National University of Computer and Emerging Sciences

Week 10 Node.JS and Express.JS

for

Web Programming

Spring 2024

FAST School of Computer Science

Today's Agenda

Module 1: Upload file using Multer

Module 2: Nodemailer

Module 3: SSR with EJS

Module 4: Authentication and JWT Authentication

Module 1: Upload file using Multer

Multer is a middleware for handling multipart/form-data, which is primarily used for uploading files in Node.js web applications. It is specifically designed for handling file uploads in forms where the enctype is set to "multipart/form-data".

Here's how Multer works:

File Upload Handling: Multer parses the incoming form data containing files and extracts them. It then stores the files in a designated location on the server's disk or in memory, depending on the configuration.

Integration with Express: Multer seamlessly integrates with Express.js, which is a popular web application framework for Node.js. It provides middleware functions that can be easily mounted in the Express application to handle file uploads.

Configuration Options: Multer offers various configuration options to customize file upload behavior, such as specifying the destination directory for storing files, setting file name formats, limiting the file size, and filtering file types.

Error Handling: Multer automatically handles errors related to file uploads, such as exceeding file size limits or unsupported file types. It provides error messages that can be customized to suit the application's requirements.

Asynchronous Processing: Multer supports asynchronous processing of file uploads, allowing developers to perform additional tasks, such as validating uploaded files or processing file contents, before completing the request

Module 2: Nodemailer

The Nodemailer module makes it easy to send emails from your computer.

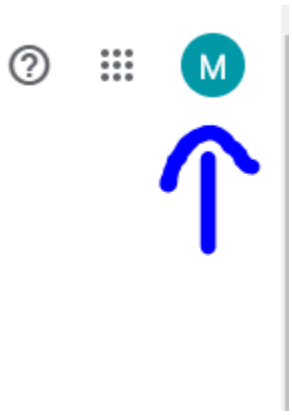
Installation:

npm install nodemailer

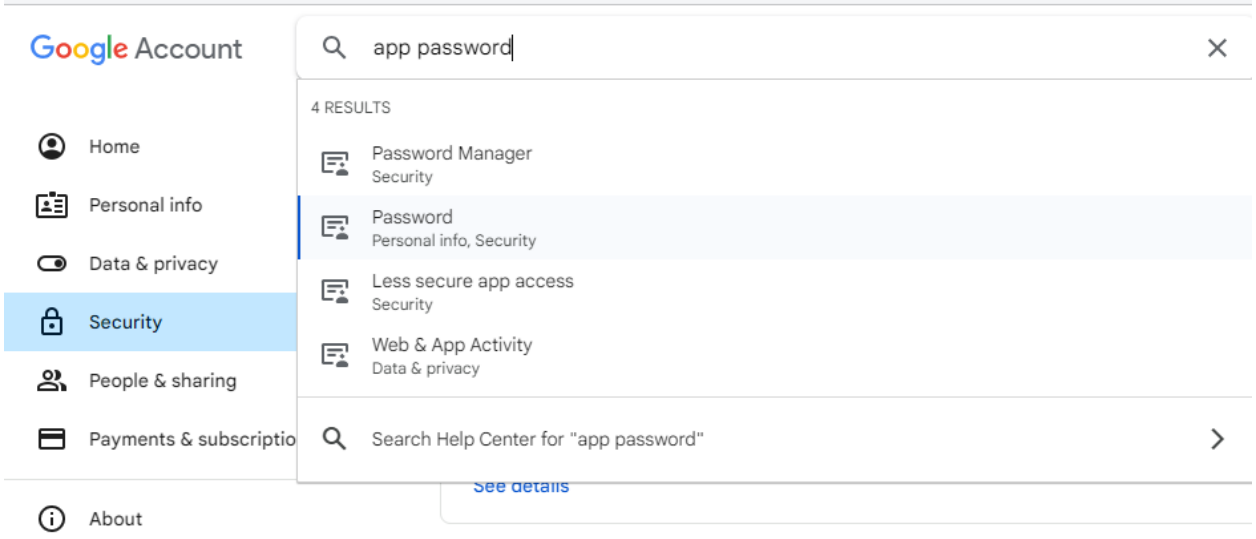
https://www.w3schools.com/nodejs/nodejs_email.asp

To get the less secure password follow the steps

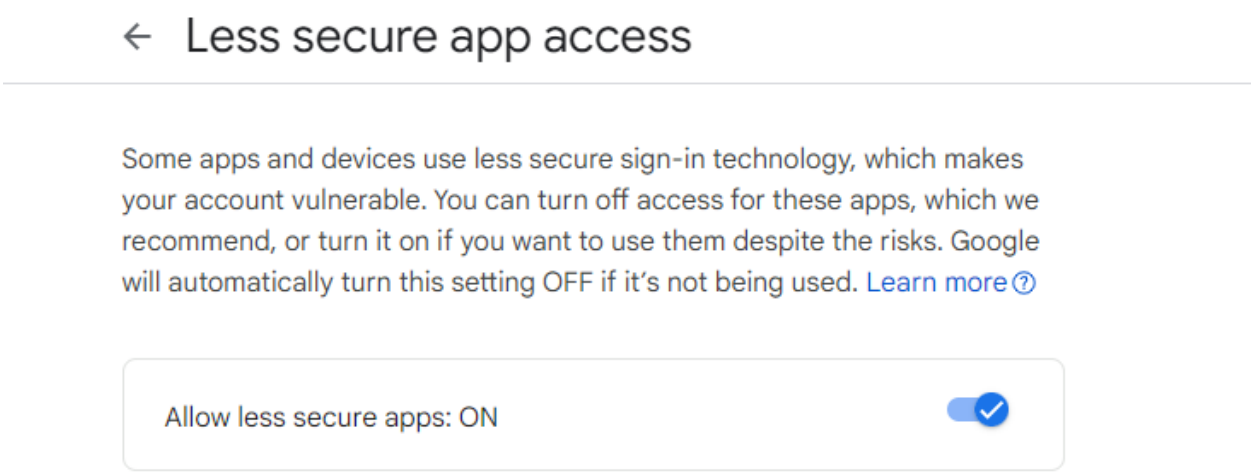
1. Click on profile icon and then click on "Manage your google account"



2. Then select security and search app password



3. Select Less secure app access option and click toggle to on



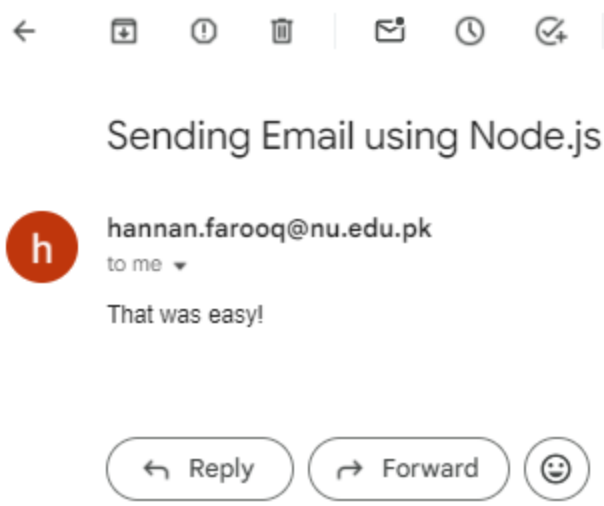
4. Then add this code "use your original password"

```
var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'hannan.farooq@nu.edu.pk',
    pass: 'yourpassword'
  }
});

var mailOptions = {
  from: 'hannan.farooq@nu.edu.pk',
  to: 'hannanfarooq8195@gmail.com',
  subject: 'Sending Email using Node.js',
  text: 'That was easy!'
};

transporter.sendMail(mailOptions, function(error, info){
  if (error) {
    console.log(error);
  } else {
    console.log('Email sent: ' + info.response);
  }
});
```

5. If it works successfully, then the receiver will get the email from you



Module 3: SSR with EJS

What is a template engine?

A template engine in Node.js is a library or framework that helps generate dynamic HTML content by merging data with pre-defined HTML templates. It simplifies the process of generating HTML responses in web applications by allowing developers to create reusable templates and inject dynamic data into them.

Template engines typically provide the following features:

Template Syntax: They define a syntax for embedding dynamic content and logic within HTML templates. This syntax usually includes placeholders, conditionals, loops, and other constructs to generate dynamic content based on data.

Data Binding: Template engines enable data binding, allowing developers to pass data from the application to the templates. This data can come from various sources, such as databases, APIs, or user input.

Template Inheritance: Many template engines support template inheritance, allowing developers to create a hierarchy of templates where child templates can extend or override content from parent templates. This promotes code reusability and maintainability.

Partial Views: Template engines often support partial views or includes, allowing developers to modularize templates by splitting them into smaller components. These components can then be reused across multiple templates.

Helpers and Filters: Some template engines provide helper functions or filters to perform common tasks within templates, such as formatting dates, manipulating strings, or generating HTML elements.

Integration with Node.js Frameworks: Template engines are designed to work seamlessly with Node.js web frameworks like Express.js. They provide middleware or integration mechanisms to render templates and serve dynamic content in response to HTTP requests.

Popular template engines for Node.js include:

Handlebars: Handlebars is a simple and lightweight template engine that uses a Mustache-like syntax for templating.

https://handlebarsjs.com/

EJS (Embedded JavaScript): EJS allows developers to embed JavaScript code directly within HTML templates, making it easy to generate dynamic content.

https://ejs.co/

Installation:

Go to https://www.digitalocean.com/community/tutorials/how-to-use-ejs-to-template-your-node-application

- npm i ejs
- ejs rendering
- ejs dynamic data
- ejs partials

Pug (formerly Jade): Pug is a concise and expressive template engine with a significant whitespace syntax. It provides a clean and elegant way to write HTML templates.

https://pugjs.org/api/getting-started.html

Nunjucks: Nunjucks is a powerful template engine inspired by Jinja2, which is used in the Flask framework for Python. It offers features like template inheritance, macros, and asynchronous rendering.

These template engines offer different features and syntaxes, so developers can choose the one that best fits their project requirements and personal preferences.

Module 4: Authentication and JWT Authentication

Authentication in Node.js refers to the process of verifying the identity of a user or entity accessing a system or application. It's a critical aspect of building secure web applications, ensuring that only authorized users can access protected resources or perform certain actions.

Authentication typically involves validating a user's credentials, such as username/password, tokens, or other authentication factors, against a stored record in a database or an external authentication provider. Once the user's identity is confirmed, the application can grant access and provide appropriate permissions.

Here's a basic overview of how authentication works in Node.js:

User Registration: Users create an account by providing necessary information (e.g., username, email, password). The application securely stores the user's credentials, often by hashing and salting passwords to protect them from unauthorized access.

User Login: When a user attempts to log in, they provide their credentials through a login form. The application validates the provided credentials against the stored data. If the credentials match, the user is considered authenticated.

Session Management: Upon successful authentication, the application creates a session for the user, usually by generating a unique session identifier (session ID) and storing it in a session store (e.g., memory, database). This session ID is often associated with the user's identity and is used to maintain the user's authentication state throughout their interaction with the application.

Authorization: After authentication, the application determines what actions the authenticated user is allowed to perform based on their assigned permissions or roles. Authorization ensures that authenticated users can only access resources or perform actions that they are authorized to.

Node.js provides various libraries and frameworks for implementing authentication, including:

Passport.js: A popular authentication middleware for Node.js that supports multiple authentication strategies (e.g., username/password, OAuth, JWT).

jsonwebtoken (JWT): A library for creating and verifying JSON Web Tokens (JWTs), which are commonly used for stateless authentication and authorization in web applications.

OAuth: A protocol for delegated authorization, allowing users to authenticate using third-party identity providers (e.g., Google, Facebook) without sharing their credentials with the application.

Express.js Middleware: Custom middleware functions can be used in Express.js to implement authentication logic, such as verifying session tokens or API keys.

Implementing robust authentication in Node.js involves considering various security concerns, such as password hashing, protection against brute force attacks, session management, and securing communication channels. Additionally, developers should stay informed about common security vulnerabilities and best practices for building secure authentication systems.

Types of authentications

Stateful	**Stateless**
1. Session-based: Stateful authentication relies on maintaining session state on the server. When a user logs in, a session is created and stored on the server. The server associates the session with the user's identity and maintains session data until the user logs out or the session expires.	**1. Token-based:** Stateless authentication uses tokens to authenticate users. When a user logs in, the server generates a token (e.g., JSON Web Token or JWT) containing the user's identity and any necessary metadata. The token is then sent to the client, which includes it in subsequent requests to authenticate itself.
2. Server-side storage: This allows the server to track the user's authentication state and session information, such as user identity, session expiration time, and other session-related data.	**2. No server-side session storage:** Unlike stateful authentication, stateless authentication does not rely on server-side session storage. The server does not maintain any session state for authenticated users. Instead, all necessary authentication information is contained within the tokens themselves.
3. Cookies: Sessions are often managed using cookies. Upon successful authentication, the server sends a session	**3. Client-side token management:** Clients are responsible for managing authentication tokens. They store tokens locally (e.g., in browser

identifier (session ID) to the client as a cookie. The client includes this session ID in subsequent requests to identify itself to the server.	storage or mobile app storage) and include them in requests as needed. Tokens are typically sent in the HTTP headers (e.g., Authorization header) or as URL parameters.
4. Scalability challenges Stateful authentication can pose scalability challenges, especially in distributed systems or when deploying applications across multiple servers. Load balancing and session affinity mechanisms may be required to ensure that subsequent requests from the same user are routed to the same server that holds the user's session data.	**4. Scalability:** Stateless authentication is more scalable than stateful authentication because it does not require server-side session storage. Each request is self-contained, and servers can handle requests independently without needing to share session state.
5. Server-side session management The server is responsible for managing session creation, validation, and expiration. It may also handle tasks such as session timeout, session termination, and session cleanup.	**5. Stateless servers:** Stateless servers do not need to maintain session state or track user sessions. This simplifies server-side architecture and makes it easier to horizontally scale the application by adding more server instances.

- https://www.npmjs.com/package/uuid
- npm i uuid