

MODULE NAME: BACKEND APPLICATION DEVELOPMENT

Competence: Develop a Backend Application using Node Js

LO1: Develop RESTFUL APIs with Node JS.

Indicative Content 1.1: Setup Node. Js Environment

1.1.1: Description of Node.js Key Concepts

❖ Node.Js:

Node.js is an open-source JavaScript runtime environment that allows developers to execute JavaScript code on the server side outside of a browser.

Runtime environment is the environment in which a program or application is executed. It provides an event-driven, non-blocking I/O model, making it highly efficient for building scalable and real-time applications.

❖ **Routing refers to determining how an application responds to a client request to a particular endpoint.**

Routes in web development define how an application responds to specific HTTP requests. In Node.js, routes are used to map URLs to specific functions or controllers, enabling the server to handle different requests and serve appropriate responses.

❖ **NPM:** Is default package **manager for Node.js**

Npm stands for **N**ode **P**ackage **M**anager.

It is used to install, manage, and distribute packages and libraries written in JavaScript. NPM simplifies the process of including external dependencies in your Node.js projects.

❖ Express Js

Express.js is a popular web application framework for Node.js. It simplifies the process of building robust, scalable, and maintainable web applications by providing a set of essential features and middleware for handling HTTP requests and routes.

Express.js, or simply Express, is a back end web application framework for building RESTful APIs with Node.js.

RESTful(Representational State Transfer)

APIs(Application Programming Interface)

❖ BACKEND APPLICATION

The backend refers to parts of a computer application or a program's code that allow it to operate and that cannot be accessed by a user.

A backend application is the server-side component of a web application responsible for processing requests, interacting with databases, and serving data or HTML content to the client-side (frontend) application.

❖ **Class**

A class is a blueprint for declaring and creating objects

- ❖ **Object** is a class instance that allows programmers to use variables and methods from inside the class.

Objects can represent real-world entities and encapsulate both data (properties) and behavior (methods) related to those entities.

❖ **Method:**

Method definition is a shorter syntax for **defining** a **function** property in an object initializer. It can also be used in classes

- ❖ **Properties** : Properties in JavaScript refer to the characteristics or attributes of an object. These are the values associated with an object that describe its state or characteristics.
- ❖ **Dependencies**

Dependencies are external packages or libraries that a Node.js application relies on to perform various tasks. Developers specify these dependencies in the project's package.json file, and NPM is used to install and manage them.

❖ **APIs**

APIs are sets of rules and protocols that allow different software applications to communicate and interact with each other. In web development, APIs are often used to enable data exchange between a frontend and a backend application.

❖ **Postman**

- ✓ Postman is a popular tool for testing and documenting APIs.
- ✓ It provides a user-friendly interface for sending HTTP requests to APIs, inspecting responses, and automating API testing.

❖ **Nodemon**

Nodemon is a utility tool for Node.js that helps developers during the development process.

It automatically monitors changes in your Node.js application and restarts the server when code changes are detected, making development more efficient.

❖ DBMS (SQL Based, NoSQL Based)

DBMS refers to software that manages and interacts with databases. In the context of web development, DBMS can be categorized into SQL-based (relational databases like MySQL, PostgreSQL) and NoSQL-based (non-relational databases like MongoDB) systems, each with its own data storage and retrieval mechanisms. These databases are commonly used for storing and managing application data.

1.1.2: Installation of Node Js Modules and packages

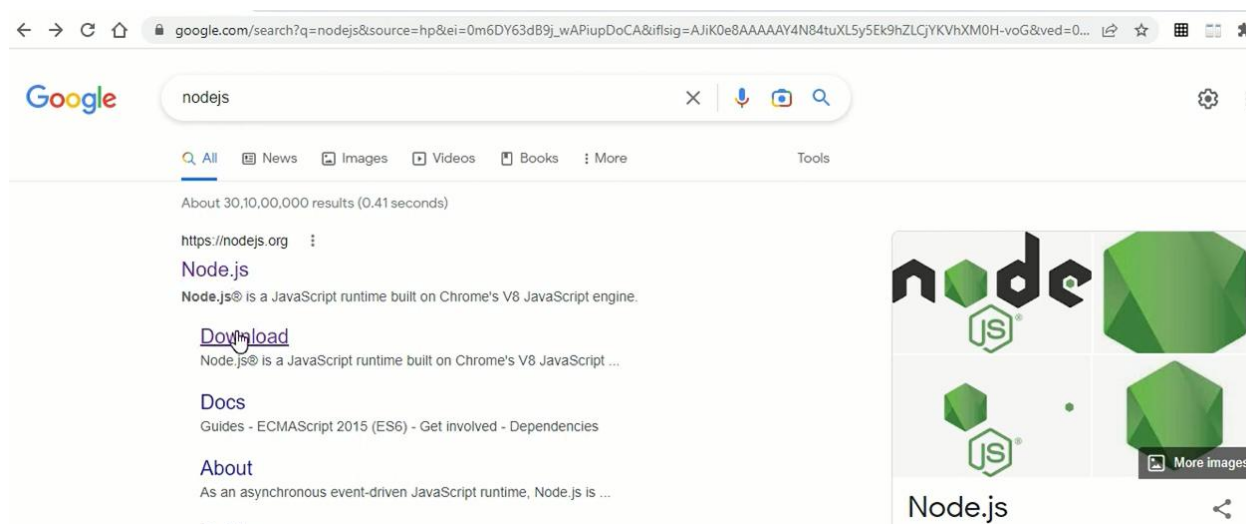
❖ Node.Js and NPM

Step1: First check if there is node.js in your computer by writing in command prompt. Node -v. and press enter

```
Command Prompt
Microsoft Windows [Version 10.0.19045.3086]
(c) Microsoft Corporation. All rights reserved.
C:\Users\HI>node -v

'node' is not recognized as an internal or external command,
operable program or batch file.
```

STEP2: Go on browser and search nodejs and download it



STEP3:
Choose

nodejs.org/en/download/


Downloads


Latest LTS Version: 18.12.1 (includes npm 8.19.2)


Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS
Recommended For Most Users

Current
Latest Features


Windows Installer
node-v18.12.1-x64.msi


macOS Installer
node-v18.12.1.pkg


Source Code
node-v18.12.1.tar.gz

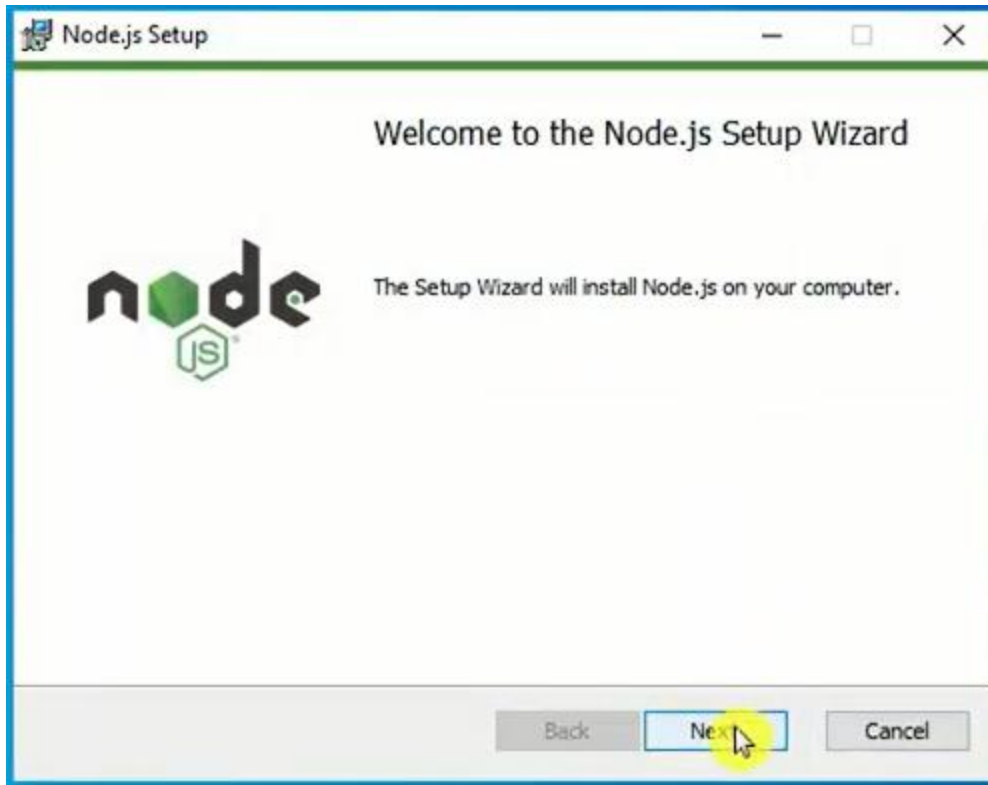
| | | |
|--------------------------|----------------|--------|
| Windows Installer (.msi) | 32-bit | 64-bit |
| Windows Binary (.zip) | 32-bit | 64-bit |
| macOS Installer (.pkg) | 64-bit / ARM64 | |
| macOS Binary (.tar.gz) | 64-bit | ARM64 |
| Linux Binaries (x64) | 64-bit | |

64 bit

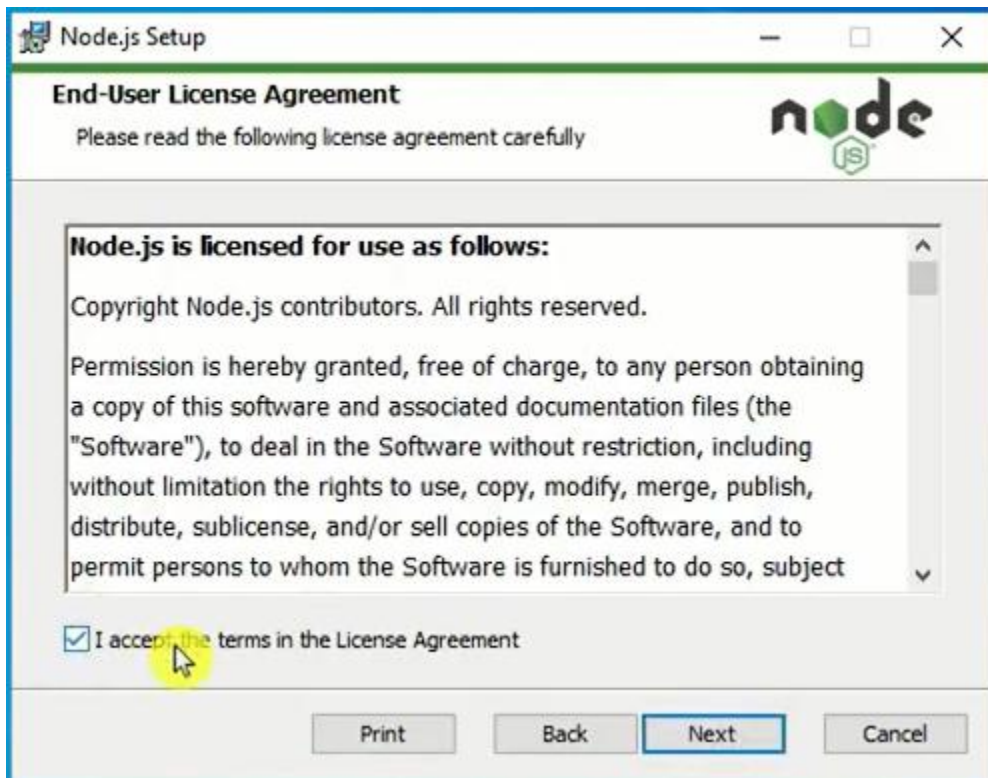
STEP4: Click on Run



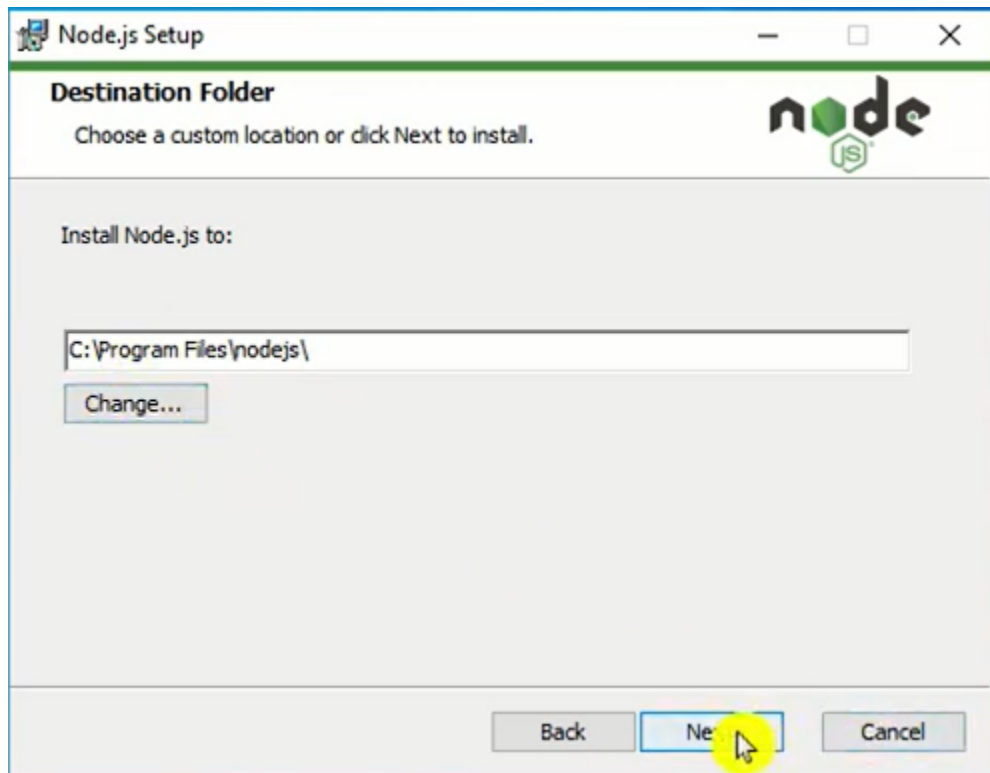
STEP5. Click on Next



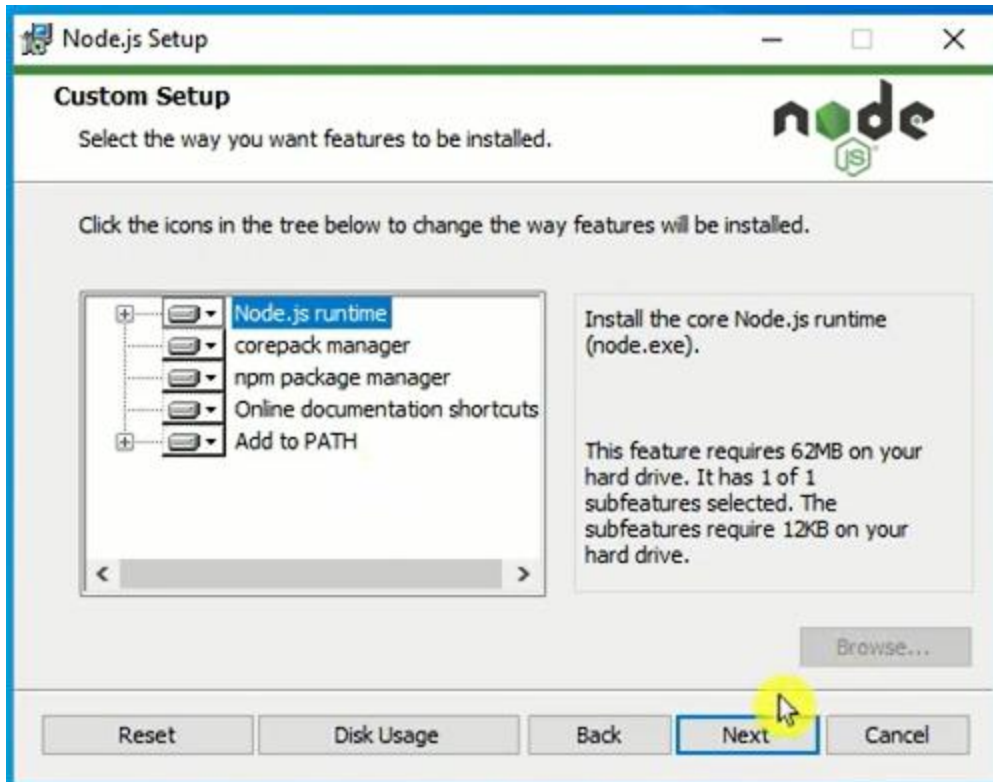
Step6: Accept the license agreement and Next



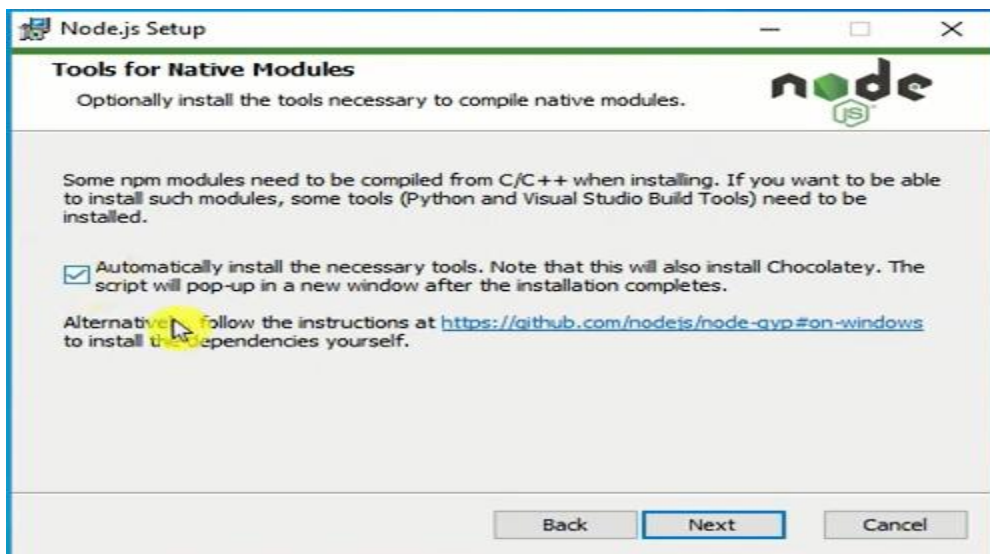
Step7: Click Next



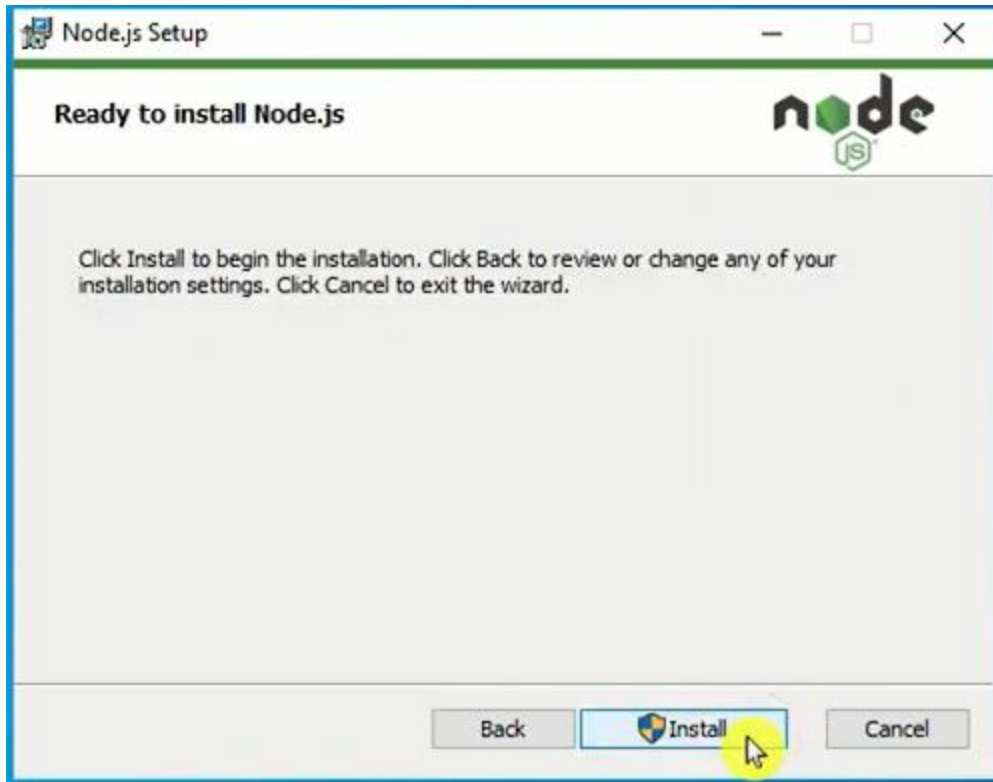
Step8: Click Next



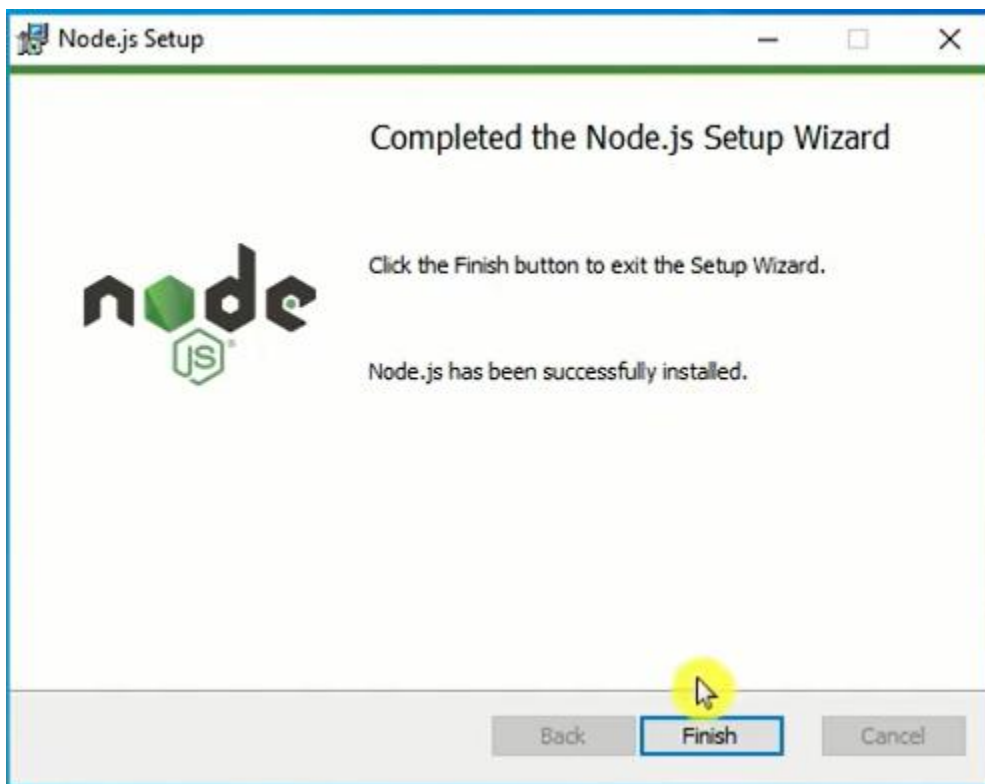
Step9: check the automatically install the necessary tools and click Next



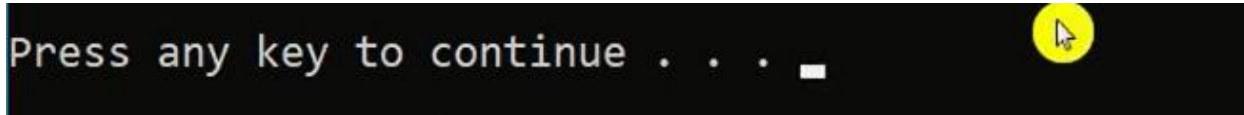
Step10: Click on Install to begin the installation



Step 11: Click on Finish button to exit the setup wizard



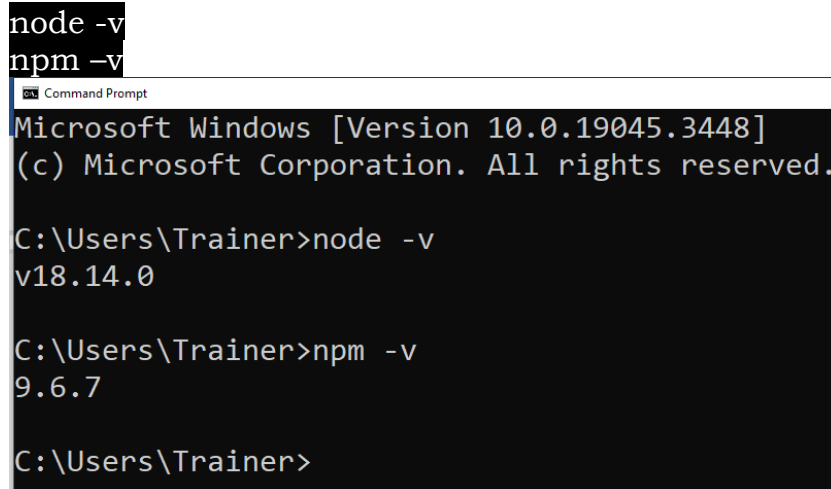
Step12: Press any key to continue and again



Summary: Node.js and NPM Installation:

- ✓ Go to the official Node.js website (<https://nodejs.org/>).
- ✓ Download the appropriate installer for your operating system (e.g., Windows Installer, macOS Installer, or Linux Binaries).
- ✓ Run the installer and follow the installation instructions.
- ✓ After installation, open your terminal or command prompt and verify the installation by running these commands.

```
node -v
npm -v
```



```
Microsoft Windows [Version 10.0.19045.3448]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Trainer>node -v
v18.14.0

C:\Users\Trainer>npm -v
9.6.7

C:\Users\Trainer>
```

These commands should display the installed Node.js and NPM versions as shown on that GUI.

Express.js Installation:

- ✓ Go on: <https://expressjs.com/>
- ✓ Create a new directory for your Express.js project (if you haven't already).
- ✓ Navigate to your project directory using the terminal.
- ✓ Initialize a new Node.js project by running:

```
npm init -y
```

Install Express.js as a project dependency using NPM:

```
npm install express --save
```

This command will install Express.js and add it to your project's package.json file.

Postman Installation:

- ✓ Postman is not installed via NPM; it's a separate desktop application.
- ✓ Go to the Postman website (<https://www.postman.com/>).
- ✓ Download the Postman app for your operating system and install it.

After installation of postman desktop application don't forget to create the user account

Nodemon Installation:

- ✓ Nodemon is typically installed globally, so you can use it across different Node.js projects.
- ✓ Open your terminal or command prompt and run the following command to install Nodemon globally:

```
npm install -g nodemon
```

The -g flag indicates a global installation.

Now, you should have Node.js and NPM, Express.js, Postman, and Nodemon installed on your system.

Note: For Express.js, you might need to create a basic Express application in your project by writing code. Express is a framework that is used within your Node.js projects, so it doesn't have a separate executable or installation process like Postman or Nodemon.

Configuration of MySQL Server

To configure MySQL Server on Windows, follow these steps:

- ✓ After the installation, the MySQL Installer will prompt you to configure the MySQL Server. Click "Next" to begin the configuration.
- ✓ Choose the appropriate configuration type. "**Standalone MySQL Server**" is suitable for most users.
- ✓ Set up the server configuration details:
- ✓ **Config Type:** Select "Development Machine", "Server Machine", or "Dedicated Machine" based on your use case.
- ✓ **Connectivity:** Choose the port number (default is 3306) and enable TCP/IP networking.
- ✓ **Authentication:** Choose the authentication method (MySQL 8.0's strong password encryption is recommended).
- ✓ **Accounts and Roles:** Set a root password and create any additional user accounts as needed.
- ✓ **Windows Service:** Configure MySQL Server to run as a Windows service. You can also choose to start the server at system start up.
- ✓ **Apply configuration**
- ✓ Click "Next" to review the configuration settings, and then click "Execute" to apply the configuration.

3. Complete installation

Once the configuration is applied, the installer will show a summary of the completed tasks. Click "Finish" to complete the installation process.

4. Verify installation

Open a Command Prompt window and type **mysql -u root -p** to log in to the MySQL server using the root account. Enter the root password you set during configuration to access the MySQL shell and verify that the server is running correctly.

Indicative content 1.2: Connection of Node Js to the ES5 or ES6 server

1. Categories and Usability of Node.js Client Libraries

HTTP Client Libraries:

Axios: A promise-based HTTP client for making requests to APIs. It works in both Node.js and browser environments.

node-fetch: A lightweight library that brings the window.fetch function to Node.js, useful for making HTTP requests.

SuperAgent: A flexible and straightforward HTTP client for making REST API requests.

Database Client Libraries:

Mongoose: An object data modeling (ODM) library for MongoDB and Node.js, providing schema-based solutions.

pg: A PostgreSQL client for Node.js that allows you to interact with PostgreSQL databases.

mysql2: A MySQL client for Node.js that supports both callbacks and promises.

Sequelize: it supports multiple SQL dialects, including MySQL, PostgreSQL, and SQLite.

Authentication Libraries:

Passport.js: A middleware for authentication in Node.js applications, supporting various strategies like OAuth, JWT, and local login.

Jsonwebtoken (JWT): A library for generating and verifying JSON Web Tokens, commonly used in secure API authentication.

Utility Libraries:

Lodash: A modern JavaScript utility library delivering modularity, performance, and extras. It provides utility functions for common programming tasks.

Moment.js: A library for parsing, validating, manipulating, and formatting dates.

Underscore.js: Another utility library that provides functional helpers for tasks like manipulating arrays, objects, and functions.

Template Engines:

EJS (Embedded JavaScript): A simple templating language that lets you generate HTML markup with plain JavaScript.

Pug: A high-performance template engine heavily influenced by Haml and implemented with JavaScript.

File Handling Libraries:

Multer: A middleware for handling multipart/form-data, primarily used for uploading files.

fs-extra: An extension of the built-in fs module, providing additional methods for file manipulation like copying, moving, and removing directories.

Real-time Communication Libraries:

Socket.io: A library for real-time web applications that enables bi-directional communication between web clients and servers.

ws: A simple, fast, and thoroughly tested WebSocket client and server for Node.js.

Task Automation Libraries:

Gulp A toolkit to automate time-consuming tasks in your development workflow, like minification, compilation, and testing.

Grunt: Another task runner that allows you to automate repetitive tasks such as minification, compilation, and linting.

Testing Libraries:

Mocha: A feature-rich JavaScript test framework running on Node.js, making asynchronous testing simple.

Jest: A JavaScript testing framework with a focus on simplicity, offering a built-in assertion library and a great developer experience.

Chai: A BDD/TDD assertion library for Node.js that can be paired with any testing framework.

Security Libraries:

Helmet: A middleware for Express.js that helps secure your apps by setting various HTTP headers.

bcryptjs: A library for hashing passwords, ensuring secure storage of user credentials.

File System Libraries:

fs-extra: An extension of the Node.js fs module with added methods like copy, move, and remove, making it easier to work with the file system.

glob: A library for finding files matching a specified pattern, useful for tasks like file searching and processing.

HTTP vs. HTTPS

The main difference between HTTP and HTTPS lies in security.

- HTTP (Hypertext Transfer Protocol) is the standard protocol used for transmitting data over the web. However, it does not provide any encryption, meaning that data sent between the user's browser and the website can be intercepted by third parties.

- HTTPS (Hypertext Transfer Protocol Secure), on the other hand, adds a layer of security by using SSL (Secure Sockets Layer) or TLS (Transport Layer Security) to encrypt the data exchanged. This means that information such as passwords, credit card numbers, and personal details are protected from eavesdroppers.

2. Difference between ES5 from ES6

The differences between ES5 (ECMAScript 5) and ES6 (ECMAScript 2015) in JavaScript are significant, as ES6 introduced many new features and improvements to the language. Here are some of the key differences:

1. Variable Declarations

ES5: Uses `var` for variable declarations, which has function scope.

ES6: Introduces `let` and `const`. `let` has block scope, while `const` is used for constants that cannot be reassigned.

2. Arrow Functions

- ES5: Functions are declared using the `function` keyword.

- ES6: Introduces arrow functions (`() => {}`), which provide a shorter syntax and lexically bind the `this` value.

3. Template Literals

- **ES5:** String concatenation is done using the `+` operator.

- **ES6:** Introduces template literals (``Hello, ${name}!``), allowing for easier string interpolation and multi-line strings.

4. **Destructuring Assignment**

- ES5: Assigning values from arrays or objects requires multiple statements.
- ES6: Introduces destructuring, allowing for unpacking values from arrays or properties from objects in a more concise way.

5. **Modules**

- ES5: No built-in module system; developers often use IIFEs or libraries like CommonJS.
- ES6: Introduces a module system with ``import`` and ``export`` keywords for better code organization and reuse.

6. **Classes**

- ES5: Uses constructor functions and prototypes to create objects.
- ES6: Introduces a class syntax, making it easier to create and manage objects and inheritance.

7. **Promises**

- ES5: Asynchronous programming often relies on callbacks, which can lead to "callback hell."
- ES6: Introduces Promises, providing a cleaner way to handle asynchronous operations.

8. **Default Parameters**

- ES5: Requires checking for ``undefined`` to set default values in functions.
- ES6: Allows default parameter values directly in function definitions.

9. **Spread and Rest Operators**

- ES5: Requires methods like ``apply()`` to spread elements of an array.
- ES6: Introduces the spread operator (``...``) to expand arrays and the rest operator to gather arguments into an array.

Creation of basic server with Express Js

Steps to Create a Basic Server with Express.js

1. Install Node.js and NPM

Make sure you have Node.js installed on your machine. NPM (Node Package Manager) is included with Node.js. You can check if Node.js and NPM are installed by running:

```
node -v
```

```
npm -v
```

2. Initialize a New Node.js Project

Create a new directory for your project and initialize a new Node.js project.

```
mkdir my-express-app
```

```
cd my-express-app
```

```
npm init -y or npm init
```

This will create a package.json file with default settings.

The commands ``npm init -y`` and ``npm init`` are both used to create a ``package.json`` file for a Node.js project, but they differ in how they handle the initialization process.

npm init

- When you run ``npm init``, it starts an interactive process that prompts you to answer several questions about your project. These questions include the package name, version, description, entry point, test command, repository, keywords, author, and license.
- You have the opportunity to customize the ``package.json`` file according to your preferences.

npm init -y

- The ``-y`` flag (or ``--yes``) automatically answers "yes" to all the prompts, creating a ``package.json`` file with default values.
- This is useful when you want to quickly set up a project without going through the interactive prompts, especially if you are okay with the default settings.

NB:

- Use ``npm init`` when you want to customize the ``package.json`` file and provide specific details about your project.
- Use ``npm init -y`` when you want to quickly create a ``package.json`` file with default values without any prompts.

3. Install Express.js

Install Express.js using NPM.

```
npm install express --save
```

4. Create the Server File

Create a new file called index.js or app.js in your project directory.

5. Write Basic Express.js Server Code

Open the index.js file and write the following code to create a basic server.

```
// Import Express
const express = require('express');
// Initialize Express
const app = express();
```



```
// Define a Port
const port = 3000;
// Create a Basic Route
app.get('/', (req, res) => {
  res.send('Hello, World!');
});
// Start the Server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

6. Run the Server

To start the server, run the following command in your terminal:

```
node index.js
```

The server will start, and you'll see the message: Server is running on <http://localhost:3000>.

7. Test the Server

Open your browser and go to <http://localhost:3000>. You should see the message Hello, World!.

Explanation of the Code

`const express = require('express');`: Imports the Express module.

`const app = express();`: Creates an instance of Express.

`const port = 3000;`: Defines the port number the server will listen to.

`app.get('/', (req, res) => { ... });`: Sets up a route to handle GET requests to the root URL (/). When someone visits this URL, the server responds with "Hello, World!".

`app.listen(port, () => { ... });`: Starts the server and listens on the defined port. The callback function runs once the server starts, logging a message to the console.

Adding More Routes (Optional)

You can add more routes to handle different requests:

```
// About Route
app.get('/about', (req, res) => {
  res.send('This is the About page.');
```



```
// Contact Route
app.get('/contact', (req, res) => {
  res.send('This is the Contact page.');
```

Middleware and Static Files

You can also use middleware to serve static files or handle specific routes.

```
// Serve Static Files from the 'public' Directory
app.use(express.static('public'));
```

```
// Middleware Example
```

```
app.use((req, res, next) => {
  console.log('A new request received at ' + Date.now());
  next();
});+
```

Establishment and Test of server connection

Steps to Establish and Test a Server Connection

Step 1: Set Up Your Node.js Environment

1. Install Node.js: Ensure you have Node.js installed on your machine. You can download it from [nodejs.org](<https://nodejs.org/>).

2. Create a New Project

```
mkdir my-server-project
cd my-server-project
npm init -y
```

Step 2: Create a Simple Server

1. Create a `server.js` file

Create a new file named `server.js` in your project directory.

2. Set Up the Server

Use the built-in `http` module to create a simple server.

```
// server.js
const hostname = '127.0.0.1'; // Localhost
const port = 3000; // Port number
const server = http.createServer((req, res) => {
  res.statusCode = 200; // HTTP status code

  res.end('Hello, World!\n'); // Response message
});
server.listen(port, hostname, () => {
  console.log('Server running at http://${hostname}:${port}/');
});
```

```
});
```

Step 3: Run the Server

1. Start the Server

In your terminal, run the following command:

```
node server.js
```

You should see a message indicating that the server is running.

Step 4: Test the Server Connection

You can test the server connection using various methods:

1. Using a Web Browser

Open your web browser and navigate to `http://127.0.0.1:3000`. You should see the message "Hello, World!".

2. Using cURL

If you have cURL installed, you can test the connection from the command line:

```
curl http://127.0.0.1:3000
```

This should return "Hello, World!".

3. Using Postman

If you prefer a GUI tool, you can use Postman to send a GET request to `http://127.0.0.1:3000` and see the response.

Step 4: Handle Errors and Debugging

If you encounter issues:

- ✓ Check the terminal for any error messages.
- ✓ Ensure the server is running and listening on the correct port.
- ✓ Verify that you are using the correct URL when testing the connection.

Indicative content 1.3: Establishment of database connection

Below are the steps to establish a database connection in Node.js including:

- ✓ creating a database
- ✓ setting up a schema
- ✓ configuring the database connection
- ✓ and testing the connection.

We'll use MySQL as an example database, but the concepts can be adapted to other databases like PostgreSQL or MongoDB.

Step 1: Install Required Packages

First, you need to install the MySQL package for Node.js. You can use ``mysql2`` or ``mysql`` package. Here, we will use ``mysql2``.

```
npm install mysql2
```

Step 2: Create Database

You can create a database using a MySQL client or through a script. Here's how to do it in a script:

```
CREATE DATABASE my_database; //change the database name as you need.
```

You can run this command in a MySQL client or include it in your Node.js script.

Step 3: Schema Setup

You can set up a schema (i.e., create tables) using SQL commands.

For example:

```
CREATE TABLE users (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  name VARCHAR(100) NOT NULL,  
  email VARCHAR(100) UNIQUE NOT NULL  
);
```

For that case we have created a database called users with three fields id, name and email.

Step 4: Configure Database Connection

Create a file named ``app.js`` to handle the database connection.

Here's how you can configure it:

```
//import required module
```

```

const express=require('express');
const mysql=require('mysql');
const app=express();
const port=600;

//Create database connection
const db=mysql.createConnection({
  host:'localhost',
  user: 'your_username', // replace with your MySQL username
  password: 'your_password', // replace with your MySQL password
  database:'my_database', // replace with your database name
})

//define a basic route
app.get('/',(req,res)=>{
const sql='select * from table_name';// // Test the connection
db.query(sql,(err,data)=>{
  if(err)
    return res.json(err);
  return res.json(data);
})
})

//start express server
app.listen(port,()=>{
  console.log(`App is runnig on:http://localhost:${port}`)
})

```

Step 6: Run the Test Script

In your terminal, run the following command to execute the test script:

node app.js

If everything is set up correctly, you should see output indicating that you are connected to the database, and the solution to the query.

Indicative content 1.4: Develop RESTFUL APIs

Develop endpoints and HTTP Methods

Defining endpoints and HTTP methods using Express.js in Node.js. you follow different steps.

We'll create a simple RESTful API to manage a list of items.

Step 1: Set Up Your Express Server

First, ensure you have Express installed in your project.

Then, create a file named `server.js` and set up a basic Express server:

```
const express = require('express');
const app = express();
const PORT = 3000;

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Step 2: Define Endpoints

Now, let's define the endpoints for the various HTTP methods.

1. Create POST Endpoint

This endpoint will allow you to create a new item.

// Create POST endpoint

```
const express=require('express');
const mysql=require('mysql');
const app=express();
const port=7000;

const db= mysql.createConnection({
  host:'localhost',
  user:'root',
  password:'',
  database:'database-name',
})

app.post('/table-name',(req,res)=>{
```

```

const sql='insert into table-name (id,name,location) values(id,"peter1","musanze)';
db.query(sql,(err,data)=>{
  if(err)
    return res.json(err);
  return res.json('data is inserted');
})
})
app.listen(port,(req,res)=>{
  console.log(`on:http://localhost:${port}`);
})

```

2. Create All Items GET Endpoint

This endpoint retrieves all items.

// Create GET endpoint for all items

```

app.get('/items', (req, res) => {
  res.json(items); // Respond with the list of items
});

```

Or

```

const express=require('express');
const mysql=require('mysql');
const app=express();
const port=7000;

const db= mysql.createConnection({
  host:'localhost',
  user:'root',
  password:'',
  database:'db-name',
})

app.get('/table-name',(req,res)=>{
  const sql='select * from table-name';
  db.query(sql,(err,data)=>{
    if(err)
      return res.json(err);
    return res.json(data);
  })
})

```



```
app.listen(port, (req, res) => {
  console.log(`on: http://localhost:${port}`);
})
```

3. Create PUT Endpoint

This endpoint updates an existing item by its ID.

// Create PUT endpoint for updating an item

```
const express = require('express');
const mysql = require('mysql');
const app = express();
const port = 7000;

const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'db-name',
});

app.put('/table-name', (req, res) => {
  const sql = `update table-name set location="karongi" where id=3`;
  db.query(sql, (err, data) => {
    if (err)
      return res.json(err);
    return res.json('inserted');
  });
});

app.listen(port, (req, res) => {
  console.log(`on: http://localhost:${port}`);
})
```

5. Create DELETE Endpoint

This endpoint deletes an item by its ID.

// Create DELETE endpoint for removing an item

```
const express = require('express');
const mysql = require('mysql');
const app = express();
const port = 7000;

const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
});
```

```

    database: ' db-name',
  })

app.delete('/table-name', (req, res) => {
  const sql = 'delete from table-name where id=200';
  db.query(sql, (err, data) => {
    if (err)
      return res.json(err);
    return res.json('inserted');
  })
})

app.listen(port, (req, res) => {
  console.log(`on: http://localhost:${port}`);
})

```

Step 3: Testing the Endpoints

Testing the Endpoints using Postman

Testing your API endpoints using Postman is a straightforward process.

Here are the steps to follow to test the endpoints you created in your Express.js application:

Step 1: Check if Postman is installed

We have covered that topic in indicative content 1.

Step 2: Start Your Express Server

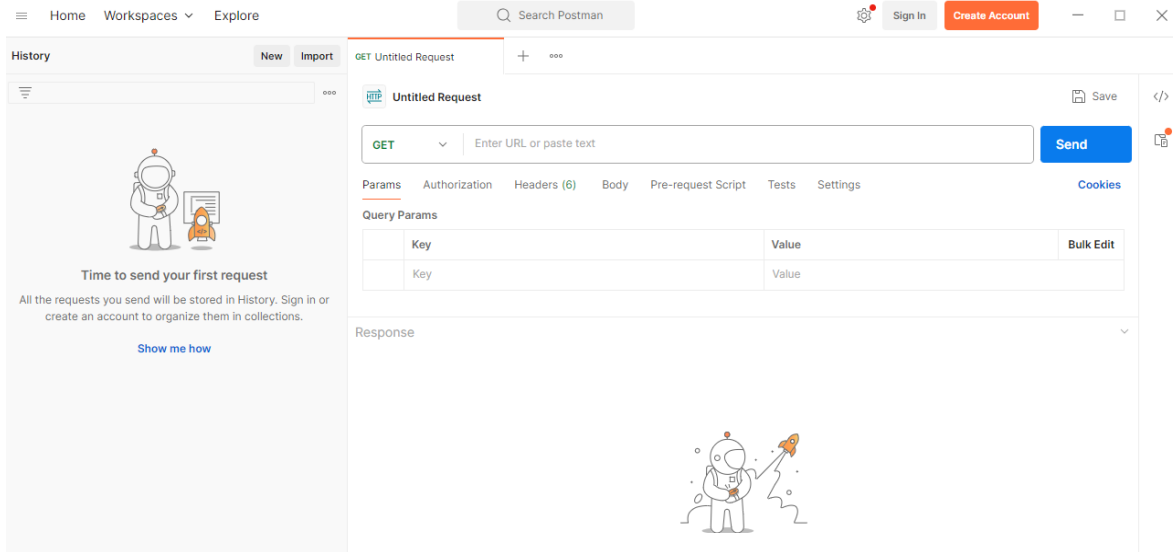
Before testing, ensure that your Express server is running. Open your terminal and navigate to your project directory, then run:

```
node server.js
```

You should see a message indicating that the server is running, e.g., `Server is running on http://localhost:3000`.

Step 3: Open Postman

Launch Postman after installation.



Step 4: Testing the Endpoints

1. Create a New Item (POST)

- ✓ Select POST from the dropdown menu next to the URL input field.
- ✓ Enter the URL: `http://localhost:3000/items`.
- ✓ Go to the Body tab and select raw. Then choose JSON from the dropdown menu.
- ✓ Enter the JSON data for the new item.

For example:

- ✓ Click the Send button. You should see a response with the created item, including its ID.

2. Get All Items (GET)

- ✓ Select GET from the dropdown menu.
- ✓ Enter the URL: `http://localhost:3000/items`.
- ✓ Click the Send button. You should see a response with an array of all items.

3. Update an Item (PUT)

- ✓ Select PUT from the dropdown menu.
- ✓ Enter the URL: `http://localhost:3000/items/1` (replace `1` with the ID of the item you want to update).
- ✓ Go to the Body tab, select raw, and choose JSON.

- ✓ Enter the new data for the item.

For example:

- ✓ Click the Send button. You should see a response with the updated item.

4. Delete an Item (DELETE)

- ✓ Select DELETE from the dropdown menu.
- ✓ Enter the URL: ``http://localhost:3000/items/1`` (replace ``1`` with the ID of the item you want to delete).
- ✓ Click the Send button. You should receive a response with a status code of ``204 No Content``, indicating that the item was successfully deleted.

Step 5: Check Responses

For each request, check the response section in Postman to see the data returned by your API.

You can test these endpoints using tools like Postman

- POST ``/items`` to create a new item.
- GET ``/items`` to retrieve all items.
- GET ``/items/:id`` to retrieve a specific item by ID.
- PUT ``/items/:id`` to update an existing item.
- DELETE ``/items/:id`` to delete an item.

1.4.4 Description of middleware services

1. Middleware services

Middleware services are software applications that act as a bridge between different applications, operating systems, and databases, allowing them to communicate and exchange data.

2. Use of Middleware Services

Middleware functions can be used for various purposes, such as:

- Logging requests: Keeping track of incoming requests.
- Parsing request bodies: Converting incoming request data into a usable format (e.g., JSON).
- Authentication: Verifying if a user is logged in or has the right permissions.
- Error handling: Catching errors and sending appropriate responses.
- Input validation: Ensuring that incoming data meets certain criteria before processing it

3. Types of Middleware Services

1. Application-Level Middleware
2. Router-level middleware
3. Error-Handling Middleware
4. Built-in Middleware
5. Third-Party Middleware (body-parser):
6. Logging middleware
7. Input Validation

1. **Application-Level Middleware:** Middleware that is bound to an instance of the application.

You can use it to apply middleware to specific routes or globally.

```
const express = require('express');
const app = express();
app.use(express.json()); // Built-in middleware to parse JSON bodies
```

Example:

```
// Application-level middleware for logging requests
// This middleware will execute for every incoming request to the app
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next(); // Pass control to the next middleware or route handler
});
```

2. **Router-Level Middleware:** Middleware that is bound to an instance of `express.Router()`.

It can be used to group routes and apply middleware to those routes.

```
const trainerRouter = express.Router();

// Middleware specific to the Router (e.g., logging requests to trainer-related routes)
trainerRouter.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, Request URL: ${req.originalUrl}`);
  next(); // Pass control to the next middleware or route handler
});
```

Example:

```
// Import required modules
const express = require('express');
const mysql = require('mysql');
```

```

// Initialize the app instance
const app = express();

// Set the port number
const port = 7000;

// Create a MySQL database connection
const db = mysql.createConnection({
  host: 'localhost',    // Database host
  user: 'root',         // Database user
  password: '',         // Database password
  database: 'rtti',     // Database name
});

// Create a Router instance for managing trainer-related routes
const trainerRouter = express.Router();

// Middleware specific to the Router (e.g., logging requests to trainer-related routes)
trainerRouter.use((req, res, next) => {
  console.log(`Request Method: ${req.method}, Request URL: ${req.originalUrl}`);
  next(); // Pass control to the next middleware or route handler
});

// Route to fetch all trainers
trainerRouter.get('/', (req, res) => {
  const sql = 'SELECT * FROM trainer'; // SQL query to fetch trainer data
  db.query(sql, (err, data) => {
    if (err) // Handle database errors
      return res.json(err);
    return res.json(data); // Send the data as JSON response
  });
});

// Route to insert a new trainer
trainerRouter.post('/', (req, res) => {
  const sql = 'INSERT INTO trainer (id, name, location) VALUES (900, "peter", "musanze)';
  db.query(sql, (err, data) => {
    if (err) // Handle database errors
      return res.json(err);
    return res.json('Trainer inserted'); // Send success message
  });
});

// Route to update a trainer's location

```

```

trainerRouter.put('/', (req, res) => {
  const sql = 'UPDATE trainer SET location="kirehe" WHERE name="peter"';
  db.query(sql, (err, data) => {
    if (err) // Handle database errors
      return res.json(err);
    return res.json('Trainer updated'); // Send success message
  });
});

// Route to delete a trainer by ID
trainerRouter.delete('/', (req, res) => {
  const sql = 'DELETE FROM trainer WHERE id=200';
  db.query(sql, (err, data) => {
    if (err) // Handle database errors
      return res.json(err);
    return res.json('Trainer deleted'); // Send success message
  });
});

// Attach the Router to the app at the '/trainer' path
app.use('/trainer', trainerRouter);

// Start the server and listen on the specified port
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});

```

3. Error-Handling Middleware:

Middleware specifically designed to catch and handle errors. It has four parameters: `err`, `req`, `res`, `next`.

```

// Error-handling middleware
// This middleware catches errors from any route or middleware above
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the error stack trace for debugging
  res.status(500).json({ // Send a 500 Internal Server Error response
    error: 'Something went wrong!', // Custom error message
    details: err.message // Include error details (optional, avoid in production for
security reasons)
  });
});

```


4. **Built-in Middleware:** Express comes with built-in middleware functions like `express.json()` and `express.urlencoded()`.

Example:

```
// Use built-in middleware to parse incoming JSON payloads
app.use(express.json()); // Parses JSON-encoded request bodies

// Use built-in middleware to parse URL-encoded data (useful for form submissions)
app.use(express.urlencoded({ extended: true })); // Parses URL-encoded data
```

5. Third-Party Middleware (body-parser): add functionality to Express apps

Third-Party Middleware :These middleware modules are typically installed via a package manager like **npm** and are used to add specific functionality or enhance existing capabilities in a web application.

Examples of Common Third-Party Middleware:

1. **body-parser:** Node.js library used to extract information from an incoming HTTP request and makes them available in `req.body`.

```
const bodyParser = require('body-parser');
app.use(bodyParser.json());
```

2. **cors:** Enables Cross-Origin Resource Sharing (CORS) for APIs.

```
const cors = require('cors');
app.use(cors());
```

3. **morgan:** Logs HTTP requests to the console.

```
const morgan = require('morgan');
app.use(morgan('dev'));
```

4. **helmet:** Helps secure Express apps by setting HTTP headers.

```
const helmet = require('helmet');
app.use(helmet());
```

Benefits of Using Third-Party Middleware:

1. **Time-Saving:** Reduces development time by leveraging existing solutions for common tasks.
2. **Reliable Solutions:** Many third-party middleware libraries are well-tested and widely used in the community.
3. **Focus on Core Logic:** Allows developers to concentrate on the application's unique features rather than reinventing the wheel.

EXAMPLES:

Step 1. **Install Packages:** Run the following command in your project directory to install `cors` and the other required modules:

```
npm install cors body-parser morgan helmet
```

Step 2.

```
// Importing required modules
const express=require('express');// Express framework for creating the server
const mysql=require('mysql');
const app=express();
const port=4000; // Initializing the Express application
const bodyParser = require('body-parser'); // Parses incoming request bodies
const cors = require('cors'); // Enables Cross-Origin Resource Sharing
const morgan = require('morgan'); // Logs HTTP requests to the console
const helmet = require('helmet'); // Secures Express apps by setting HTTP headers

const db= mysql.createConnection({
  host:'localhost',
  user:'root',
  password:'',
  database:'rtti',
})

// 1. Body-Parser Middleware
// Parses incoming JSON request bodies and makes them available in req.body
app.use(bodyParser.json());

// 2. CORS Middleware
// Enables Cross-Origin Resource Sharing to allow requests from different origins
app.use(cors());

// 3. Morgan Middleware
// Logs HTTP requests with a predefined format (in this case, 'dev' format)
app.use(morgan('dev'));

// 4. Helmet Middleware
// Adds security-related HTTP headers to the response
app.use(helmet());

// Route handler for fetching trainers
app.get('/trainer',(req,res)=>{
  const sql='select * from trainer';
  db.query(sql,(err,data)=>{
    if(err)
```

```

        return res.json(err);
        return res.json(data);
    })
})

app.post('/trainer',(req,res)=>{
    const sql='insert into trainer (id,name,location) values(900,"peter","musanze)';
    db.query(sql,(err,data)=>{
        if(err)
            return res.json(err);
        return res.json('inserted');
    })
})

app.put('/trainer',(req,res)=>{
    const sql='update trainer set location="kirehe" where name="peter" ';
    db.query(sql,(err,data)=>{
        if(err)
            return res.json(err);
        return res.json('updated');
    })
})

app.delete('/trainer',(req,res)=>{
    const sql='delete from trainer where id=200';
    db.query(sql,(err,data)=>{
        if(err)
            return res.json(err);
        return res.json('inserte');
    })
})

app.listen(port,(req,res)=>{
    console.log(`on:http://localhost:${port}`);
})

```

6. **Logging middleware** is essential for tracking requests and debugging issues in your application. It helps you monitor incoming requests, the status of responses, and any errors that occur.

7. Input Validation

Input validation middleware is crucial for ensuring that incoming data is in the expected format before it's processed by your application. This helps prevent errors and potential security vulnerabilities.

NB: These middleware services can be used together to create strong and secure applications.

Example Workflow:

- ✚ **Logging Middleware** records the details of the incoming request.
- ✚ **Input Validation Middleware** checks the data in the request and ensures it meets the necessary requirements.

If an error occurs (e.g., due to invalid input or server issues), **Error Handling Middleware** catches the error and sends an appropriate response to the client.

- **HTTP Status Codes**

100 Continue: The server has received the initial part of the request and is asking the client to continue with the rest(Representational State Transfer).

response sent by the server to inform the client that the initial part of the request has been received and is understood, and the client should continue with the rest of the request

101 Switching Protocols: The server is switching to a different protocol as requested by the client (e.g., upgrading from HTTP to WebSocket).

201 Created: Used when a new resource (developer) is successfully created.

200 OK: Used for successful requests that return data or indicate successful updates/deletions.

404 Not Found: Used when a requested resource (developer by ID) does not exist in the database.

500 Internal Server Error: Used when there is a server-side error, such as a database query failing.

How the Responses Work

Create Developer (POST /developers): Returns 201 Created if successful, along with the new developer's ID. If there's a server error, it returns 500 Internal Server Error.

Read All Developers (GET /developers): Returns 200 OK with the list of developers if successful. If there's a server error, it returns 500 Internal Server Error.

Read Developer by ID (GET /developers/:id): Returns 200 OK with the developer's details if found, 404 Not Found if the developer doesn't exist, and 500 Internal Server Error if there's a server error.

Update Developer (PUT /developers/:id): Returns 200 OK if successful, 404 Not Found if the developer doesn't exist, and 500 Internal Server Error if there's a server error.

Delete Developer (DELETE /developers/:id): Returns 200 OK if successful, 404 Not Found if the developer doesn't exist, and 500 Internal Server Error if there's a server error.

L.O.2 Secure Backend Application

IC.2.1.Data encryption in securing RESTFUL APIs.

Encryption is the process of encoding information.

Data encryption can be classified into several types based on different criteria. Here are some common types:

1. **Symmetric Encryption:** In symmetric encryption, the same key is used for both encryption and decryption. Examples include **AES** (Advanced Encryption Standard) and **DES** (Data Encryption Standard).
2. **Asymmetric Encryption (Public-Key Encryption):** Asymmetric encryption uses two keys, a public key for encryption and a private key for decryption. **RSA** (Rivest-Shamir-Adleman) and Elliptic Curve Cryptography (**ECC**) are examples of asymmetric encryption algorithms.
3. **Hashing:** Data is transformed into a fixed-size string of characters (hash value). It's used for data integrity verification rather than encryption. Common hash functions include SHA-256(Secure Hash Algorithm 256 bit) and MD5.

Example of Hashing:

The most commonly used method for hashing is known as modular hashing, which involves mapping a key k into one of the m slots by taking the remainder of k divided by m . This can be represented by the hash function $h(k) = k \bmod m$. For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$.

4. **Quantum Encryption:** Quantum encryption relies on the principles of quantum mechanics to secure data. Quantum key distribution (QKD) is a notable example that uses quantum properties to ensure secure communication channels.
5. **Homomorphic Encryption:** Homomorphic encryption allows computation on encrypted data without decrypting it first.

Homomorphic encryption is the conversion of data into ciphertext that can be analyzed and worked with as if it were still in its original form.

This enables operations such as addition and multiplication on ciphertext, producing results that are consistent with operations performed on plaintext.

6. ****End-to-End Encryption:** End-to-end encryption ensures that data is encrypted from the sender's device and only decrypted on the recipient's device, preventing intermediaries from accessing the plaintext. It's commonly used in messaging apps and secure communication protocols.

7. **Transport Layer Security (TLS) Encryption:** TLS encryption secures communication between clients and servers over a network. It ensures data confidentiality, integrity, and authentication during transmission, commonly used for securing web traffic (HTTPS).

Encryption techniques

In a RESTful API, **encryption techniques** are commonly used to secure data transmission between the client and server. ,

Encryption techniques used in REST APIs

1. **Transport Layer Security (TLS)/Secure Sockets Layer (SSL):** TLS/SSL is the most widely used encryption protocol for securing communication over a network. It ensures that data transmitted between the client and server is encrypted and authenticated, preventing eavesdropping and tampering.

2. **JSON Web Tokens (JWT):** JWT is a compact, URL-safe means of representing claims to be transferred between two parties. It's commonly used for authentication and contains encrypted data that can be verified and trusted.

3. **HTTPS:** HyperText Transfer Protocol Secure (HTTPS) combines HTTP with TLS/SSL encryption for secure communication over a computer network. It ensures that data transmitted between the client and server is encrypted, authenticated, and protected against man-in-the-middle attacks.

4. **Encryption Algorithms:** Various encryption algorithms such as AES (Advanced Encryption Standard) or RSA (Rivest-Shamir-Adleman) can be used to encrypt sensitive data before transmission. AES is commonly used for symmetric encryption, while RSA is used for asymmetric encryption.

5. **Hashing:** Hashing techniques like Secure Hash Algorithm 256-bit(SHA-256) can be used to generate unique fixed-size hash values from input data.

6. **OAuth:** OAuth is an authorization framework that allows third-party services to access a user's resources without sharing credentials.

OAuth allow users to authenticate using an existing account with a third-party service (such as Google or Facebook) rather than creating a new account.

Benefits and importance of data encryption

1. Confidentiality: This ensures that even if unauthorized parties gain access to the encrypted data, they cannot understand or misuse it.

2. Data Security: Encryption helps prevent unauthorized access to sensitive data(Data to be protected), whether it's stored on a device, transmitted over a network, or stored in the cloud.

It adds an extra layer of security to protect against data breaches (attacks for data) and unauthorized disclosure.

3. Protection Against Insider Threats: Encryption can also protect against insider threats (Risks from inside organization)by limiting access to sensitive data only to authorized users with the proper decryption keys.

5. Data Integrity: Encryption techniques often include mechanisms to ensure data integrity, meaning that the encrypted data cannot be altered without detection.

STEPS IN SECURING RESTFUL APIs

- **Install the crypto module**

You can not install the **crypto** module via npm, because it is a built-in module in Node.js, and it doesn't need to be installed separately using npm.

Js code


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

Once you've required the `crypto` module, you can use its various methods and classes to perform cryptographic operations in your application. Here's an example of how you can generate a cryptographic hash using the `crypto` module:

Js code

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

```
const data = 'Hello, world!';  
const hash = crypto.createHash('sha256').update(data).digest('hex');  
  
console.log('Hash:', hash);
```

- Create a key for encryption
- Use the key to encrypt data
- Convert the data to a buffer

What is buffer?

In Node.js, a **Buffer** is a built-in object that provides a way to work with binary data directly.

Example in JavaScript

```
// Create a Buffer from a string  
const buffer = Buffer.from('Hello, world!', 'utf-8');  
  
// Print the contents of the buffer  
console.log(buffer); // <Buffer 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21>  
  
// Convert the buffer back to a string  
const text = buffer.toString('utf-8');  
console.log(text); // Hello, world!
```

- Encrypt the data

- Store the encrypted data

NODE.JS THIRD-PARTY LIBRARIES

- Express: web application framework for Node.js
- Lodash: A utility library that provides functions for common programming tasks, such as manipulating arrays, objects, and strings.
- Moment.js

INTERACTING WITH THIRD-PARTY LIBRARIES

Interacting with third-party libraries in Node.js often involves asynchronous operations like making HTTP requests, reading/writing files, or querying databases.

Callbacks:

Callbacks are the traditional way of handling asynchronous operations in Node.js.

Promises:

Promises provide a more structured way to handle asynchronous code and mitigate the callback hell problem.

A Promise represents the eventual completion or failure of an asynchronous operation and allows chaining of asynchronous operations using `.then()` and `.catch()` methods.

async/await:

async/await is a syntactic sugar built on top of Promises, making asynchronous code look and behave more like synchronous code. The `async` keyword is used to define an asynchronous function, and the `await` keyword is used to pause the execution of the function until a Promise is resolved.

I.C 2.3 MAINTAINING AND UPDATING THIRD-PARTY LIBRARIES

Monitoring of library dependencies and version numbers

- Package. Json: This file lists all the dependencies, scripts for a Node.js project.

Example:

```
{  
  "dependencies": {  
    "express": "^4.18.0",
```

```
"lodash": "^4.17.21"
}
}
```

- Npm-shrinkwrap. Json: A file that locks down dependency versions.

Checking for library updates and security vulnerabilities using tools

- NPM outdated: Lists outdated packages

npm outdated

- NPM audit: Scans the dependencies for known vulnerabilities.

npm audit

- Snyk : An advanced tool for scanning vulnerabilities and suggesting fixes

snyk test

Updating third-party libraries safely

Third-party libraries in Node.js refer to external software packages That provide additional functionality and features that extend the capabilities of Node.js beyond its built-in modules.

Versioning
semver rules

1. Versioning:

Versioning is the practice of assigning unique identifiers to software releases to distinguish between different versions of the software.

2. Semantic Versioning (semver) Rules:

Semantic versioning (semver) is a versioning scheme that specifies how version numbers are assigned and incremented for software releases.

Semver defines version numbers using three components: MAJOR.MINOR.PATCH.

- **MAJOR**: Breaking changes (e.g., 1.0.0 → 2.0.0).
- **MINOR**: New features (e.g., 1.1.0 → 1.2.0).
- **PATCH**: Bug fixes (e.g., 1.1.0 → 1.1.1).

Strategies for managing and testing library updates

Staging Environments:

- Deploy updated dependencies to a staging server.
- Run integration and user acceptance tests.

Version Control Systems:

Tools like Git allow developers to track changes and updates if needed.

IC.2.4 IMPLEMENTATION OF AUTHENTICATION

Define Authentication:

Authentication is the process of verifying the identity of a user or system.

It ensures that the entity trying to access a particular resource or system is who or what it claims to be.

In digital contexts, authentication commonly involves presenting credentials, such as usernames and passwords, biometric data, cryptographic keys, or other forms of identification, to confirm one's identity.

Types of authentication:

1. Password-based authentication: Users provide a secret password or passphrase to verify their identity.

This method is widely used but can be vulnerable to password guessing or theft.

2. Biometric authentication: This involves verifying identity using unique biological traits such as fingerprints, iris patterns, facial features, or voice recognition.

Biometric methods offer strong security and are difficult to duplicate.

3. Token-based authentication: Users are issued physical or digital tokens, such as smart cards, or mobile authentication apps, which generate one-time passwords (OTPs) or cryptographic keys for authentication.

4. Multi-factor authentication (MFA): MFA combines two or more authentication factors, such as something the user knows (e.g., password), something the user has (e.g., smartphone), or something the user is (e.g., fingerprint). It provides an extra layer of security compared to single-factor authentication.

5. OAuth and OpenID Connect: These are authentication protocols commonly used for single sign-on (SSO). They allow users to authenticate using an existing account with a third-party service (such as Google or Facebook) rather than creating a new account.

6. Time-based authentication: Users are authenticated based on the time of access.

Principles of authentication

1. **Identification:** This is the first step where a user or entity provides a unique identifier, such as a username or email address.
2. **Authentication Factors:**
There are typically three types of authentication factors:
 - Knowledge factors: Something the user knows, like a password or PIN.
 - Possession factors: Something the user has, like a physical token, smart card, or mobile device.
 - Inherence factors: Something the user is, like biometric traits such as fingerprints, iris patterns, or facial recognition.
3. **Multi-factor Authentication (MFA):** Combining two or more authentication factors increases security.
4. **Secure Transmission:** Authentication data should be transmitted securely over networks to prevent interception or altering. This is often achieved through encryption protocols like TLS/ SSL.
5. **Session Management:** Once authenticated, systems must manage user sessions securely, including session expiration, logout mechanisms.
6. **Authorization:** Authentication is often followed by authorization, where the authenticated user's access rights and permissions are determined based on their identity and other attributes.
7. **User Education and Awareness:** Users need to understand the importance of strong authentication practices, such as creating strong passwords, safeguarding authentication tokens, and being vigilant against phishing attacks (steal user's data).

Role of authentication in system security

Authentication technology provides access control for systems by checking to see if a user's credentials match the credentials in a database of authorized users or a data authentication server.

ACCOUNTABILITY

Accountability is an assurance that an person or organization is evaluated on its performance or behaviour related to something for which it is responsible.

Roles of Accountability in system security

Every individual who works with an information system should have specific responsibilities for information assurance.

libraries for logging and auditing features in Node.js

- Winston
- Morgan

I.C 2.5 SECURE ENVIRONMENT VARIABLES

Environment variables are dynamic values that are set within the operating system environment and are accessible to all processes running on the system.

Types of information stored in environment variables

1. Database Credentials:

- Database credentials such as usernames, passwords, and connection strings are often stored in environment variables to secure access to databases.

2. API Keys:

- API keys are unique identifiers used to authenticate and authorize access to external APIs
- Storing API keys in environment variables helps protect them from unauthorized access and accidental exposure.

3. Encryption Keys:

- Encryption keys are used to encrypt and decrypt sensitive data to ensure its confidentiality and integrity.
- Storing encryption keys in environment variables helps secure sensitive data and prevent unauthorized access to encrypted information.

Storing environment variables in a secure location

- key management service
- a.env file: It is a simple text file that stores your environment variables in a key-value format.

In Node.js, using a .env file allows you to store variables like API keys, database credentials, or configuration options.

L.O 3. TEST BACKEND APPLICATION

I.C.3.1. Implementation of Unit testing

What is unit test:

Unit Testing is *the process of checking small pieces of code to deliver information.*

Importance of Unit testing

Unit testing plays a crucial role in software development for several reasons:

1. **Early Bug Detection:** Unit tests are typically written before the actual code is implemented. This practice helps catch bugs and defects early in the development process, reducing the cost and effort required to fix them later.
2. **Code Quality Assurance:** Unit tests ensure that each unit of code behaves as expected.
3. **Increased Confidence:** Having a comprehensive suite of unit tests gives developers and stakeholders confidence in the codebase, making it easier to deploy changes and updates without fear of breaking existing functionality.
4. **Faster Development Iterations:** With automated unit tests in place, developers can iterate more quickly. They can make changes to the code with confidence, knowing that unit tests will quickly identify any errors.

Unit Testing Process

The unit testing process typically involves the following steps:

1. **Identifying Units:** Break down the code into individual units or components that can be tested in isolation. These units can be functions, methods, classes, or modules.
2. **Writing Test Cases:** Develop test cases for each unit that cover different scenarios, including normal inputs, edge cases, and error conditions.
3. **Setting Up Test Environment:** Create a test environment that mimics(imitates) the production environment.
4. **Running Tests:** Execute the test cases using a unit testing framework or tool. The tests should automatically run and check whether the actual output matches the expected output.

5. Analyzing Results: Review the test results to identify any failures or errors. Investigate the root cause of the failures and make necessary corrections to the code.
6. Refactoring and Re-testing: If issues are found, refactor the code to fix the problems and re-run the tests to ensure that the changes have not introduced new bugs.
7. Continuous Integration: Integrate unit tests into the continuous integration (CI) process to run tests automatically whenever code changes are made.

INIT TESTING TOOLS

1. Jest
2. Chai
3. Sinon
4. AVA(Asynchronous Visual Assertions)

UNIT TESTING FRAMEWORKS

In Node.js, there are several popular unit testing frameworks that developers commonly use to test their JavaScript code. Some of the prominent ones include:

1. Mocha
2. Jasmine
3. Tape
4. Jest-circus
5. Sinon
6. Supertest
7. Webdriver

MOCHA TESTING FRAMEWORK

Installation of mocha in vscode

- npm install mocha for local installation in your project.
- npm install -g mocha for global installation

Chai installation

npm install chai

I.C.3.2. IMPLEMENTATION OF USABILITY TESTING

Topic 1: Introduction to Usability tests

Usability tests are a type of testing methodology used in software development to evaluate how user-friendly of system is and evaluate if it meets with the user's needs.

The Process of usability testing:

1. Define Objectives: Clearly define the goals and objectives of the usability testing. Determine what aspects of the product you want to evaluate like User satisfaction.
2. Recruit Participants: Identify and recruit participants who will help you to gather a range of perspectives and feedback.
3. Create Test Scenarios: Develop realistic and relevant test scenarios that reflect how users would interact with the product in real-life situations.
4. Conduct Testing: During the testing session, observe participants as they complete the assigned tasks. Encourage them to think aloud and provide feedback on their experiences, defeats, and successes.
5. Collect Data: Gather both qualitative and quantitative data during the testing session.
6. Analyze Results: Analyze the data collected during the usability testing to identify designs, styles, and areas for improvement.
7. Report Findings: Prepare a detailed report summarizing the findings of the usability testing, including key understandings(insights), recommendations for improvements.
8. Iterate and Improve: Use the insights gained from the usability testing to make iterative improvements to the product. Implement changes based on user feedback and retest the product to validate the improvements.

By following these steps and incorporating usability testing into the development process, you can create products that are more user-friendly, sensitive, and aligned with user needs and expectations.

Usability testing Tools

1. UserTesting
2. UsabilityHub
3. Lookback
4. Hotjar
5. Optimal Workshop
6. Google Analytics

Topic 2. Postman Testing Tool

1. Installation of Postman
2. Create a collection
3. Define Request
4. Write test Cases
5. Run tests
6. Iterate and improve

Topic 3. Puppeteer Testing Tool

Puppeteer is a Node.js library that provides a high-level API to control headless Chrome or Chromium over the DevTools Protocol. It's primarily used for automating tasks in web browsers, such as testing.

Installation of Puppeteer:

To install Puppeteer, you typically use npm (Node Package Manager), the package manager for Node.js.

```
npm install puppeteer
```

This will download and install the latest version of Puppeteer and its dependencies in your project.

Define Test Scenarios:

Once Puppeteer is installed, you can define test scenarios by writing JavaScript code that uses Puppeteer's API to control a headless browser. Test scenarios can include actions like navigating to a web page, interacting with elements on the page (e.g., clicking buttons, filling out forms), and verifying that certain elements or behaviors are present.

Automate User Interaction:

Puppeteer allows you to automate user interaction with web pages by simulating user actions like clicks, keyboard input, mouse movements, and more.

Test Accessibility:

Puppeteer can be used to test the accessibility of web pages by programmatically inspecting the DOM (Document Object Model) and verifying that it meets accessibility standards and guidelines.

Generate Report:

After running tests with Puppeteer, you can generate reports to summarize the results and provide insights into the status of your web application.

I.C.3.3. IMPLEMENTATION OF SECURITY TESTING

Topic 1. Introduction Node.js Security

1. Injection Attacks:

Injection attacks in Node.js, like in any other programming language, occur when untrusted data is sent to an interpreter as part of a command or query.

Types of injection attacks

1. SQL Injection: This occurs when untrusted user input is included in SQL queries sent to a database.
2. NoSQL Injection: While traditional SQL injection targets relational databases, NoSQL injection targets NoSQL databases like MongoDB. Attackers exploit vulnerabilities (Vulnerability is the quality of being easily hurt or attacked) in the query structure to manipulate data or retrieve sensitive information.
3. Command Injection: In Node.js applications, command injection occurs when untrusted input is passed to functions or methods that execute system

commands. Attackers can inject malicious commands to gain unauthorized access in order to disturb the system's operation.

4. XPath Injection: XPath injection targets applications that use XPath (XML Path Language) to query XML data. Attackers exploit vulnerabilities in XPath queries by injecting malicious input.

5. Header Injection: In Node.js applications, header injection occurs when untrusted input is used to construct HTTP headers. Attackers can inject malicious content into headers, potentially exploiting vulnerabilities such as HTTP response splitting or cross-site scripting (XSS).

BROKEN AUTHENTICATION

Broken authentication is a security vulnerability that occurs when an application's authentication mechanisms are not implemented correctly, allowing attackers to negotiate user accounts, gain unauthorized access to sensitive data.

A Security Vulnerability is a weakness, or error found within a security of the system.

Session management refers to the process of securely handling and maintaining user sessions within a web application.

A session is a period of interaction between a user and a web application, typically starting when the user logs in and ending when they log out or their session expires due to inactivity.

Session management involves several key characteristics:

1. Session Creation: When a user logs into a web application, a session is created to track their interactions. This usually involves generating a unique session identifier (session ID) and associating it with the user's authentication credentials.
2. Session Tracking: During the user's interaction with the application, the session ID is used to track their activities and maintain their state.

3. Session Security: It's crucial to ensure the security of sessions to prevent unauthorized access or tampering(altering). This involves securely generating session IDs, protecting them from being intercepted or stolen (e.g., by using HTTPS), and validating session data to prevent session hijacking(stealing session cookie) or tampering.

4. Session Expiry: Sessions should have a defined expiry time to limit their lifespan and reduce the risk of unauthorized access. Sessions can expire after a certain period of inactivity or be manually invalidated when the user logs out.

5. Session Termination: When a user logs out of the application, their session should be properly terminated to invalidate the session ID and clear any associated session data. This helps prevent unauthorized access if the user forgets to log out or if their session is hijacked(stolen).

CROSS-SITE SCRIPTING (XSS)

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application.

There are three main types of XSS attacks:

- [Reflected XSS](#), where the malicious script comes from the current HTTP request.
- [Stored XSS](#), where the malicious script comes from the website's database.
- [DOM-based XSS](#),(Document Object Model) where the vulnerability exists in client-side code rather than server-side code.

Security Misconfiguration

Security misconfiguration is a prevalent security issue that arises when a system is deployed with insecure default settings, incomplete configurations, or outdated software.

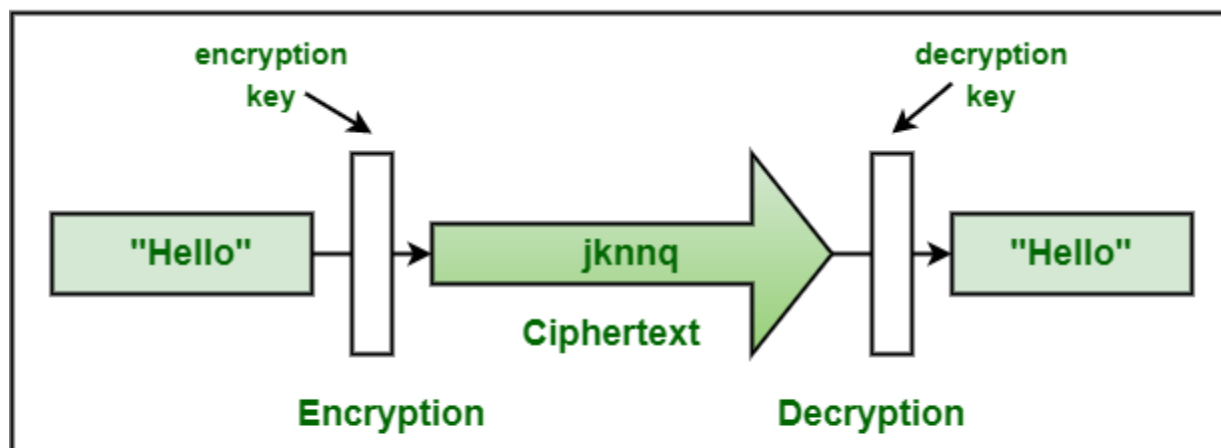
Causes of Security Misconfiguration:**

- 1. Default Configurations:** Default settings are typically not optimized for security and may expose sensitive information or allow unauthorized access.
- 2. Outdated Software:** Outdated software may contain known vulnerabilities that attackers can exploit to compromise the system.
- 3. Unnecessary Services and Features:** Running unnecessary services, ports, or features increases the attack surface and exposes the system to potential vulnerabilities.

INSECURE CRYPTOGRAPHIC STORAGE

Cryptography deals with secure communication techniques. It involves encoding information in such a way that only authorized parties can access and understand it, while keeping it hidden from unauthorized individuals.

Example :

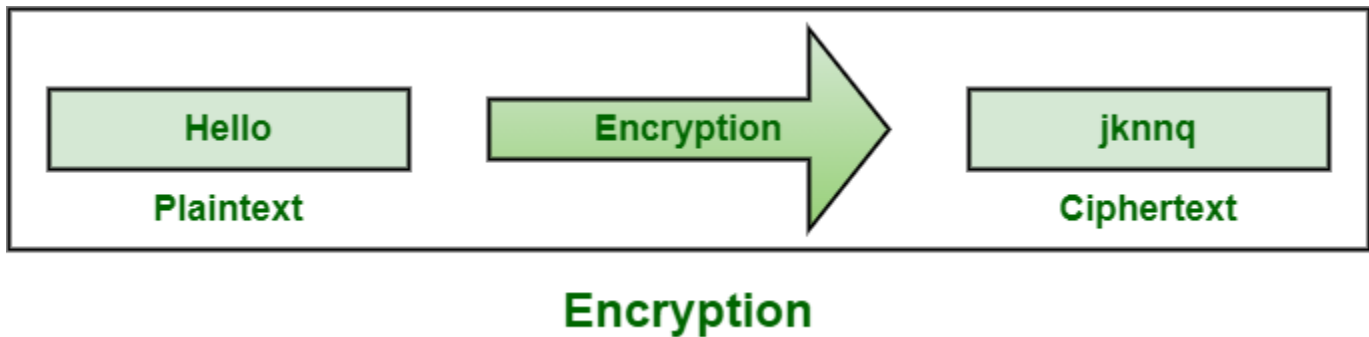


Cryptography

Encryption :

Encryption, as name suggests, is generally a technique that is used to conceal(hid) message using algorithms. It is fundamental application of cryptography that encodes a message with an algorithm.

Example :



Insecure Cryptographic Storage refers to a situation where sensitive information, such as passwords, encryption keys, or other confidential data, is stored using weak or inadequate cryptographic mechanisms or practices.

LOGGING

Logging is the process of recording events, actions, or transactions that occur within a system, application, or network.

MONITORING

Monitoring involves the continuous observation and analysis of system metrics, performance indicators, and log data to detect anomalies, identify trends, and ensure operational efficiency.

Topic 2. TOOLS FOR SECURITY TESTING IN NODE.JS

Security testing in Node.js involves assessing(evaluating) various aspects such as authentication, authorization, input validation, encryption, and more.

Tools For Security Testing In Node.Js are:

1. OWASP ZAP (Zed Attack Proxy):

- OWASP ZAP is an open-source web application security testing tool that can be used to test for vulnerabilities such as cross-site scripting (XSS), SQL injection.
- It can be used to perform automated scans as well as manual testing of web applications built on Node.js.

2. Node Security Project (NSP):

- NSP is a command-line tool and a database of known security vulnerabilities in **Node.js modules**.

- It helps developers identify and fix vulnerable dependencies in their Node.js applications by scanning their package.json file.

3. Snyk:

- Snyk is a developer-first security tool that helps identify, fix, and prevent security vulnerabilities in **Node.js dependencies**.

4. NPM Audit:

- NPM Audit is a built-in security feature in the Node Package Manager (NPM) that identifies security vulnerabilities in Node.js dependencies.

- It automatically runs during the installation of packages and provides a report of any vulnerabilities found.

5. Retire.js:

- Retire.js is a scanner tool that checks for outdated JavaScript libraries and frameworks, including those used in Node.js applications.

- It helps identify dependencies that have known security vulnerabilities and need to be updated.

6. Burp Suite:

- Burp Suite is a powerful web application security testing tool that can be used to test Node.js applications for various vulnerabilities.

- It includes features for intercepting and modifying HTTP requests, scanning for vulnerabilities, and more.

STATIC analysis tools

Static analysis tools in Node.js are software programs used to analyse source code written in JavaScript within Node.js applications without executing it.

static analysis tools in Node.js :

1. ESLint
2. JSHint
3. npm audit
4. Node Security Platform (NSP)
5. Flow
6. SonarQube
7. StandardJS.

Dynamic analysis tools

Dynamic analysis tools are software utilities designed to analyze the behavior of a running program or system during execution.

- 1. Profiling Tools:** Examples include Node.js's built-in profiler, Clinic.js, and New Relic.
- 2. Debuggers:** Examples include Node.js's built-in debugger, ndb, and Visual Studio Code's debugger.
- 3. Monitoring Tools:** Examples include Prometheus, Grafana, Datadog, and Splunk.
- 4. Security Testing Tools:** Examples include OWASP ZAP, Burp Suite, Snyk, and Qualys.
- 5. Fuzzers:** Examples include AFL (American Fuzzy Lop) and Peach Fuzzer.

Secure Coding Practices in Node.js

1. Libraries like ``validator`` or ``express-validator`` can assist in input validation.
2. Use Secure Dependencies: Consider using `package-lock.json` or `yarn.lock` to lock dependency versions.
- 3. Authentication and Authorization:** Implement strong authentication mechanisms such as JWT (JSON Web Tokens) or OAuth for user authentication.
4. Data Encryption: Use libraries like ``crypto`` to encrypt and decrypt data securely.

Security Testing Lifecycle

The Security Testing Lifecycle refers to the process of identifying, assessing, and mitigating security risks in software applications or systems throughout their development and operational phases.

REMEDIATION AND MITIGATION

1. Remediation: Remediation refers to the process of fixing or resolving identified security vulnerabilities or weaknesses in a system.
2. Mitigation: involves reducing the impact of a security vulnerability or risk, even if the root cause cannot be completely eliminated.

COMPLIANCE AND REGULATIONS

Compliance and regulations are terms used in the context of ensuring that organizations obey specific laws, standards, guidelines, or requirements that are relevant to their industry or operations.

1. Compliance: It involves the ensuring that an organization follows all applicable laws and regulations, as well as internal policies and procedures, to maintain legal and ethical standards in its operations.
2. ****Regulations****: Regulations are official rules established by government authorities to govern specific industries, activities, or behaviors.

L.0.4 MANAGE BACKEND APPLICATION

I.C.1 Preparation of deployment Environment

Topic1. Description of NodeJS application deployment

Deployment

Deployment refers to the process of releasing and making a software application available to be accessed and used by end-users in a specific environment.

Types of NodeJS application deployment

1. Manual Deployment:

- Manual deployment involves manually transferring the Node.js application files and dependencies to the target server or hosting environment.

2. Continuous Deployment:

- Continuous deployment automates the deployment process, allowing developers to release updates to the Node.js application frequently and predictably.

3. Docker-based Deployment:

- Docker-based deployment involves packaging the Node.js application and its dependencies into lightweight, portable containers using Docker.

I.C 2. Implementation of Manual Deployment of NodeJS application

Steps of Manual Deployment of NodeJS application

- ✓ Copy the application source code to the server
- ✓ Installation of dependencies
- ✓ Start the application using command line

I.C.3 Maintenance of NodeJS application

- ✓ **Developing a maintenance plan**

Steps of maintenance plan

1. Identification of Maintenance Requirements:

This step involves identifying the specific maintenance tasks and requirements for the Node.js application.

2. Schedule Regular Updates:

Regular updates are essential to keep the Node.js application up-to-date with the latest features, bug fixes, and security areas.

3. Automate Maintenance Tasks:

Automating maintenance tasks can help update the maintenance process, reduce human error, and save time and effort.

4. Monitor Application Performance:

Use monitoring tools to collect and analyze the response times, CPU usage, memory usage, and error rates. Set up alerts to notify you of any anomalies or performance degradation, allowing you to take timely action to resolve issues and optimize performance.

5. Test Regularly:

Regular testing is essential to ensure the consistency, functionality, and quality of the Node.js Application.

6. Disaster Recovery Plan:

Identify potential risks and vulnerabilities in the application infrastructure and data storage. Implement backup and recovery procedures, data replication, failover mechanisms, and disaster recovery drills to minimize downtime (time when one is not working) and data loss in the event of a disaster.

7. Document Changes:

Keep detailed records of changes to the codebase, configuration settings, dependencies, and infrastructure.

✓ Continuous maintenance and improvement of NodeJS applications

- Upgrade and maintain previously developed functionalities,
- develop new functionalities,
- Secure new and previously developed functionalities,
- Test new functionalities,
- Deploy new changes

I.C. 4 APPLICATION OF NODEJS DOCUMENTATION TOOLS AND FRAMEWORKS

1. Documentation Overview:

Documentation refers to the process of creating and maintaining written information about a software application, including its functionality, architecture, APIs, and usage.

2. The Importance of Documentation:

Documentation is essential for understanding and using a software application effectively. It serves as a reference guide for developers, helping them understand the codebase, APIs, and implementation details.

Types of Documentation:

There are several types of documentation that may be created for a Node.js application, including:

1. Code Documentation
2. API Documentation
3. User Documentation
4. Technical Documentation

Popular Documentation Tools and Frameworks:

1. Swagger/OpenAPI: A framework for defining, documenting, and testing APIs using a standard specification format.

2. Postman: An API development platform that includes tools for creating, testing, and documenting APIs.

3. JSDoc: A documentation generator for JavaScript that extracts comments from source code and generates HTML documentation.

Publishing Documentation:

Publishing documentation involves making it accessible to users, stakeholders, and contributors.

1. Hosting documentation : This involves selecting a platform or service where the documentation will be hosted and made available to users.

2. GitHub for Collaborative Documentation

GitHub is a popular platform for hosting code repositories, but it's also widely used for collaborative documentation projects.

Teams can use GitHub to store documentation files, track changes made by team members, suggest edits, review changes, and manage the workflow of creating and updating documentation.

3. Documentation Maintenance:

Documentation maintenance is an ongoing process that involves regularly reviewing, updating, and improving documentation to ensure its accuracy, relevance, and usefulness.