutions to coding problems and answers to technical questions.

Mozilla Developer Network (<https://developer.mozilla.org>): A comprehensive resource for web development, including documentation on HTML, CSS, JavaScript, and related technologies.

W3Schools (<https://www.w3schools.com>): A popular online learning platform that provides tutorials and references for web development technologies.

GitHub (<https://github.com>): A platform for version control and collaboration, where you can find open-source projects, code samples, and documentation.

NodeJS ([Documentation | Node.js \(nodejs.org\)](#)) for understanding the packages

Wikipedia ([Wikipedia](#)) for designing documentation

ChatGPT([College Project Acknowledgement \(openai.com\)](#)) for finding errors and bugs in code

codePen([Trending - CodePen](#)) for front end development

youtube chennels such as [\(30\) CodeWithHarry - YouTube](#) [\(30\) Thapa Technical - YouTube](#) [Tutorials Website - YouTube](#) etc

GeekforGeeks ([GeeksforGeeks | A computer science portal for geeks](#)) for data handling

Pexels ([Free Stock Photos, Royalty Free Stock Images & Copyright Free Pictures · Pexels](#)) for require images.

THANK
YOU