RTB | RWANDA TVET BOARD

RQF LEVEL 4

SWDBD401

SOFTWARE DEVELOPMENT

Backend Application Development

TRAINEE'S MANUAL

October, 2024

RTB | RWANDA TVET BOARD

BACKEND APPLICATION DEVELOPMENT

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

© Rwanda TVET Board

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate IV in Software Development specifically for the module "**SWDBD401 Backend Application Development".**

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMA Usher Emmanuel

PRODUCTION TEAM

Authoring and Review

SEKABANZA Jean de la Paix

BUGIRUWENDA Jean Bosco

Validation

MUNYEMANA Alexis

ISHIMWE Gilbert

Conception, Adaptation, and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

NTEZIMANA Aimable

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

MINANI Aloys

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

AAA: Authentication, Authorization, and Accountability

ABAC: Attribute-Based Access Control

AES: Advanced Encryption Standard

APIs: Application Programming Interfaces

AWS: Amazon Web Services

CI/CD: Continuous Integration/Continuous Deployment

GDPR: General Data Protection Regulation

HIPAA: Health Insurance Portability and Accountability Act

HTTPS: Hypertext Transfer Protocol Secure

JWT: JSON Web Token

NPM: Node Package Manager

OWASP: Open Worldwide Application Security Project

RBAC: Role-based Access Control

RSA: Rivest-Shamir-Adleman

RTB: Rwanda TVET Board

SSL: Secure Socket Layer

SSO: Single Sign-On

TLS: Transport Layer Security

TQUM Project: TVET Quality Management Project

TVET: Technical Vocational and Education Training

UAC: Universal Access Control

INTRODUCTION

This trainer's manual includes all the methodologies required to effectively deliver the module titled **“Backend Application Development“** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies.

This training manual was developed using the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainer's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainer, you will begin by asking questions related to the activities to encourage critical thinking and guide trainees toward real-world applications in the labour market. The manual also outlines essential information such as learning hours, didactic materials, and suggested methodologies.

This manual outlines the procedures and methodologies for guiding trainees through various activities as detailed in their respective trainee manuals. The activities included in this training manual are designed to offer students opportunities for both individual and group work. Upon completing all activities, you will assist trainees in conducting a formative assessment known as the end learning outcome assessment. Ensure that students review the key reading and the points to remember section.

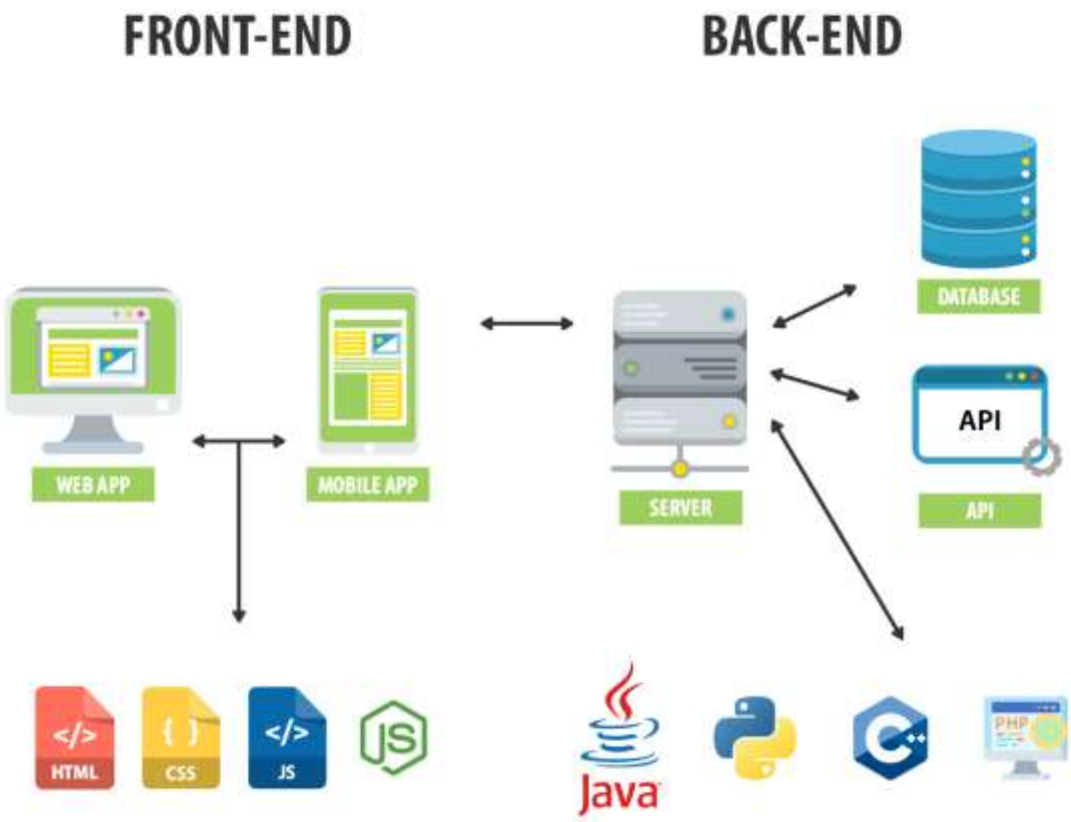
MODULE CODE AND TITLE: SWDBD401 BACKEND APPLICATION DEVELOPMENT

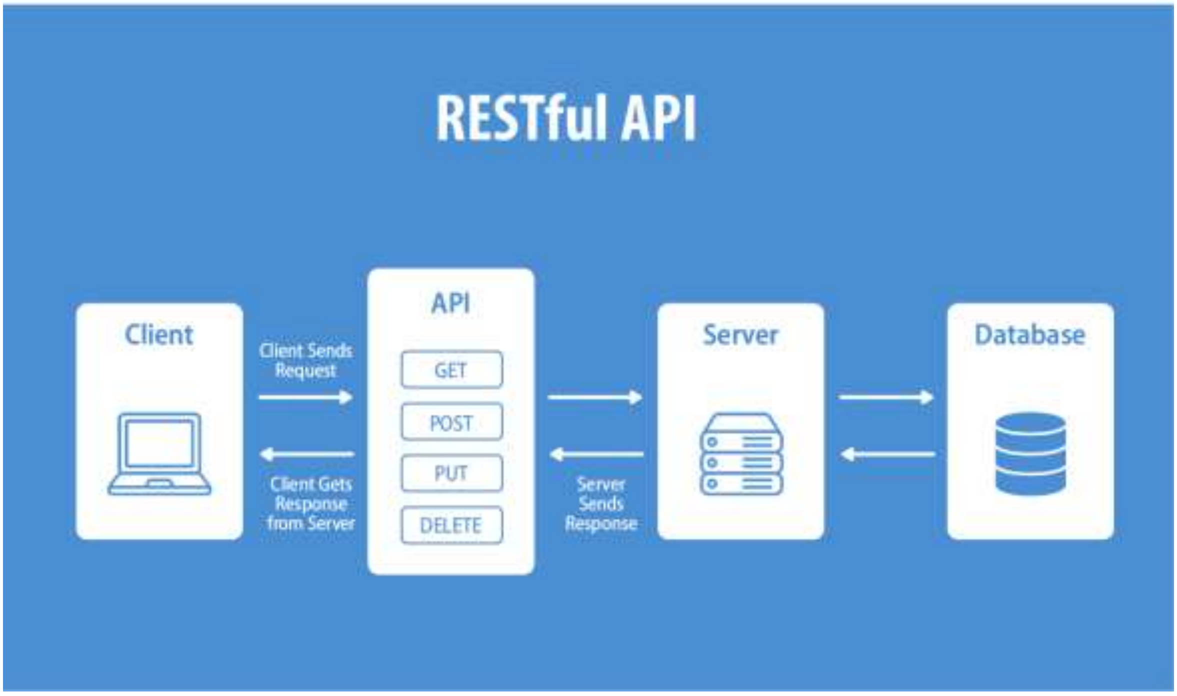
Learning Outcome 1: Develop RESTFUL APIs with Node JS

Learning Outcome 2: Secure Backend Application

Learning Outcome 3: Test the Backend Application

Learning Outcome 4: Manage Backend Application


FRONT-END
BACK-END
WEB APP
MOBILE APP
SERVER
DATABASE
API
API
HTML
CSS
JS
JS
Java
PHP


RESTful API
Client
Client Sends Request
Client Gets Response from Server
API
GET
POST
PUT
DELETE
Server Sends Response
Server
Database

Indicative contents

1.1 Setup Node. Js Environment

1.2 Connection of Node Js to the ES5 or ES6 server

1.3 Establishment of database connection

1.4 Develop RESTFUL APIs

Key Competencies for Learning Outcome 1: Develop RESTFUL APIs with Node JS

Knowledge	Skills	Attitudes
● Description of Node.js Key Concepts ● Description of APIs ● Description of database management system	● Installing Node Js Modules and packages ● Connecting node Js to the ES5 or ES6 server ● Configuring MySQL server ● Debugging endpoints/APIs ● Installing postman ● Establishing database connection ● Developing RESTFUL APIs ● Using Middleware services ● Implementing CRUD	● Having Teamwork ability ● Being a critical thinker ● Being Innovative ● Being creative ● Being honesty ● Having a Passion for Learning ● Having Problem-Solving Mindset ● Having good Collaboration and Communication ● Being Attentive to Security ● Having Ethical Coding

	operations using MySQL Database ● Using HTTP Status code	

Duration: 45 hrs

Learning Outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly Node.js Key Concepts as applied in software development
2. Describe correctly database management system as they are used in backend development.
3. Install and configure properly Postman as it is used in Backend development.
4. Installing and configuring MySQL Server properly as is used in backend development.
5. Describe correctly the term API as used in software development.
6. Install correctly Node Js Modules and packages as used in backend application development
7. Connect properly node Js to the ES5 or ES6 server as applied in the server configuration
8. Develop correctly RESTFUL APIs as applied in the software development process.
9. Establish correctly database connection as applied in backend system development.
10. Use correctly Middleware services as applied in software development.
11. Perform properly CRUD operations using MySQL Database as used in the database connection.
12. Use correctly HTTP Status code correctly as required in error handling in software development

Resources

Equipment	**Tools**	**Materials**
● Computer	● Browser ● NodeJS	● Internet ● Electricity

	● Postman ● MySQL server	

Indicative content 1.1: Setup Node. Js Environment

Duration: 10 hrs

Theoretical Activity 1.1.1: Description of Node.js Key Concepts

Tasks:

1: You are requested to answer the following questions related to the description of Node.js Key Concepts:

i. Define the following terms:

   a) Node js
   b) Npm
   c) Dependencies
   d) Backend application
   e) Routes
   f) express js
   g) Nodemon

ii. Differentiate the following terms used in function:

   a) Class
   b) Object
   c) Method
   d) Properties

iii. Describe Postman as a tool used in backend system development.

iv. Describe APIs as used in backend system development.

v. Describe the database management system.

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 1.1.1. In addition, ask questions where necessary.

Key readings 1.1.1. Description of Node.js Key Concepts

1. **Node.js:**

   Node.js is an open-source JavaScript runtime environment that allows developers to execute JavaScript code on the server side. It provides an event-driven, non-blocking I/O model, making it highly efficient for building scalable and real-time applications.

2. **NPM (Node Package Manager):**

   NPM is the default package manager for Node.js. 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.

3. **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 are used to install and manage them.

4. **Backend Application:**

   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.

5. **Routes:**

   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.

6. **Class:**

   In JavaScript, a class is a blueprint for creating objects with shared properties and methods. It is a fundamental concept in object-oriented programming (OOP) and allows for the creation of structured and reusable code.

7. **Object:**

   An object in JavaScript is a composite data type that stores key-value pairs. Objects can represent real-world entities and encapsulate both data (properties) and behavior (methods) related to those entities.

8. **Method:**

   In JavaScript, a method is a function that is associated with an object and can be called to perform actions or manipulate data related to that object.

9. **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.

10. **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.

11. 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.

12. 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.

13. **API**

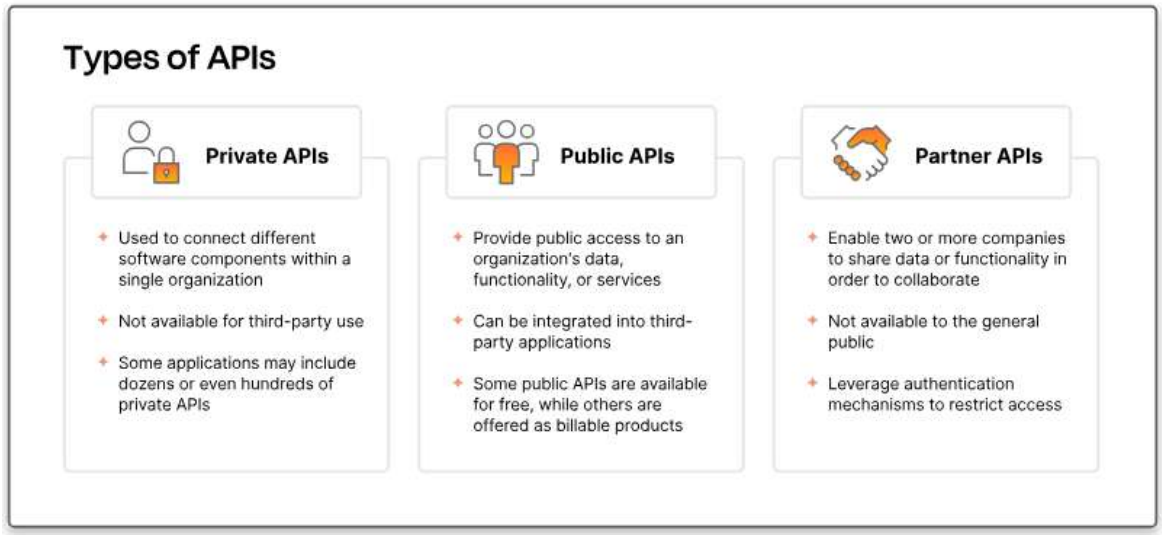
API stands for Application Programming Interface.

It defines a set of rules and protocols that allow one software application to interact with another.

APIs specify the methods and data formats that applications can use to request and exchange information, enabling seamless integration and communication between different software systems.

APIs are commonly used in web development to connect web applications with external services and resources.

Types of API



Types of APIs

Private APIs

- Used to connect different software components within a single organization
- Not available for third-party use
- Some applications may include dozens or even hundreds of private APIs

Public APIs

- Provide public access to an organization's data, functionality, or services
- Can be integrated into third-party applications
- Some public APIs are available for free, while others are offered as billable products

Partner APIs

- Enable two or more companies to share data or functionality in order to collaborate
- Not available to the general public
- Leverage authentication mechanisms to restrict access

Advantages of APIs

- **Automation:** APIs can be used to automate repetitive, time-consuming work so that humans can focus on more complex tasks. This improves productivity, especially for developers and testers.

- **Innovation:** Public APIs can be used by external engineering teams, which spurs innovation and accelerates development by enabling developers to repurpose existing **functionality** to create new digital experiences.
- **Security:** APIs can provide an additional layer of protection against unauthorized breaches by requiring authentication and authorization for any request to access sensitive data.
- **Cost efficiency:** APIs provide access to useful third-party tools and infrastructure, which helps businesses avoid the expense of building complex in-house systems.

Common API use cases

1. Integrating with internal and external systems

One of the most common reasons developers turn to APIs is to integrate one system with another. For instance, you can use an API to integrate your customer relationship management (CRM) system with your marketing automation system, which would allow you to automatically send a marketing email when a sales representative adds a new prospective customer to the CRM.

2. Adding or enhancing functionality

APIs let you incorporate additional functionality into your application, which can improve your customers' experience. For instance, if you're working on a food delivery application, you might incorporate a third-party mapping API to let users track their order while it's en route.

3. Connecting IoT devices

APIs are essential to the Internet of Things (IoT) ecosystem, which includes devices such as smart watches, fitness trackers, doorbells, and home appliances. Without APIs, these devices would not be able to connect to the cloud—or to one another—which would render them useless.

4. Creating more scalable systems

APIs are used to implement microservice-based architectures, in which applications are built as a collection of small services that communicate with one another through private APIs. Microservices are managed, deployed, and provisioned independently of one another, which enables teams to scale their systems in a reliable yet cost-efficient way.

5. Reducing costs

APIs help organizations reduce operational costs by automating time-intensive tasks, such as sending emails, pulling reports, and sharing data between systems. They can also reduce development costs by enabling teams to reuse existing functionality, instead of reinventing the wheel.

6. Improving organizational security and governance

APIs power many workflows that are essential for organizational security. For instance, single sign-on (SSO), which enables users to use one username and password for multiple systems, is made possible by APIs. APIs are also used to enforce and automate corporate governance rules and policies, such as a requirement that expenses be approved before employees are reimbursed

API working process

APIs work by sharing data between applications, systems, and devices. This happens through a request and response cycle. The request is sent to the API, which retrieves the data and returns it to the user. Here's a high-level overview of how that process works.

1. API client

The API client is responsible for starting the conversation by sending the request to the API server. The request can be triggered in many ways. For instance, a user might initiate an API request by entering a search term or clicking a button. API requests may also be triggered by external events, such as a notification from another application.

2. API request

An API request will look and behave differently depending on the type of API, but it will typically include the following components:

- **Endpoint:** An API endpoint is a dedicated URL that provides access to a specific resource. For instance, the /articles endpoint in a blogging app would include the logic for processing all requests that are related to articles.

- **Method:** The request's method indicates the type of operation the client would like to perform on a given resource. REST APIs are accessible through standard HTTP methods, which perform common actions like retrieving, creating, updating, and deleting data.

- **Parameters:** Parameters are the variables that are passed to an API endpoint to provide specific instructions for the API to process. These parameters can be included in the API request as part of the URL, in the query string, or in the request body. For example, the /articles endpoint of a blogging API might accept a “topic” parameter, which it would use to access and return articles on a specific topic.

- **Request headers:** Request headers are key-value pairs that provide extra details about the request, such as its content type or authentication credentials.

- **Request body:** The body is the main part of the request, and it includes the actual data that is required to create, update, or delete a resource. For instance, if you were creating a new article in a blogging app, the request body would likely include the article's content, title, and author.

3. API server

The API client sends the request to the API server, which is responsible for handling authentication, validating input data, and retrieving or manipulating data.

4. API response

Finally, the API server sends a response to the client. The API response typically includes the following components:

- **Status code:** HTTP status codes are three-digit codes that indicate the outcome of an API request.

Some of the most common status codes include:

1. Informational (1xx)

- **100 Continue:** The server has received the request headers, and the client should proceed to send the request body (if any).
- **101 Switching Protocols:** The server is switching protocols as requested by the client (e.g., switching from HTTP to WebSocket).
- **102 Processing: The server has received and is processing the request, but no response is available.**

2. Successful (2xx)

These indicate that the request was successfully received, understood, and accepted by the server.

- **200 OK:** The request was successful, and the server returned the requested data.
- **201 Created:** The request was successful, and the server created a new resource (often used in POST requests).
- **202 Accepted:** The request has been accepted for processing, but it has not yet been completed.
- **204 No Content:** The request was successful, but there is no content to return (e.g., a successful DELETE request).
- **205 Reset Content:** The request was successful, but the client should reset the document view.
- **206 Partial Content:** The server returns a part of the resource (usually for range requests).

3. Redirection (3xx)

These codes indicate that further action is needed to fulfill the request, usually a URL redirect.

- **301 Moved Permanently:** The resource has been permanently moved to a new

URL, and future requests should use the new URL.

- **302 Found (Temporary Redirect):** The resource has been temporarily moved to a different URL.
- **303 See Other:** The response to the request can be found at another URL, often used after a POST request.
- **304 Not Modified:** The resource has not been modified since the last request, and the client can use the cached version.
- **307 Temporary Redirect:** The resource is temporarily available at another URL, and the client should use the same HTTP method.
- **308 Permanent Redirect:** The resource has been permanently moved to a new URL, and the client should use the new URL with the same HTTP method.

4. Client Error (4xx)

These codes indicate that the client seems to have made an error in the request.

- **400 Bad Request:** The request could not be understood or was malformed.
- **401 Unauthorized:** The request requires authentication, but the client has not provided valid credentials.
- **402 Payment Required:** Reserved for future use (though it’s rarely used).
- **403 Forbidden:** The server understood the request, but it refuses to authorize it.
- **404 Not Found:** The requested resource could not be found on the server.
- **405 Method Not Allowed:** The HTTP method used is not allowed for the resource.
- **406 Not Acceptable:** The server cannot produce a response that matches the client's requested content type.
- **407 Proxy Authentication Required:** The client must authenticate with a proxy before proceeding.
- **408 Request Timeout:** The server timed out waiting for the client to send the request.
- **409 Conflict:** The request could not be completed due to a conflict with the current state of the resource (e.g., attempting to create a resource that already exists).
- **410 Gone:** The resource is no longer available and will not be available again.
- **411 Length Required:** The server requires the Content-Length header to be set.
- **412 Precondition Failed:** One or more conditions in the request header fields were not met.
- **413 Payload Too Large:** The request entity is too large to process.
- **414 URI Too Long:** The request URI is too long for the server to process.
- **415 Unsupported Media Type:** The server does not support the media type of the request.
- **416 Range Not Satisfiable:** The range specified by the client cannot be fulfilled.

- **417 Expectation Failed:** The server cannot meet the expectations set by the Expect header.
- **418 I'm a teapot:** An April Fools' joke from the HTTP/1.1 specification. It's a humorous error.
- **422 Unprocessable Entity:** The server understands the request but cannot process it (often used for validation errors).
- **429 Too Many Requests:** The client has sent too many requests in a given amount of time (rate limiting).

5. Server Error (5xx)

These codes indicate that the server failed to fulfill a valid request due to an error on its part.

- **500 Internal Server Error:** A generic error when the server encounters an unexpected condition.
- **501 Not Implemented:** The server does not support the functionality required to fulfill the request.
- **502 Bad Gateway:** The server, while acting as a gateway or proxy, received an invalid response from the upstream server.
- **503 Service Unavailable:** The server is currently unable to handle the request, usually due to being overloaded or down for maintenance.
- **504 Gateway Timeout:** The server, acting as a gateway, did not receive a timely response from the upstream server.
- **505 HTTP Version Not Supported:** The server does not support the HTTP protocol version used in the request.
- **507 Insufficient Storage:** The server is unable to store the representation needed to complete the request.
- **508 Loop Detected:** The server detected an infinite loop while processing a request.
- **510 Not Extended:** The server requires further extensions to fulfill the request.
- **511 Network Authentication Required:** The client needs to authenticate to gain network access.

These status codes are essential for understanding how a server processes requests and helps developers troubleshoot and optimize their applications.

- **Response headers**: HTTP response headers are very similar to request headers, except they are used to provide additional information about the server's response.
- **Response body**: The response body includes the actual data or content the client asked for—or an error message if something went wrong.

In order to better understand this process, it can be useful to think of APIs like restaurants. In this metaphor, the customer is like the user, who tells the waiter what she wants. The waiter is like an API client, receiving the customer's order and translating it into easy-to-follow instructions for the kitchen—sometimes using specific codes or abbreviations that the kitchen staff will recognize. The kitchen staff is like the API server because it creates the order according to the customer's specifications and gives it to the waiter, who then delivers it to the customer.

DBMS (Database Management System):

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)

✓ 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.

SQL-Based Databases

Characteristics:

Schema: SQL databases use a predefined schema with tables, rows, and columns.

ACID Compliance: They are designed to ensure Atomicity, Consistency, Isolation, and Durability (ACID properties) which guarantee reliable transactions.

Structure: Data is stored in a structured format using tables.

Query Language: Uses Structured Query Language (SQL) for defining and manipulating data.

Normalization: Data is often normalized to reduce redundancy.

Examples:

MySQL, PostgreSQL, Microsoft SQL server, Oracle and Microsoft Office access.

NoSQL-Based Databases

Characteristics:

Schema-less: NoSQL databases are typically schema-less, allowing for flexible data models.

BASE: They follow the BASE properties (Basically Available, Soft state, Eventual consistency), which is more suitable for distributed systems.

Variety of Data Models: Includes document-based, key-value pairs, wide-column stores, and graph databases.

Scalability: Designed for horizontal scaling, which makes them suitable for handling large volumes of unstructured data.

Flexibility: Can handle a variety of data types, including structured, semi-structured, and unstructured data.

Examples:

MongoDB, Redis, Cassandra

Practical Activity 1.1.2: Installation of Node Js Modules and packages

Task:

1. Read the task below

You are requested to go to the computer lab to install node js, make a directory, navigate to the created directory, initialize npm, install express, and Nodemon, and open the created project in VS Code.

2. Refers to provided key reading 1.1.2, perform the task described above.

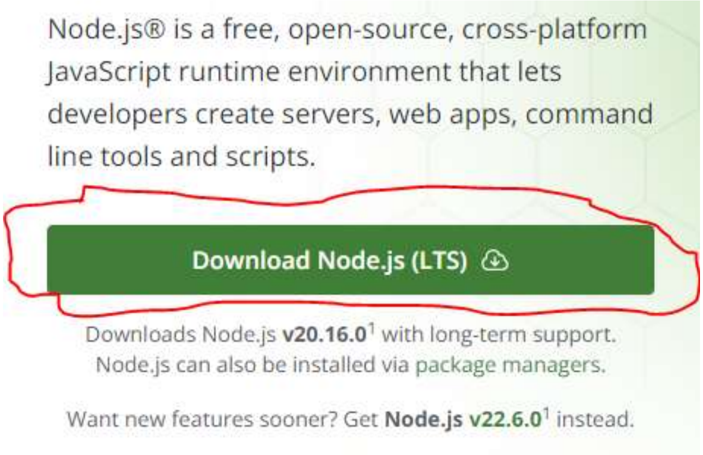
3. Present your work to the trainer and whole class.

Key readings 1.1.2: Installation of Node Js Modules and packages

1. **Install node js**

✓ Visit the Node.js official website (https://nodejs.org/en) or install it from external driver such as flash disk.

✓ Download the installer for your operating system (Windows, macOS, Linux).



Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

Download Node.js (LTS)

Downloads Node.js **v20.16.0**[1] with long-term support.
Node.js can also be installed via package managers.

Want new features sooner? Get **Node.js v22.6.0**[1] instead.

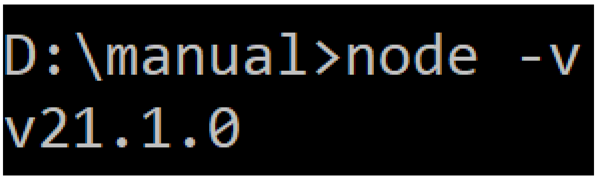
✓ Run the installer and follow the prompts to install Node.js.

2. **Check if node and npm are installed**

✓ Open a terminal or command prompt.

✓ To check Node.js installation, run:

```
node –v
```

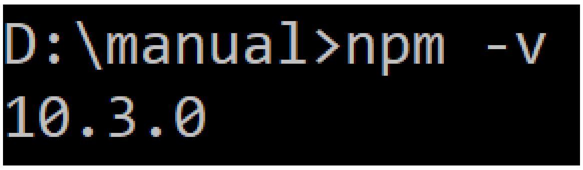


```
D:\manual>node -v
v21.1.0
```

This command should output the version of Node.js installed.

✓ To check npm installation, run:

npm –v



```
D:\manual>npm -v
10.3.0
```

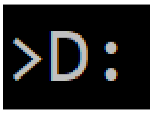
This command should output the version of npm installed.

3. **Make a directory**

✓ Open your terminal or command prompt.

✓ Navigate to the directory where you want to create your project, for example:

D: then press Enter will move you to local disk D of your computer

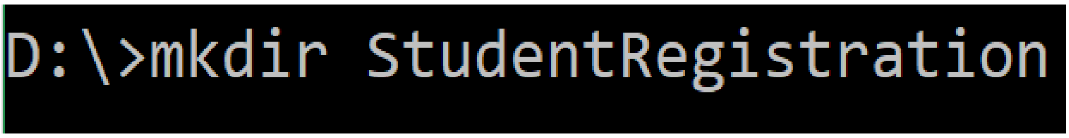


```
>D:
```

✓ Create a new directory (replace project-name with your desired project name):

mkdir project-name

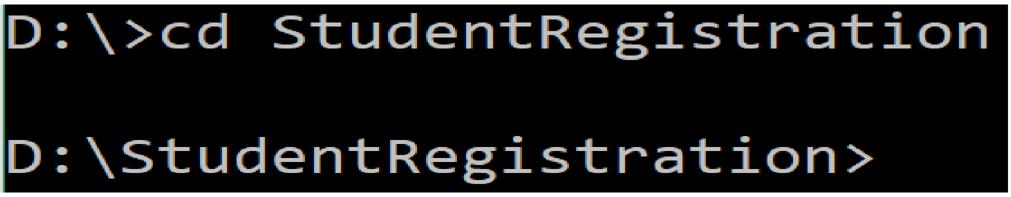
Example: Let us create a project called StudentRegistartion



```
D:\>mkdir StudentRegistration
```

4. **Navigate to created directory**

After creating the directory, navigate into it:

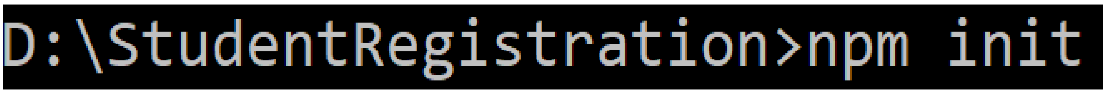


```
D:\>cd StudentRegistration

D:\StudentRegistration>
```

5. **Initialise npm**

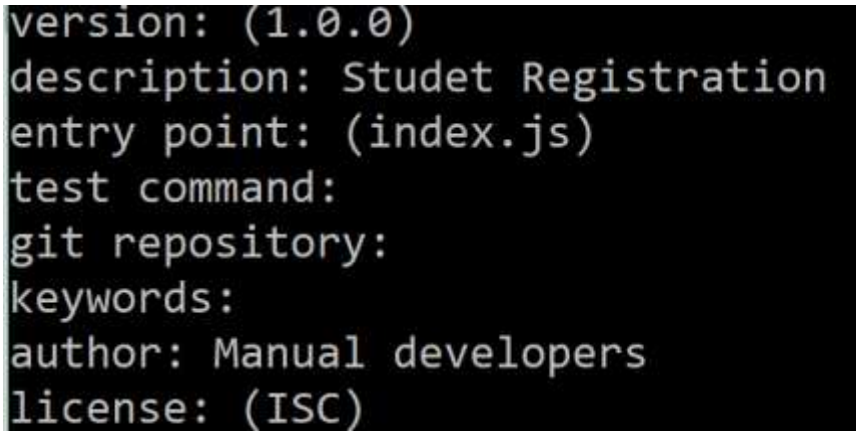
Once inside your project directory, initialize npm (Node Package Manager) by using npm init or npm init –y which creates a package. json file:

npm init is interactive and allows customization, while **npm init -y** is non-interactive and uses default values to quickly create a package. json file.



```
D:\StudentRegistration>npm init
```

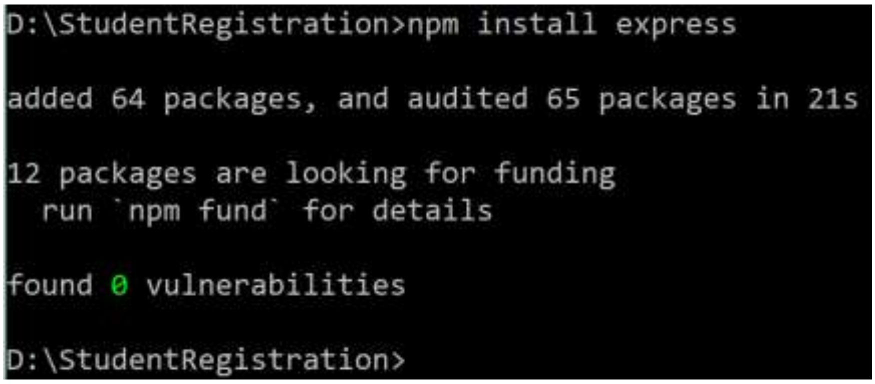
Then press on Enter and customise your project accordingly (such as package name, version, description, entry point usually index.js, test command, git repository, keywords, author and license.



```
version: (1.0.0)
description: Studet Registration
entry point: (index.js)
test command:
git repository:
keywords:
author: Manual developers
license: (ISC)
```

6. **Install express and Nodemon**

Install Express (a web framework for Node.js) and Nodemon (a utility that monitors for changes in your source code and restarts your server):



```
D:\StudentRegistration>npm install express

added 64 packages, and audited 65 packages in 21s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

D:\StudentRegistration>
```

For installing nodemon replace the code by npm install nodemon

NB: You can install all necessary packages at the same time by writing them in single line of code

Ex: npm install express nodemon mysql2

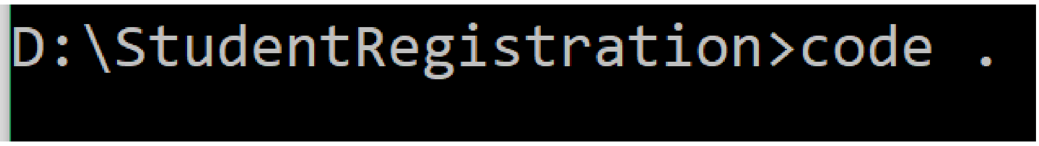
For that case it will insatall express, nodemon and mysql2.

Once you install express js in your project folder it come up with a new folder called node_modules.

7. **Open the created project in vs code.**

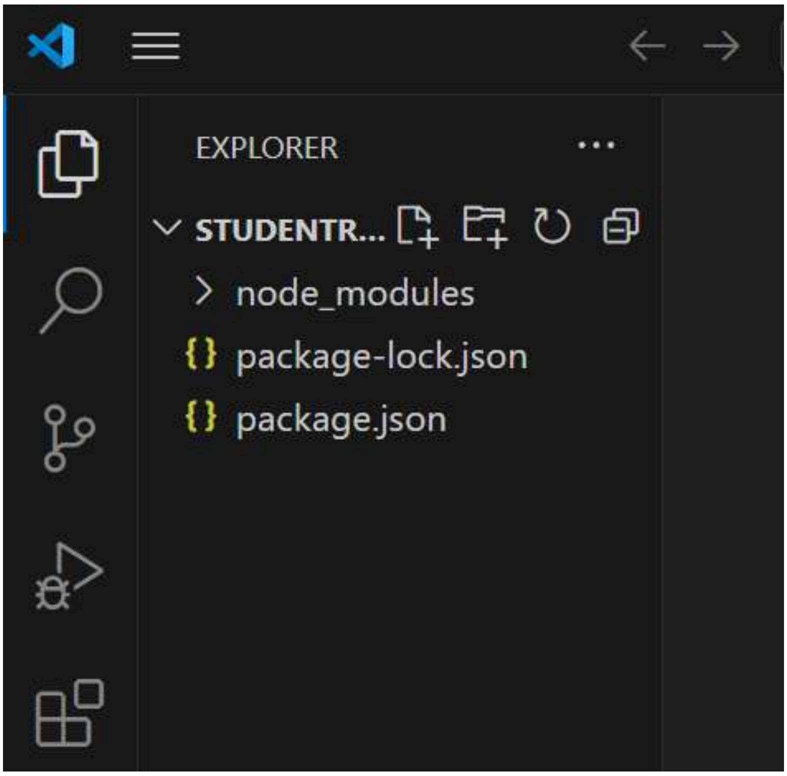
Open Visual Studio Code (ensure it's installed on your system).

Navigate to your project directory in VS Code



```
D:\StudentRegistration>code .
```

It will open your project in visual studio code as shown bellow.



Practical Activity 1.1.3: Installation and configuration of Postman

Task:

1. Read key reading 1.1.3 and ask clarification where necessary

2. You are requested to go to the computer lab to install and configure postman as used in documenting and testing APIs.

3. Apply safety precautions

5. Referring to the steps provided in key readings, install and configure postman.

6. Present your work to the trainer and whole class.

Key readings 1.1.3: Installation and configuration of Postman

•To install and configure postman you follow the following steps:

1. **Download the Installer from official website or use it installer file on external device**

   Go to the official Postman website (https://www.postman.com/downloads/) and download the installer suitable for your operating system (Windows, macOS, Linux).

2. **Run the Installer**

   Once the download is complete, run the installer file. Follow the installation prompts to complete the installation process.

3. **Launch Postman**

   After installation, launch Postman from your desktop or applications folder.

4. **Configure the basic settings like user account (Email, Username and password) or sign up using google or sign in with SSO.**

✓ Upon launching Postman for the first time, you'll be prompted to sign in or sign up.

✓ If you have an existing Postman account, enter your email and password to sign in.

✓ If you don't have an account, you can sign up using your email address or sign in with Google or a Single Sign-On (SSO) provider.

✓ Follow the on-screen instructions to complete the sign-up or sign-in process.

✓ Once signed in, you can configure basic settings such as default environments, workspace preferences, and more from the Postman interface.

✓ Postman is now installed and configured. You can start using it to create, test, and manage APIs.

Practical Activity 1.1.4: Configuration of MySQL Server

Task:

1. Read key reading 1.1.4 and ask clarification where necessary

2. Referring to the previous theoretical activities (1.1.1) you are requested to go to the computer lab to configure MySQL Server installed on your computer.

3. Apply safety precautions

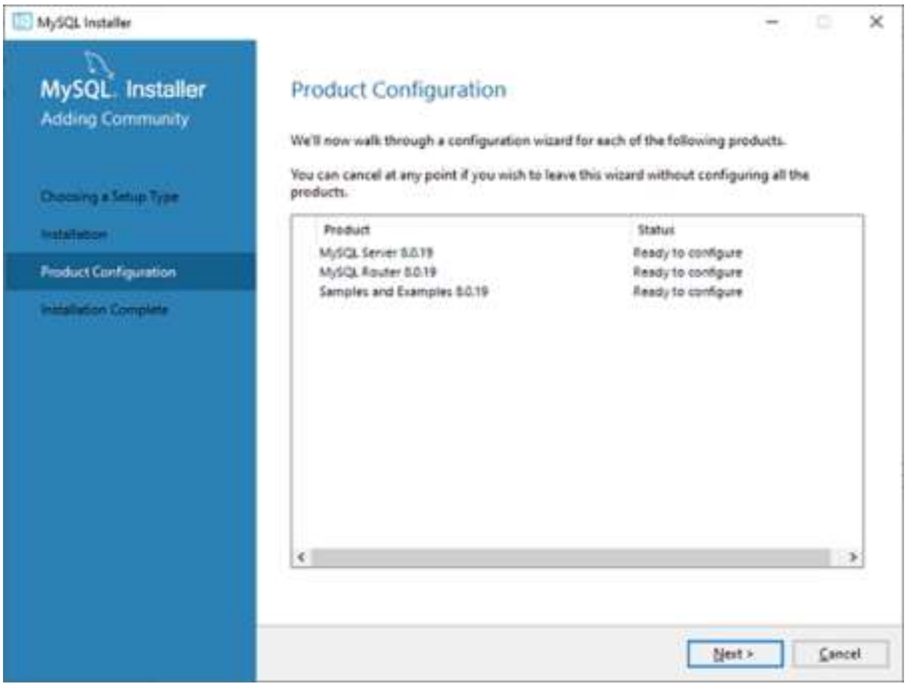
4. Referring to the steps provided in key readings, configure MySQL.

5. Present your work to the trainer and whole class.

Key readings 1.1.4: 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.



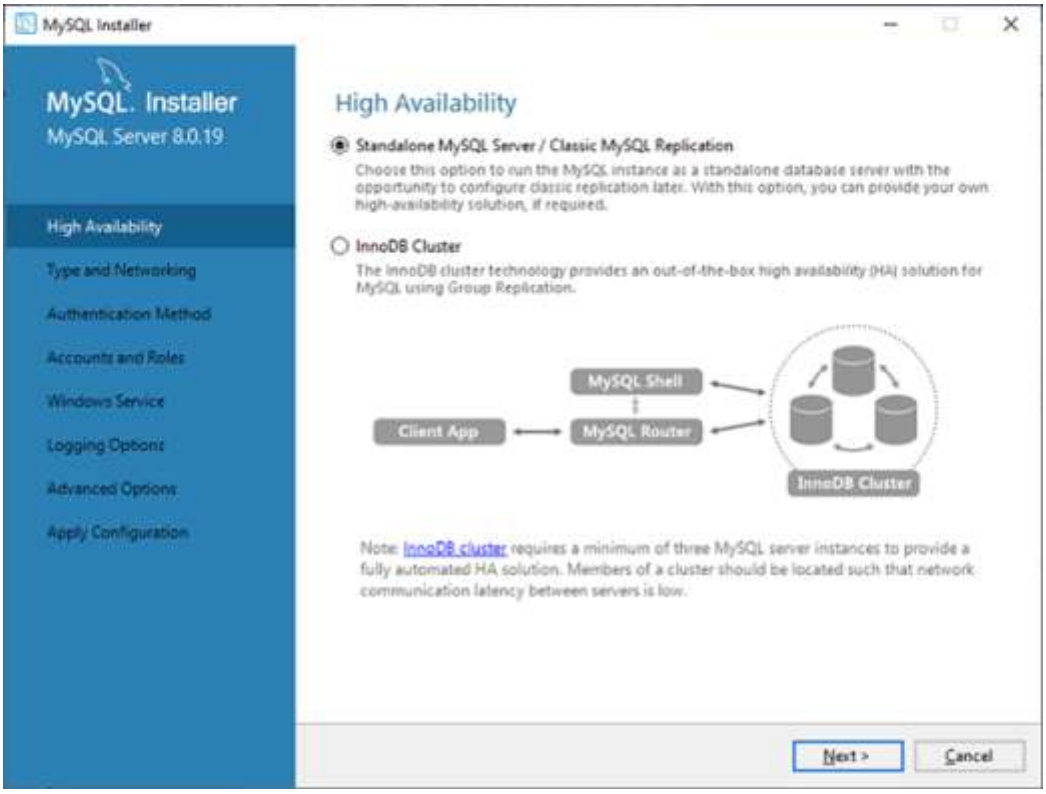
Product Configuration

We'll now walk through a configuration wizard for each of the following products.

You can cancel at any point if you wish to leave this wizard without configuring all the products.

Product	Status
MySQL Server 8.0.19	Ready to configure
MySQL Router 8.0.19	Ready to configure
Samples and Examples 8.0.19	Ready to configure

✓ Choose the appropriate configuration type. "**Standalone MySQL Server**" is suitable for most users.



High Availability

● Standalone MySQL Server / Classic MySQL Replication

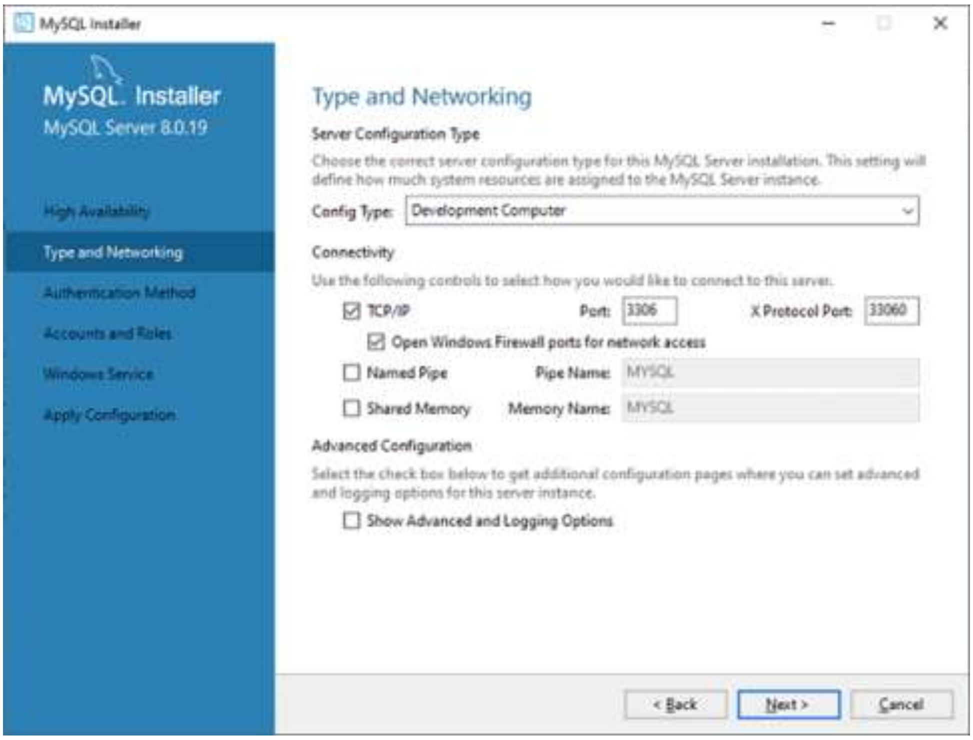
Choose this option to run the MySQL instance as a standalone database server with the opportunity to configure classic replication later. With this option, you can provide your own high-availability solution, if required.

○ InnoDB Cluster

The InnoDB cluster technology provides an out-of-the-box high availability (HA) solution for MySQL using Group Replication.

Note: InnoDB cluster requires a minimum of three MySQL server instances to provide a fully automated HA solution. Members of a cluster should be located such that network communication latency between servers is low.

- ✓ Set up the server configuration details:
- ✓ **Config Type:** Select "Development Machine", "Server Machine", or "Dedicated Machine" based on your use case.



Type and Networking

Server Configuration Type

Choose the correct server configuration type for this MySQL Server installation. This setting will define how much system resources are assigned to the MySQL Server instance.

Config Type: Development Computer

Connectivity

Use the following controls to select how you would like to connect to this server.

☑ TCP/IP Port: 3306 X Protocol Port: 33060

☑ Open Windows Firewall ports for network access

□ Named Pipe Pipe Name: MYSQL

□ Shared Memory Memory Name: MYSQL

Advanced Configuration

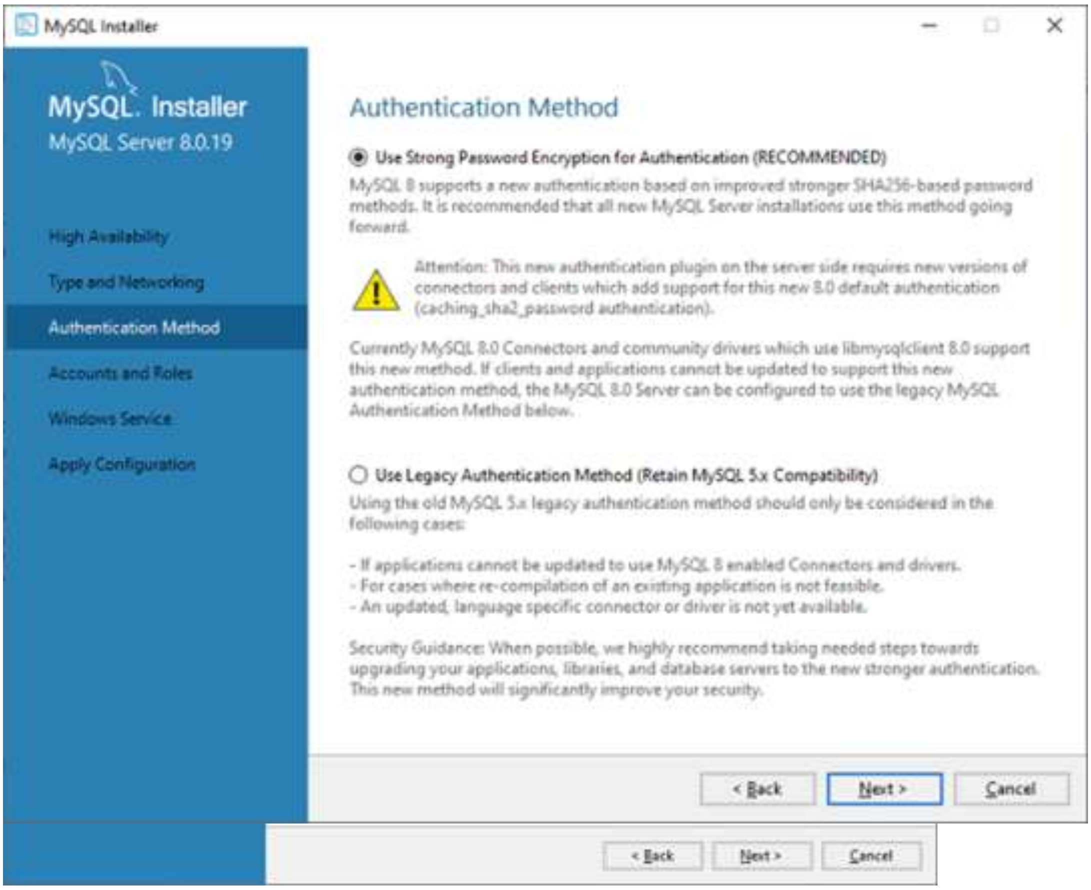
Select the check box below to get additional configuration pages where you can set advanced and logging options for this server instance.

□ Show Advanced and Logging Options

- ✓ **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).

✓



Authentication Method

● Use Strong Password Encryption for Authentication (RECOMMENDED)

MySQL 8 supports a new authentication based on improved stronger SHA256-based password methods. It is recommended that all new MySQL Server installations use this method going forward.

⚠ Attention: This new authentication plugin on the server side requires new versions of connectors and clients which add support for this new 8.0 default authentication (caching_sha2_password authentication).

Currently MySQL 8.0 Connectors and community drivers which use libmysqlclient 8.0 support this new method. If clients and applications cannot be updated to support this new authentication method, the MySQL 8.0 Server can be configured to use the legacy MySQL Authentication Method below.

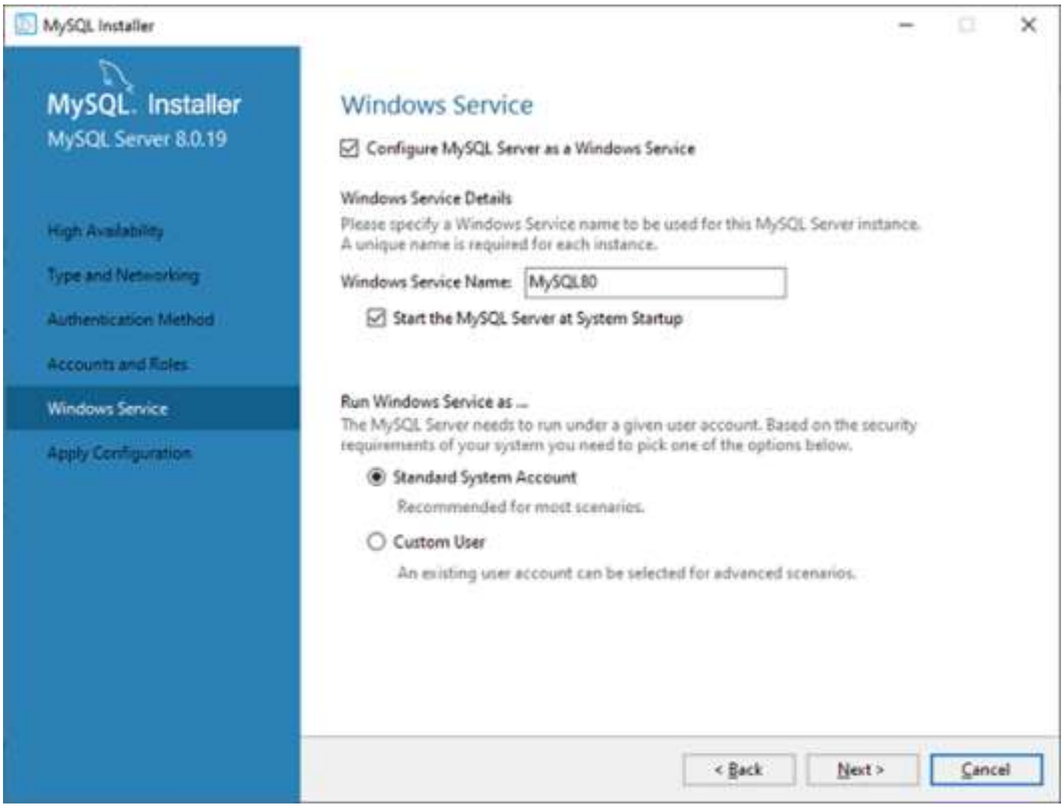
○ Use Legacy Authentication Method (Retain MySQL 5.x Compatibility)

Using the old MySQL 5.x legacy authentication method should only be considered in the following cases:

- If applications cannot be updated to use MySQL 8 enabled Connectors and drivers.
- For cases where re-compilation of an existing application is not feasible.
- An updated, language specific connector or driver is not yet available.

Security Guidance: When possible, we highly recommend taking needed steps towards upgrading your applications, libraries, and database servers to the new stronger authentication. This new method will significantly improve your security.

✓ **Windows Service:** Configure MySQL Server to run as a Windows service. You can also choose to start the server at system start up.



Windows Service

☑ Configure MySQL Server as a Windows Service

Windows Service Details

Please specify a Windows Service name to be used for this MySQL Server instance. A unique name is required for each instance.

Windows Service Name: MySQL80

☑ Start the MySQL Server at System Startup

Run Windows Service as ...

The MySQL Server needs to run under a given user account. Based on the security requirements of your system you need to pick one of the options below.

● Standard System Account

Recommended for most scenarios.

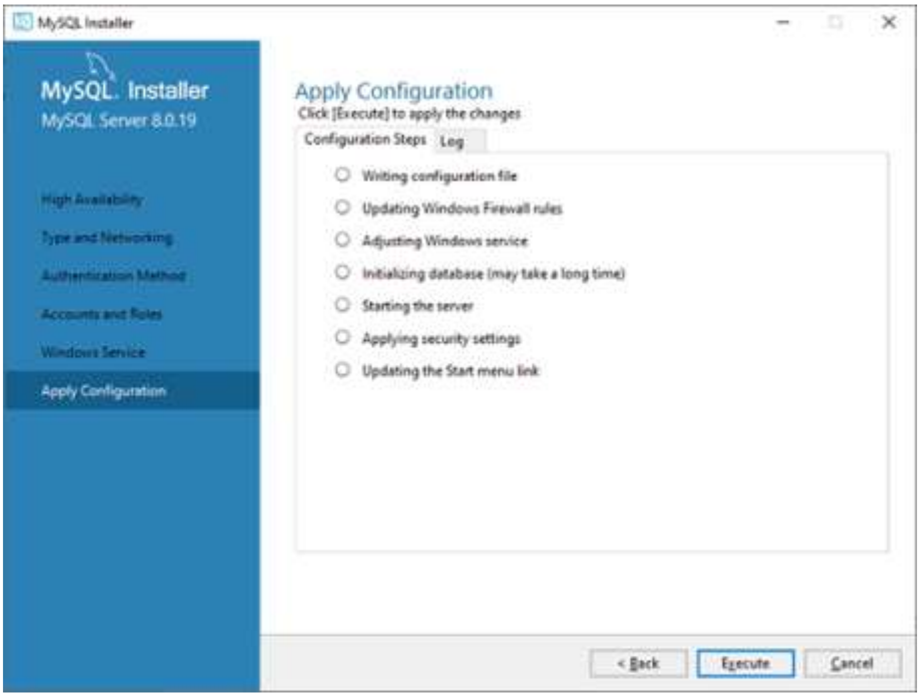
○ Custom User

An existing user account can be selected for advanced scenarios.

- ✓ **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.



Apply Configuration

Click [Execute] to apply the changes

○ Writing configuration file

○ Updating Windows Firewall rules

○ Adjusting Windows service

○ Initializing database (may take a long time)

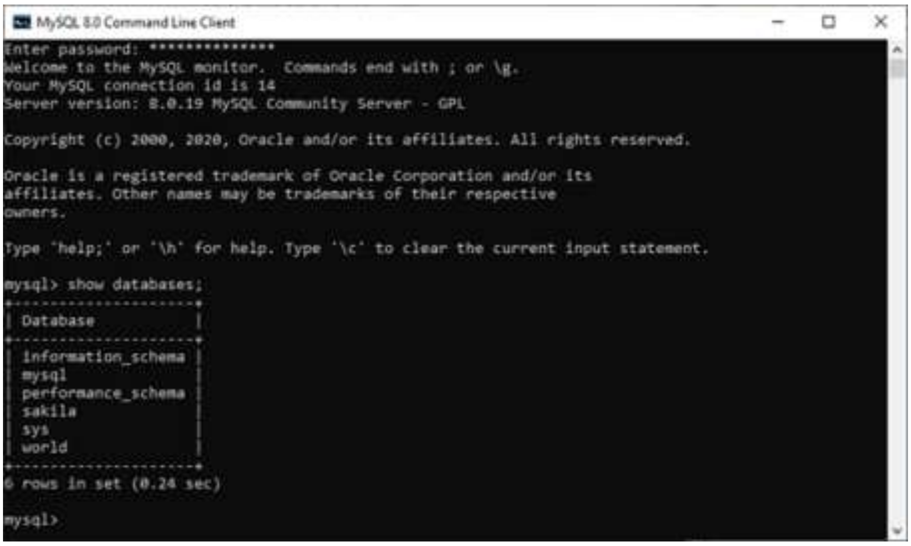
○ Starting the server

○ Applying security settings

○ Updating the Start menu link

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.



```
Enter password: **************
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14
Server version: 8.0.19 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sakila             |
| sys                |
| world              |
+--------------------+
6 rows in set (0.24 sec)

mysql>
```

Points to Remember

- Node.js is an open-source JavaScript runtime environment that allows developers to execute JavaScript code on the server side. Once you install node js it come up with it default package manager (NPM) that is used to install necessary packages and dependencies like express js and Nodemon.
- In development of backend application, you have to include concepts of object-oriented programming (OOP) like class, object, methods and properties. And as it deals with server side it has to be connected to database such as SQL based or NoSQL based one.
- The backend application has to respond to the HTTP methods via routes and once connecting to frontend there is the application of APIs and those ones are tested and documented by using postman.
- Once you want to develop a backend application you follow the following starting steps:
    - ✓ Install node js
    - ✓ Check if node and npm are installed
    - ✓ Make directory as project name
    - ✓ Navigate to created directory
    - ✓ Initialise npm
    - ✓ Install Express and Nodemon
    - ✓ Open the created project in vs code.
- To install and configure Postman you follow the following steps:
    - ✓ Download the Installer from the official website or use it installer file on an external device
    - ✓ Run the Installer
    - ✓ Launch Postman
    - ✓ Configure the basic settings like user account (Email, Username, and password) or sign-up using Google or sign in with SSO.
- To configure MySQL Server in Windows you follow those steps:
    - ❖ Set up the server configuration details like:
        - ✓ Config Type
        - ✓ Connectivity
        - ✓ Authentication
        - ✓ Accounts and Roles

    - ✓ Windows Service

- ❖ Apply Configuration
- ❖ Complete the Installation
- ❖ Verify Installation

Application of learning 1.1.

STU LTD is a software development company located in Musanze district, it develops software both frontend and backend for different companies, due to different activities that they have, you are hired as a backend developer in node js responsible for setting the working environment by creating the project folder (OurWebinfo) on local disk D, and set all other necessary settings to be ready for development of back end in node js, testing and documenting APIs.

All tools, materials, and equipment will be provided by the company.

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

Duration: 10 hrs

Theoretical Activity 1.2.1: Descriprion of client libraries and ES5 or ES6

Tasks:

1. You are requested to discuss the following:

   i. Categories and Usability of Node.js Client Libraries

   ii. Difference between ES5 from ES6

2. Provide your findings and write them on paper.

3. Present the findings to the whole class

4. For more clarification, read the key readings 1.2.1. ask questions where necessary.

Key readings 1.2.1.: Description of client libraries and ES5 or ES6

Node.js has a vast ecosystem of client libraries available through NPM (Node Package Manager). These libraries cover a wide range of functionalities, from HTTP requests to database access, and even utility functions. The number of libraries is enormous, with over a million packages available on NPM, and new ones being added regularly.

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: An ORM for Node.js that 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.

3. **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.

Practical Activity 1.2.2: Creation of basic server with Express Js

Task:

1. Read key reading 1.2.2

2. You are requested to go to the computer lab to create a basic server with express js.

3. Apply safety precautions

4. Referring to the steps provided in key readings, create a basic server with express js.

5. Present your work to the trainer and the whole class

Key readings 1.2.2: Creation of basic server with Express Js

Creating a basic server with Express.js involves a few straightforward steps. Express.js is a minimal and flexible Node.js web application framework that provides robust features for building web and mobile applications.

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. sends ('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. sends ('This is the About page.');
});

// Contact Route
app.get ('/contact', (req, res) => {
res. sends ('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 ();
});
```

Practical Activity 1.2.3: Application of Client Libraries

Task:

1. Read key reading 1.2.3

2. Referring to the previous practical activities (1.2.2.) you are requested to go to the computer lab to lab to apply client libraries in existing project developed on activity 1.2.2.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, apply client libraries in existing project
5. Present your work to the trainer and whole class

Key readings 1.2.3: Application of Client Libraries

1. HTTP and HTTPS Modules

Node.js has built-in `http` and `https` modules that allow you to make HTTP and HTTPS requests without needing to install any additional libraries.

Example using `http`:

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

http.get ('http://example.com', (resp) => {

let data = '';

// A chunk of data has been received.

resp. on('data', (chunk) => {

data += chunk;

});


// The whole response has been received.

resp. on ('end', () => {

console.log(data);

  });

}).on("error", (err) => {

  console.log("Error: " + err.message);

});
```

Example using `https`:

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

```
https.get('https://example.com', (resp) => {
  let data = '';
  resp.on('data', (chunk) => {
    data += chunk;
  });
  resp.on('end', () => {
    console.log(data);
  });
}).on("error", (err) => {
  console.log("Error: " + err.message);
});
```

2. Axios

Axios is a popular promise-based HTTP client for the browser and Node.js. It provides an easy-to-use API for making requests.

Example using Axios:

First, you need to install Axios if you haven't already:

```
npm install axios
```

Then, you can use it in your `server.js`:

```
const axios = require('axios');
axios.get('https://example.com')
then (response => {
console.log(response.data);
})
catch (error => {
console. error ('Error:', error);
});
```

3. Request

The Request library was once a popular choice for making HTTP requests, but it has been deprecated. However, if you're working on a legacy project that still uses it, here's how you can implement it.

Example using Request:

First, install the Request library:

```
npm install request
```

Then, you can use it in your `server.js`:

```
const request = require('request');
request ('https://example.com', (error, response, body) => {
if (! error && response. statusCode == 200) {
console.log(body); // Print the response body
} else {
console.error('Error:', error);
}
});
```

Practical Activity 1.2.4: Establishment and Test of server connection

Task:

1. Read key reading 1.2.4
2. Referring to the previous practical activities (1.2.3.) you are requested to go to the computer lab to establish and test server connection.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, establish and test server connection
5. Present your work to the trainer and whole class

Key readings 1.2.4: Establishment and Test of server connection

Establishing and testing a server connection in a Node.js application involves several steps.

Below, is general approach to set up a simple server and test the connection using the libraries we discussed in key readings 1.2.3.

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.

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 http = require('http');

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. setHeader ('Content-Type', 'text/plain');

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

f 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.

4. Using Axios

You can also create another script to test the server using Axios. Create a file named `test.js`:

```
// test.js
const axios = require('axios');
axios.get('http://127.0.0.1:3000')
then (response => {console.log ('Response from server:', response.data);
})
catch (error => {
console. error ('Error:', error);
});
```

Run this script in your terminal:

```
node test.js
```

Step 5: 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.

Points to Remember

- Node.js has a vast ecosystem of client libraries available through NPM (Node Package Manager). These libraries cover a wide range of functionalities, from HTTP requests to database access, and even utility functions. The number of libraries is enormous, with over a million packages available on NPM, and new ones being added regularly.
- Creating a basic server with Express.js involves different steps.

Here are steps to Create a Basic Server with Express.js

- ❖ Install Node.js and NPM
- ❖ Initialize a New Node.js Project
- ❖ Install Express.js
- ❖ Create the Server File
- ❖ Write Basic Express.js Server Code
- ❖ Run the Server
- ❖ Test the Server

- There are different libraries in express js but as developer you have to apply them depending on the project structure. Here are the basic ones
  - ✓ Use the built-in `**http**` and `**https**` modules for basic requests.
  - ✓ Use **Axios** for a more modern and flexible approach to making HTTP requests.
  - ✓ The **Request** library is deprecated, but you may encounter it in older projects.
- While establishing and testing the server you have to follow those steps.

1. Set up your Node.js environment and create a new project.

2. Create a simple server using the built-in `http` module.
3. Run the server and test the connection using a browser, cURL, Postman, or Axios.
4. Handle any errors and debug as necessary.

Application of learning 1.2.

STU LTD is a software development company located in Musanze district, it develops software both frontend and backend for different companies, due to different activities that they have, you are hired as backend developer in node js responsible for creating the project folder (**OurWebinfo**) on local disk D, and establish server file that will be used as entry point including http and axios as client libraries.

All tools, materials and equipment will be provided by the company.

Indicative content 1.3: Establishment of Database Connection

Duration: 10 hrs

Practical Activity 1.3.1: Establishment of database connection

Task:

1. Read key reading 1.3.1
2. Referring to the previous practical activities (1.2.4.) you are requested to go to the computer lab to establish database connection.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, establish database connection
5. Present your work to the trainer and whole class

Key readings 1.3.1: 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 `**db.js**` to handle the database connection.
Here’s how you can configure it:

```
const mysql = require('mysql2');

// Create a connection to the database
const connection = 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
});

// Connect to the database
connection. connect((err) => {
if (err) {
console. error ('Error connecting to the database:', err. stack);
return;
}
console.log ('Connected to the database as ID', connection.threadId);
});

module. exports = connection;
```

Step 5: Test Database Connection

You can test the database connection by creating a simple script. Create a file named `testConnection.js`:

```
const connection = require('./db');
```

```
// Test the connection
connection. query ('SELECT 1 + 1 AS solution', (err, results) => {
if (err) {
console. error ('Error executing query:', err);
return;
}
console.log ('The solution is:', results[0].solution);
});

// Close the connection
connection.end ();
```

Step 6: Run the Test Script

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

```
node testConnection.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 (which should be `2`).

Points to Remember

- In Establishment of database connection, you have to follow those steps:

  1. Install MySQL package: Use `npm install mysql2`.

  2. Create a database: Use SQL commands in a MySQL client.

  3. Set up a schema: Create tables using SQL commands.

  4. Configure database connection: Use `mysql2` to set up a connection in a Node.js file.

  5. Test the connection: Run a simple query to ensure the connection works.

Application of learning 1.3.

TelaTech ltd is a company that develop websites for different institutions they have tasked you to develop for them a database (**KigaliinnovationDB**) and inside create a table called clients with the following fields ID, Names, Sex, Address, Phone and Email in node js and set the basic connections that will be used while creating APIs?

Indicative content 1.4: Develop RESTFUL APIs

Duration: 15 hrs

Practical Activity 1.4.1: Develop endpoints and HTTP Methods

Task:

1. Read key reading 1.4.1
2. As backend developer who has trained in database development you are requested to go to the computer lab to define endpoints and HTTP Methods.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, define endpoints and HTTP Methods.
5. Present your work to the trainer and whole class.

Key readings 1.4.1: 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;

// Middleware to parse JSON bodies

app.use(express. json());

// Sample in-memory data store

let items = [];

// 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
app.post ('/items', (req, res) => {
const newItem = {
id: items. length + 1, name: req.body.name,
};
items. push(newItem);
res. status(201).json(newItem); // Respond with the created item
});
```

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
});
```

3. Create Specific ID GET Endpoint

This endpoint retrieves a specific item by its ID.

```
// Create GET endpoint for a specific item by ID
app.get ('/items/:id', (req, res) => {
const itemId = parseInt (req.params.id, 10);
const item = items. find (i => i.id === itemId);
if (! item) {
return res. status (404). json({ message: 'Item not found' });
}
res. json(item); // Respond with the found item
});
```

4. Create PUT Endpoint

This endpoint updates an existing item by its ID.

```
// Create PUT endpoint for updating an item
app.put ('/items/:id', (req, res) => {
const itemId = parseInt (req.params.id, 10);
const item = items.find(i => i.id === itemId);
if (!item) {
return res.status(404).json({ message: 'Item not found' });
}
// Update the item
item.name = req.body.name;
res.json(item); // Respond with the updated item
});
```

5. Create DELETE Endpoint

This endpoint deletes an item by its ID.

```
// Create DELETE endpoint for removing an item
app. delete ('/items/:id', (req, res) => {
const itemId = parseInt(req.params.id, 10);
const itemIndex = items.findIndex(i => i.id === itemId);
if (itemIndex === -1) {
return res.status(404).json({ message: 'Item not found' });
}
items.splice(itemIndex, 1); // Remove the item from the array
res.status(204).send(); // Respond with no content
});
```

Full Example

Here’s the complete code for `server.js` with all the endpoints defined:

```
const express = require('express');
const app = express();
const PORT = 3000;
app.use(express.json());
let items = [];
// Create POST endpoint
app.post('/items', (req, res) => {
const newItem = {
id: items.length + 1,
name: req.body.name,
};
items.push(newItem);
res.status(201).json(newItem);
});
// Create GET endpoint for all items
app.get('/items', (req, res) => {
res.json(items);
});
// Create GET endpoint for a specific item by ID
app.get('/items/:id', (req, res) => {
const itemId = parseInt(req.params.id, 10);
const item = items.find(i => i.id === itemId);
```

```
if (!item) {
return res.status(404).json({ message: 'Item not found' });
}
res.json(item);
});
// Create PUT endpoint for updating an item
app.put('/items/:id', (req, res) => {
const itemId = parseInt(req.params.id, 10);
const item = items.find(i => i.id === itemId);
if (!item) {
return res.status(404).json({ message: 'Item not found' });
}
item.name = req.body.name;
res.json(item);
});
// Create DELETE endpoint for removing an item
app.delete('/items/:id', (req, res) => {
const itemId = parseInt(req.params.id, 10);
const itemIndex = items.findIndex(i => i.id === itemId);
if (itemIndex === -1) {
return res.status(404).json({ message: 'Item not found' });
}
items.splice(itemIndex, 1);
res.status(204).send();
});
// Start the server
app.listen(PORT, () => {
```

```
console.log(`Server is running on http://localhost:${PORT}`);

});
```

Step 3: Testing the Endpoints

Practical Activity 1.4.2: Testing the Endpoints using Postman

Task:

1. Read key reading 1.4.2.
2. As a backend developer, you are requested to go to the computer lab to test the developed Endpoints on activity 1.4.1 using Postman.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, test the Endpoints using Postman.
5. Present your work to the trainer and the whole class.

Key readings 1.4.2: 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

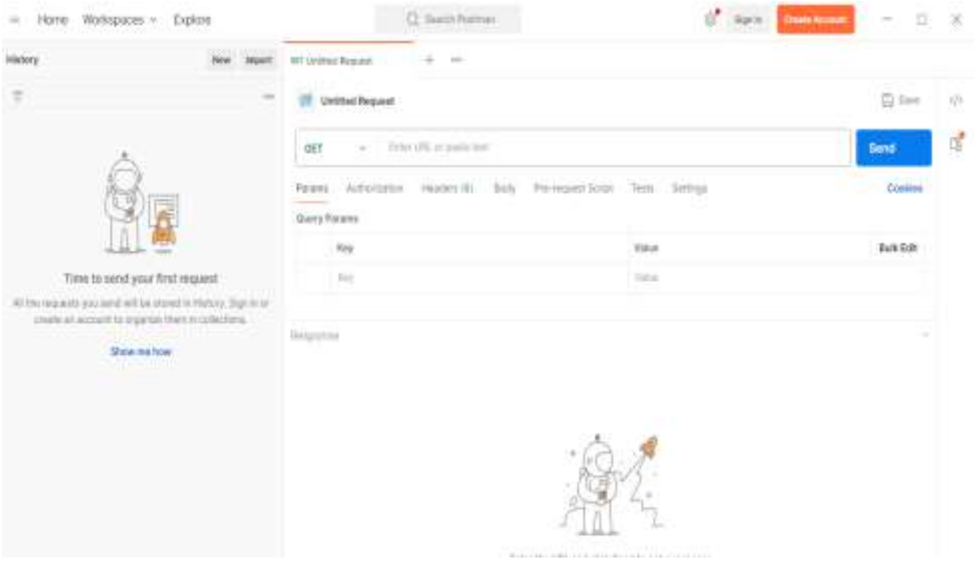
Before testing, ensure that your Express server is running. Open your terminal and navigate to your project directory, then run:

```
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.



Time to send your first request

All the requests you send will be stored in History. Sign in or create an account to organize them in collections.

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:

```
json

{

   "name": "Item 1"

}
```

- ✓ 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. **Get a Specific Item (GET)**

- ✓ Select GET from the dropdown menu.
- ✓ Enter the URL: `http://localhost:3000/items/1` (replace `1` with the ID of the item you want to retrieve).
- ✓ Click the Send button. You should see a response with the specific item if it exists.

4. **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:

```
json

{

"name": "Updated Item 1"

}
```

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

5. **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 also view the status codes to ensure that your requests are being handled correctly (e.g., `200 OK`, `201 Created`, `404 Not Found`, etc.).

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.

Practical Activity 1.4.3: Implementation of API endpoints

Task:

1. Read key reading 1.4.3.
2. Refers to practical activity 1.4.2 you are requested to go to the computer lab to implement API endpoints.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, to implement API endpoints.
5. Present your work to the trainer and whole class.

Key readings 1.4.3: Implementation of API endpoints

Implementing API endpoints in Node.js using Express involves several steps, from setting up your environment to writing the code for your endpoints.

Below, is step-by-step guide along with sample code to help implement a simple RESTful API.

Step 1: Set Up Your Environment

1. Create a New Project Directory

2. Initialize a New Node.js Project

3. Install Express

Step 2: Create the Server File

Step 3: Write the Basic Server Code

Basic structure for your `server.js` file:

```
const express = require('express');
const app = express();
const PORT = 3000;
// Middleware to parse JSON bodies
app.use(express.json());
// Sample in-memory data store
let items = [];
// Start the server
app.listen(PORT, () => {
console.log(`Server is running on http://localhost:${PORT}`);
});
```

Step 4: Define API Endpoints

Now, let’s add the API endpoints to the `server.js` file.

1. Create a POST Endpoint

This endpoint will allow you to create a new item.

```
// Create POST endpoint
app.post('/items', (req, res) => {
const newItem = {
id: items.length + 1,
name: req.body.name,
};
items.push(newItem);
res.status(201).json(newItem); // Respond with the created item
});
```

2. Create a GET Endpoint for All Items

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
});
```

3. Create a GET Endpoint for a Specific Item by ID

This endpoint retrieves a specific item by its ID.

```
// Create GET endpoint for a specific item by ID
app.get('/items/:id', (req, res) => {
const itemId = parseInt(req.params.id, 10);
const item = items.find(i => i.id === itemId);
if (!item) {
return res.status(404).json({ message: 'Item not found' });
}
res.json(item); // Respond with the found item
});
```

4. Create a PUT Endpoint for Updating an Item

This endpoint updates an existing item by its ID.

```
// Create PUT endpoint for updating an item
app.put('/items/:id', (req, res) => {
const itemId = parseInt(req.params.id, 10);
const item = items.find(i => i.id === itemId);
if (!item) {
return res.status(404).json({ message: 'Item not found' });
}
// Update the item
item.name = req.body.name;
res.json(item); // Respond with the updated item
});
```

5. Create a DELETE Endpoint for Removing an Item

This endpoint deletes an item by its ID.

```
// Create DELETE endpoint for removing an item

app.delete('/items/:id', (req, res) => {

const itemId = parseInt(req.params.id, 10);

const itemIndex = items.findIndex(i => i.id === itemId);
```

Theoretical Activity 1.4.4: Description of middleware services

Tasks:

1: You are requested to answer the following questions related to the description of middleware services:

i. Explain the term middleware.

ii. Describe the use of middleware services

iii. Describe types of middleware services

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 1.4.4. ask questions where necessary.

Key readings 1.4.4: Description of middleware services

1. **Middleware services**
   In Express.js are functions that execute during the request-response cycle. They can modify the request and response objects, end the request-response cycle, and call the next middleware function in the stack.
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:** Middleware that is bound to an instance of the app. 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
```

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 router = express.Router();
router.use(express.json()); // Apply middleware to all routes in this router
```

3. **Error-Handling Middleware:**

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

```
app.use((err, req, res, next) => {
console.error(err.stack);
res.status(500).send('Something broke!');
});
```

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

5. **Third-Party Middleware:** Middleware created by the community, such as `morgan` for logging and `cors` for handling Cross-Origin Resource Sharing.

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 robust 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.

Practical Activity 1.4.5: Use middleware services

Task:

1. Read key reading 1.4.5
2. Refers to theoretical activity 1.4.4 you are requested to go to the computer lab to use middleware services.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, use middleware services.
5. Present your work to the trainer and whole class.

Key readings 1.4.5: Use middleware services

1. Error Handling Middleware

Error handling middleware is used to catch and manage errors that occur during the request-response cycle. It helps ensure that your application can gracefully handle unexpected situations without crashing.

Define an error-handling middleware function at the end of your middleware stack. It should have four parameters: `err`, `req`, `res`, and `next`.

Example

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

const app = express();

const PORT = 3000;

// Sample route

app.get('/', (req, res) => {

throw new Error('Something went wrong!'); // Simulate an error

});

// Error handling middleware

app.use((err, req, res, next) => {
```

```
console.error(err.stack); // Log the error stack

res.status(500).json({ message: 'Internal Server Error' }); // Send a response with a 500 status

});

app.listen(PORT, () => {

console.log(`Server is running on http://localhost:${PORT}`);

});
```

2. Logging Middleware

Purpose: Logging middleware records information about each incoming request, such as the HTTP method, URL, status code, response time, and more. This helps in monitoring and debugging applications.

Implementation:

- Logging middleware can log data to the console or save it to a file or external logging service.
- Libraries like morgan are commonly used for logging HTTP requests in Node.js applications.

Example using morgan:

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

app.use(morgan('combined')); // Logs requests in Apache combined format
```

- Custom logging middleware can also be created to log specific information.

```
app.use((req, res, next) => {

console.log(`${req.method} ${req.url}`);

next();

});
```

3. Input Validation Middleware

Purpose: Input validation middleware ensures that the data coming into the application (via request bodies, query parameters, or headers) meets the required format, type, and constraints. This is crucial for preventing security vulnerabilities like SQL injection and ensuring data integrity.

Implementation:

- Libraries like **express-validator** provide tools to validate and sanitize user inputs easily.
- Input validation can be done by defining rules and running them against the incoming data before passing it to the next middleware or route handler.

Example using express-validator:

```
const { body, validationResult } = require('express-validator');
app.post('/register', [
body('email').isEmail().withMessage('Enter a valid email'),
body('password').isLength({ min: 5 }).withMessage('Password must be at least 5 characters long')
], (req, res, next) => {
const errors = validationResult(req);
if (!errors.isEmpty()) {
return res.status(400).json({ errors: errors.array() });
}
next();
});
```

- Custom validation middleware can be written to handle specific validation logic.

Practical Activity 1.4.6: Perform CRUD operations using MySQL Database

Task:

1. Read key reading 1.4.6.
2. As backend, developer who has skills in database development you have given task to go in computer lab to perform CRUD operation using MySQL in node js.
3. Referring to the steps provided in key readings, perform CRUD operation using MySQL in node js.
4. Present your work to the trainer and whole class.

Key readings 1.4.6: Perform CRUD operations using MySQL Database

- **To perform CRUD operation in node js you have to follow the following steps**

1. **Setup Your Node.js Environment**

   Install Node.js: Make sure Node.js is installed on your machine

2. **Initialize a Node.js Project**

```
mkdir manualdevdb-project

cd manualdevdb-project

npm init -y
```

3. **Install Required Packages**

   mysql2, express

```
npm install express mysql2
```

4. **Create a MySQL Connection**

   Create a db.js file to establish a connection to your MySQL database.

```
// db.js

const mysql = require('mysql2');

const connection = mysql.createConnection({

host: 'localhost',

user: 'your_mysql_username',

password: 'your_mysql_password',

database: 'manualdevdb'

});

connection.connect((err) => {

if (err) {

console.error('Error connecting to the database:', err.stack);

return;

  }
console.log('Connected to the MySQL database.');

});

module.exports = connection;
```

5. Set Up Express Server

Create an index.js file to set up an Express server and define CRUD routes.

```
// index.js

const express = require('express');

const connection = require('./db');

const app = express();

app.use(express.json()); // To parse JSON bodies

const PORT = process.env.PORT || 3000;

// 1. CREATE: Add a new developer

app.post('/developers', (req, res) => {

const { Names, sex, phone, district, school, trade, module, Degree, accountnumber } = req.body;

const sql = 'INSERT INTO developers (Names, sex, phone, district, school, trade, module, Degree, accountnumber) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)';

connection.query(sql, [Names, sex, phone, district, school, trade, module, Degree, accountnumber], (err, results) => {

if (err) {

return res.status(500).send(err);

}

res.status(201).send('Developer added successfully!');

});

});

// 2. READ: Get all developers

app.get('/developers', (req, res) => {

const sql = 'SELECT * FROM developers';
```

```
connection.query(sql, (err, results) => {
if (err) {
return res.status(500).send(err);
}
res.status(200).json(results);
});
});
// 3. READ: Get a developer by ID
app.get('/developers/:id', (req, res) => {
const { id } = req.params;
const sql = 'SELECT * FROM developers WHERE id = ?';
connection.query(sql, [id], (err, results) => {
if (err) {
return res.status(500).send(err);
}
res.status(200).json(results[0]);
});
});
// 4. UPDATE: Update a developer by ID
app.put('/developers/:id', (req, res) => {
const { id } = req.params;
const { Names, sex, phone, district, school, trade, module, Degree, accountnumber } = req.body;
const sql = 'UPDATE developers SET Names = ?, sex = ?, phone = ?, district = ?, school = ?, trade = ?, module = ?, Degree = ?, accountnumber = ? WHERE id = ?';
connection.query(sql, [Names, sex, phone, district, school, trade, module, Degree, accountnumber, id], (err, results) => {
if (err) {
```

```
return res.status(500).send(err);
}
res.status(200).send('Developer updated successfully!');
});
});
// 5. DELETE: Delete a developer by ID
app.delete('/developers/:id', (req, res) => {
const { id } = req.params;
const sql = 'DELETE FROM developers WHERE id = ?';
connection.query(sql, [id], (err, results) => {
if (err) {
return res.status(500).send(err);
}
res.status(200).send('Developer deleted successfully!');
});
});
// Start the server
app.listen(PORT, () => {
console.log(`Server is running on port ${PORT}`);
});
```

6. Run Your Application

Start your Node.js application:

```
node index.js
```

7.Test the CRUD operations using tools like Postman.

Sample CRUD Requests

Create (POST):

Endpoint: http://localhost:3000/developers

Body (JSON):

```
{
"Names": "John Peace",
"sex": "Male",
"phone": "1234567890",
"district": "Kigali",
"school": "BTICTHUB",
"trade": "Software Development",
"module": "Module 1",
"Degree": "Bachelor",
"accountnumber": "123456789"
}
```

Read All (GET):

Endpoint: http://localhost:3000/developers

Read by ID (GET):

Endpoint: http://localhost:3000/developers/1

Update (PUT):

Endpoint: http://localhost:3000/developers/1

Body (JSON):

```
{
"Names": "Jane Peace",
"sex": "Female",
"phone": "0987654321",
"district": "Kigali",
"school": "BTICTHUB",
"trade": "Web Development",
"module": "Module 2",
"Degree": "Masters",

"accountnumber": "987654321"

}
```

Delete (DELETE):

Endpoint: http://localhost:3000/developers/1

Practical Activity 1.4.7: Use HTTP Status code

Task:

1. Read key reading 1.4.7.
2. Refers to the practical activity 1.4.6, you are asked to go in computer lab to use HTTP status code in existing project developed on activity 1.4.6.
3. Referring to the steps provided in key readings, use HTTP status code
4. Present your work to the trainer and whole class.

Key readings 1.4.7: Use HTTP Status code

To implement proper HTTP status codes in the CRUD operations using Node.js and MySQL, you should modify the response handling in each route.

Here's how you can implement it:

Updated index.js with HTTP Status Codes

Example containing sample codes

```
// index.js

const express = require('express');

const connection = require('./db');

const app = express();

app.use(express.json()); // To parse JSON bodies

const PORT = process.env.PORT || 3000;
```

```
// 1. CREATE: Add a new developer

app.post('/developers', (req, res) => {

const { Names, sex, phone, district, school, trade, module, Degree, accountnumber } = req.body;

const sql = 'INSERT INTO developers (Names, sex, phone, district, school, trade, module, Degree, accountnumber) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)';

connection.query(sql, [Names, sex, phone, district, school, trade, module, Degree, accountnumber], (err, results) => {

if (err) {

return res.status(500).json({ error: 'Failed to add developer', details: err });

}

res.status(201).json({ message: 'Developer added successfully!', developerId: results.insertId });

});

});

// 2. READ: Get all developers

app.get('/developers', (req, res) => {

const sql = 'SELECT * FROM developers';

connection.query(sql, (err, results) => {

if (err) {

return res.status(500).json({ error: 'Failed to retrieve developers', details: err });

}

res.status(200).json(results);

});

});

// 3. READ: Get a developer by ID

app.get('/developers/:id', (req, res) => {

const { id } = req.params;

const sql = 'SELECT * FROM developers WHERE id = ?';
connection.query(sql, [id], (err, results) => {
if (err) {
return res.status(500).json({ error: 'Failed to retrieve developer', details: err });
}
if (results.length === 0) {
return res.status(404).json({ error: 'Developer not found' });
}
res.status(200).json(results[0]);
});
});
// 4. UPDATE: Update a developer by ID
app.put('/developers/:id', (req, res) => {
const { id } = req.params;
const { Names, sex, phone, district, school, trade, module, Degree, accountnumber } = req.body;
const sql = 'UPDATE developers SET Names = ?, sex = ?, phone = ?, district = ?, school = ?, trade = ?, module = ?, Degree = ?, accountnumber = ? WHERE id = ?';
connection.query(sql, [Names, sex, phone, district, school, trade, module, Degree, accountnumber, id], (err, results) => {
if (err) {
return res.status(500).json({ error: 'Failed to update developer', details: err });
}
if (results.affectedRows === 0) {
return res.status(404).json({ error: 'Developer not found' });
}
res.status(200).json({ message: 'Developer updated successfully!' });
});
});
```

```
// 5. DELETE: Delete a developer by ID
app.delete('/developers/:id', (req, res) => {
const { id } = req.params;
const sql = 'DELETE FROM developers WHERE id = ?';
connection.query(sql, [id], (err, results) => {
if (err) {
return res.status(500).json({ error: 'Failed to delete developer', details: err });
}
if (results.affectedRows === 0) {
return res.status(404).json({ error: 'Developer not found' });
}
res.status(200).json({ message: 'Developer deleted successfully!' });
});
});
// Start the server
app.listen(PORT, () => {
console.log(`Server is running on port ${PORT}`);
});
```

✓ **Explanation of HTTP Status Codes**

- **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.

Practical Activity 1.4.8: Debugging RESTFUL APIs

Task:

1. Read key reading 1.4.8.
2. Refers to the practical activity 1.4.7, you are asked to go in computer lab to debug RESTFUL APIs of existing project developed on activity 1.4.7.
3. Apply safety precautions.
4. Referring to the steps provided in key readings, debug RESTFUL APIs
5. Present your work to the trainer and whole class.

Key readings 1.4.8: Debugging RESTFUL APIs

Debugging a RESTful API can be an essential part of development, ensuring that your API behaves as expected.

Steps to Debug a RESTful API

Step 1: Set Up Your Development Environment

1. **Use a Reliable Code Editor:** Choose an editor like Visual Studio Code, which has excellent support for JavaScript and Node.js.

2. **Install Debugging Tools:** Make sure you have Node.js installed and consider using tools like Postman or cURL for testing API endpoints.

Step 2: Implement Logging

1. **Add Logging to Your API:** Use `console.log()` statements or a logging library like `winston` or `morgan` to log requests, responses, and errors.

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

const morgan = require('morgan');

const app = express();

app.use(morgan('dev')); // Logs HTTP requests

app.get('/api/example', (req, res) => {

console.log('Received request for /api/example');

res.json({ message: 'Hello, World!' });

});
```

2. **Log Errors:** Make sure to log errors with sufficient detail, including stack traces if necessary.

```
app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).send('Something broke!');

});
```

Step 3: Use a Debugger

1. **Node.js Debugger:** You can use the built-in Node.js debugger. Start your server with the `inspect` flag:

```
node inspect server.js
```

Then, you can connect to it using Chrome DevTools or Visual Studio Code.

2. **Set Breakpoints:** In your code editor, set breakpoints in your API routes to inspect variable states and flow.

Step 4: Test API Endpoints

1. **Use Postman or cURL:** Send requests to your API endpoints and inspect the

responses. Look for:

- Status codes (e.g., 200, 404, 500)
- Response body
- Headers

2. **Check Request Payloads:** Ensure that the data you send in requests (for POST or PUT) is correctly formatted.

Step 5: Validate Input Data

1. **Input Validation:** Ensure that your API validates incoming data. Use libraries like `Joi` or `express-validator` to validate request bodies.

```
const { body, validationResult } = require('express-validator');
app.post('/api/data', [
body('name').isString().notEmpty(),
], (req, res) => {
const errors = validationResult(req);
if (!errors.isEmpty()) {
return res.status(400).json({ errors: errors.array() });
}
// Handle valid data
});
```

Step 6: Monitor Network Traffic

1. **Browser Developer Tools:** If your API is called from a web application, use the Network tab in your browser's developer tools to inspect requests and responses.

2. **Check for CORS Issues:** If your API is accessed from a different origin, watch for Cross-Origin Resource Sharing (CORS) issues in the browser console.

Step 7: Review API Documentation

1. **Check API Specifications:** Ensure that your API endpoints, methods, and expected request/response formats match your documentation (e.g., OpenAPI/Swagger).

2. **Update Documentation:** Keep your API documentation up to date as you

make changes.

Step 8: Use Automated Testing

1. **Write Unit Tests:** Use testing frameworks like Mocha, Chai, or Jest to write unit tests for your API endpoints.

2. **Integration Tests:** Test the interaction between different parts of your application to ensure they work together correctly.

```
const request = require('supertest');
const app = require('./server');
describe('GET /api/example', () => {
it('should return Hello, World!', (done) => {
request(app)
get('/api/example')
.expect('Content-Type', /json/)
expect(200, { message: 'Hello, World!' }, done);
});
});
```

Step 9: Analyze Performance

1. **Use Profiling Tools:** Monitor the performance of your API using tools like New Relic, Datadog, or built-in Node.js profiling tools.

2**. Identify Bottlenecks:** Look for slow responses or excessive resource usage and optimize your code accordingly.

Points to Remember

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

  Step 1: Set Up Your Express Server

  Step 2: Define Endpoints

  Step3: Testing the Endpoints

- Using Postman, you can easily test all the endpoints of your Express.js application by performing the following actions:

  1. Open Postman
  2. Select the HTTP method (GET, POST, PUT, DELETE).
  3. Enter the appropriate URL.
  4. Provide any necessary request body in JSON format.
  5. Click Send and observe the response.

- Implementing API endpoints in Node.js using Express involves several steps, from setting up your environment to writing the code for your endpoints.
- Middleware services in Express.js are functions that execute during the request-response cycle. They can modify the request and response objects, end the request-response cycle, and call the next middleware function in the stack.
- In node js we have different types of middleware services but that are used depending on the application to be developed, among them we can use the followings:
  - **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.
- To perform CRUD operation in node js you have to follow the following steps

  - ✓ Setup Your Node.js Environment
  - ✓ Initialize a Node.js Project
  - ✓ Install Required Packages:
  - ✓ Create a MySQL Connection
  - ✓ Set Up Express Server that contains all operations included in CRUD (Create, Read, Update and Delete as endpoints)
  - ✓ Run Your Application
  - ✓ Test the CRUD operations using tools like Postman.

- HTTP Status Codes can be classified as the followings:

  - ✓ **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.

- Debugging a RESTful API can be an essential part of development, ensuring that your API behaves as expected. Here’s a step-by-step guide to help you debug a RESTful API effectively:

    1. Set up your development environment and implement logging.
    2. Use a debugger to set breakpoints and inspect your code.
    3. Test API endpoints with tools like Postman.
    4. Validate input data and monitor network traffic for issues.
    5. Review and update your API documentation.
    6. Write automated tests for your API endpoints.
    7. Analyse performance to identify bottlenecks.

Application of learning 1.4.

TelaTech ltd is a company that develop websites for different institutions they have tasked you to develop for them a database (**KigaliinnovationDB**) and inside create a table called clients with the following fields ID, Names, Sex, Address, Phone and Email in node js and develop APIs that will be used to insert, update, select and delete client’s records via frontend where you will be provide the developed APIs to be integrated with front end.

Include http status codes in order to overcome error handling issues

Learning Outcome 1 End assessment

Written assessment

I. Answer by True to the correct statements or False to the wrong statements

1. The POST HTTP method is typically used to retrieve data from a server.

2. Middleware in Express.js can be used to modify the request and response objects.

3. A 404 Not Found status code indicates that the requested resource was successfully created.

4. In Express.js, the next () function is used to pass control to the next middleware function in the stack.

5. Debugging a RESTful API involves reviewing and updating the API documentation as one of the steps.

6. CRUD operations in Node.js using Express.js include creating, reading, updating, and deleting resources.

II. Complete the following with the correct world

7. To define endpoints in an Express.js application, you must first ____________.

8. The HTTP status code __________ is used when a server-side error occurs, such as a database query failing.

9. In Node.js, the __________ middleware checks if the data in the request meets necessary requirements.

10. When testing API endpoints with Postman, you should first select the __________.

11. A __________ is a tool in Express.js used to perform tasks like logging, input validation, and error handling during the request-response cycle.

12. The 201 Created status code is used when __________.

13. Once you wont to develop a back-end application the followings are requirements except select one option?

a) Node js installed on your computer
b) Npm
c) Nodemon
d) Install MS word
e) Express
f) Postman

14. Refers to postman in back end application development Respond by TRUE or False to the followings?

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

15. Nodemon is a utility tool for Node.js that helps developers during the development process Respond by TRUE or FALSE.

16. Node.js is an open-source JavaScript runtime environment that allows developers to execute algorithm code on the server side. Respond by TRUE or False

17. Postman is a popular tool for developing and documenting APIs. Respond by TRUE or False

18. What will be performed once you use the following command

mkdir ubudeheproject

19. Expand the followings:

    a) API:
    b) REST:
    c) NPM:
    d) DBMS:

20. Dependencies are internal packages or libraries that a Node.js application relies on to perform various tasks. Respond by TRUE or False

21. What is the command that can be used to install express as node package?

22. What is command that can be used to install Nodemon as global?

23. What command is used to initialize a Node.js project?

    A) npm start
    B) npm init -y
    C) node init
    D) npm create

24. Which package is used to parse incoming request bodies in Express?

    A) express
    B) body-parser
    C) mysql
    D) nodemon

25. What HTTP method is typically used to retrieve data from a server?

A) POST
B) PUT
C) GET
D) DELETE

26. What status code indicates that a resource was successfully created?

A) 404
B) 200
C) 500
D) 201

27. Which middleware function is used to log details of incoming requests?

A) Error Handling Middleware
B) Input Validation Middleware
C) Logging Middleware
D) Authentication Middleware

28. What is the correct method to delete a resource in REST?

A) POST
B) DELETE
C) GET
D) PUT

29. Which command installs Express in your Node.js project?

A) npm install express
B) npm add express
C) npm get express
D) npm express install

30. What is the purpose of Error Handling Middleware?

A) To log requests
B) To validate input
C) To handle errors and send appropriate responses
D) To connect to the database

31. Which HTTP status code indicates a resource was not found?

A) 200
B) 201
C) 404
D) 500

32. What is the default port number for an Express server if not specified?

A) 8080
B) 3000
C) 5000
D) 4000

33. Match the HTTP methods of column B with their typical usage of column C

Column A Answers	Column B Methods	Column C Usage
A……….	A) GET	1) Retrieve data
B……….	B) POST	2) Update data
C……….	C) PUT	3) Delete data
D……….	D) DELETE	4) Create new resource

34. Match the middleware of column B to its function in column C

Column A answers	Middleware	Function
A……….	A) Logging Middleware	1) Logs details of requests
B………	B) Input Validation	2) Validates incoming data
C……….	C) Error Handling	3) Catches errors and sends responses

Complet the followings with correct key word from the listed one below (POST, end, npm install mysql, 500 Internal Server Error, Postman, Logging, 404 Not Found, 200 OK, Error Handling, Create, Read, Update, and Delete, PUT, listen,meets,201 Created,MySQL Connection,GET,body-parser,validate)

35. To create a new resource, you typically use the __________ HTTP method.

36. In Express.js, middleware functions can modify the request and response objects, __________ the request-response cycle, and call the next middleware function.

37. The command to install the MySQL package in a Node.js project is __________.

38. When a server encounters an unexpected issue, it typically responds with a __________ status code.

39. To test API endpoints, you can use a tool called __________.

40. The __________ middleware is responsible for logging the details of incoming requests.

41. A resource that does not exist in the database should return a __________ status code.

42. The status code __________ indicates that a request was successful and data is returned.

43. To handle errors in an Express application, you can implement __________ middleware.

44. CRUD operations in Node.js require setting up an Express server that contains endpoints for __________.

45. The method used to update an existing resource is typically __________.

46. An Express server can be started by calling the __________ method on the app object.

47. The purpose of input validation middleware is to ensure that incoming data __________ the necessary requirements.

48. A successful resource creation in a RESTful API returns a __________ status code.

49. To connect to a MySQL database in Node.js, you need to create a __________.

50. The __________ method is used to request data from a specified resource.

51. In Express, to parse JSON data from incoming requests, you must use the __________ middleware.

52. When debugging a RESTful API, it is important to __________ input data and monitor network traffic for issues.

Practical assessment

Tela Tech ltd is a company that develop websites for different institutions they have tasked you to develop for them a database (**Kigali innovation DB**) and inside create a table called clients with the following fields ID, Names, Sex, Address, Phone and Email in node js and develop APIs that will be used to insert, update, select and delete client's records via frontend where you will be provide the developed APIs to be integrated with front end.

Include http status codes in order to overcome error handling issues

END

References

Books

Doglio, F. (2018). *Rest API Development with Node.Js.* Uruguay: Apress.

Mardan, A. (2018). *Practical Node.js.* California: Apress.

Mike Cantelon, M. H. (2014). *Node.js IN ACTION.* Shelter Island, NY11964: Manning Shelter Island.

Web site links

Nael, D. (2019, March 11). Retrieved from okta: https://developer.okta.com/blog/2019/03/11/node-sql-server

Palmer, S. (2024). *how-to-build-a-secure-web-application-with-nodejs*. Retrieved from devteam: https://www.devteam.space/blog/how-to-build-a-secure-web-application-with-nodejs/

Syed, B. A. (2014). *Beginning Node.js.* New york: Apress.

W3schools. (2024). *nodejs_get_started.asp*. Retrieved from w3schools: https://www.w3schools.com/nodejs/nodejs_get_started.asp

Watmore, J. (2022, January 01). *nodejs-ms-sql-server-simple-api-for-authentication-registration-and-user-management*. Retrieved from jasonwatmore: https://jasonwatmore.com/post/2022/07/01/nodejs-ms-sql-server-simple-api-for-authentication-registration-and-user-management

REST API
Using Node.js
API
node
TEST
MOCHA
chai
JS
node
JS
API
WEB APPLICATION SECURITY
BEST PRACTICES

Indicative contents

2.1 Data encryption in securing RESTFUL APIs

2.2 Integrating and Using Third-Party Libraries

2.3 Maintaining and Updating Third-Party Libraries

2.4 Implementation of Authentication, Authorization, and Accountability

2.5 Secure, Monitor, and Manage Environment Variables

Key Competencies for Learning Outcome 2: Secure Backend Application

Knowledge	Skills	Attitudes
● Introduction to data encryption ● Description of authentication, authorization, and accountability	● Applying data encryption ● Checking Third-party libraries ● Applying User Authentication, Authorization and Accountability (AAA) ● Securing Environment variables	● Having Team work spirit ● Critical thinker ● Being Innovative ● Being creative ● Practical oriented ● Detail oriented ● Being honesty ● Having a passion for Learning ● problem-Solving Mindset ● Attention to Security ● Ethical Coding

Duration: 20 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Introduce correctly data encryption as used in nodejs backend application security
2. Describe perfectly authentication, authorization, and accountability as used securing nodejs backend application
3. Apply correctly data encryption based on system security
4. Checking perfectly Third-party libraries based on system security
5. Apply effectively user Authentication, Authorization and Accountability (AAA) based on NPM Universal Access Control (UAC)
6. Secure correctly environment variables according to system security requirements

Resources

Equipment	**Tools**	**Materials**
● Computer	● Browser ● Node.Js ● Text Editor ● Express. Js ● Postman ● Git ● Swagger	● Internet ● Electricity

Indicative content 2.1: Data Encryption in Securing RESTFUL APIs

Duration: 3 hrs

Theoretical Activity 2.1.1: Introduction to data encryption

Tasks:

1: You are requested to answer the following questions related to the Introduction to data encryption:

i. Define the following terms:

- Data encryption
- Hashing

ii. Differentiate the types of data encryption.

iii. Explain encryption techniques used in node js.

iv. Explain the benefits of data encryption.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.1.1. In addition, ask questions where necessary.

Key readings 2.1.1.: Introduction to data encryption

1. **Data encryption**

Data encryption is the process of converting plaintext data into a coded format (ciphertext) to prevent unauthorized access. It uses algorithms and keys to transform the original data, ensuring that only authorized users with the correct decryption key can access the original information. Encryption is commonly used to protect sensitive data such as passwords, personal information, and financial records during transmission or storage.

Hashing

Hashing is the process of converting input data (or a message) into a fixed-size string of characters, which is typically a sequence of numbers and letters. This transformation is done using a hash function, which produces a unique hash value (or digest) for each unique input. Hashing is primarily used for data integrity verification and password storage. Unlike encryption, hashing is a one-way process, meaning it cannot be reversed to retrieve the original data. Even a small change in the input will produce a significantly different hash value.

2. **Types of Data Encryption**

Symmetric Encryption

Symmetric encryption uses the same key for both encryption and decryption.

Example in Node.js: The AES (Advanced Encryption Standard) algorithm is commonly used for symmetric encryption. Node.js provides built-in support for AES through the crypto module.

Use Case: Symmetric encryption is typically used for encrypting large amounts of data quickly, such as securing data at rest.

Asymmetric Encryption

Definition: Asymmetric encryption uses a pair of keys—one public and one private. The public key encrypts the data, while the private key decrypts it.

Example in Node.js: The RSA (Rivest-Shamir-Adleman) algorithm is a widely used asymmetric encryption method. Node.js's crypto module also supports RSA.

Use Case: Asymmetric encryption is commonly used for securing small amounts of data, such as encrypting a session key or digital signatures.

Hashing

Definition: Hashing is a one-way encryption technique that converts data into a fixed-size hash value. It is not intended for decryption.

Example in Node.js: The crypto module supports various hashing algorithms like SHA-256, which can be used for hashing passwords or verifying data integrity.

Use Case: Hashing is often used for storing passwords securely and verifying the integrity of data files.

3. **Encryption Techniques in Node.js**

Using the crypto Module

Node.js has a built-in crypto module that provides various cryptographic functions, including encryption, decryption, and hashing. Developers can use this module to implement both symmetric and asymmetric encryption, as well as hashing techniques.

Third-Party Libraries

bcrypt: A popular library for hashing passwords securely. It automatically handles the addition of salt to the hashing process, making it more secure.

jsonwebtoken (JWT): Used for implementing JSON Web Tokens, a form of asymmetric encryption that securely transmits information between parties as a JSON object.

4. **Benefits and Importance of Data Encryption**

Data Protection

Encryption ensures that sensitive data remains confidential, even if it is intercepted or accessed by unauthorized individuals. This is particularly important for protecting user credentials, personal information, and financial data.

Compliance with Regulations

Many industries are subject to data protection regulations (e.g., GDPR, HIPAA) that require the encryption of sensitive information. Implementing encryption in Node.js applications help organizations comply with these legal requirements.

Data Integrity

Encryption not only protects data from unauthorized access but also ensures that the data has not been tampered with. This is crucial for maintaining the integrity of data during transmission and storage.

User Trust

Users are more likely to trust applications that prioritize the security of their personal information. By implementing encryption, Node.js developers can build trust and credibility with their users.

Mitigating Risks

In the event of a data breach, encrypted data is much harder for attackers to exploit. This reduces the potential damage and liability for the organization, making encryption a key component of a robust security strategy.

Practical Activity 2.1.2: Securing RESTFUL APIs

Task:

1. Read key reading 2.1.2.

2. Referring to the previous theoretical activities (2.1.1) and practical activities (1.4.8) you are requested to go to the computer lab to secure RESTFUL APIs: install the Crypto Module, generate key for data Encryption, encrypt data, convert data to buffers, store the encrypted data.

3. Apply safety precautions

4. Refer to the steps provided in key readings, secure developed APIs.

5. Present out the Steps of securing RESTFUL APIs.

Key readings 2.1.2: Securing RESTful APIs in Node.js

Securing RESTful APIs is crucial for protecting sensitive data and ensuring that only authorized users can access the API's resources. Below are the steps to secure RESTful APIs in Node.js, with a focus on data encryption using the crypto module.

1. Install the crypto Module

The crypto module is built into Node.js and provides cryptographic functionality, including encryption, decryption, and hashing. Since it's a core module, there's no need to install it separately. You can require it directly in your Node.js application.

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

2. Create a Key for Encryption

To encrypt data, you need a secret key. This key can be generated using the crypto module's randomBytes function. The length of the key depends on the encryption algorithm you choose (e.g., 32 bytes for AES-256).

Example:

```
const key = crypto.randomBytes(32); // 256-bit key for AES-256
const iv = crypto.randomBytes(16);  // Initialization vector for CBC mode
```

3. Use the Key to Encrypt Data

Once you have the key, you can use it to encrypt data. Choose an encryption algorithm (e.g., AES-256-CBC) and create a cipher object using the **crypto.createCipheriv** method.

Example:

```
const algorithm = 'aes-256-cbc';
const encrypt = (text) => {
let cipher = crypto.createCipheriv(algorithm, key, iv);
let encrypted = cipher.update(text, 'utf-8', 'hex');
encrypted += cipher.final('hex');
return encrypted;
};
```

4. Convert the Data to a Buffer

Before encrypting, you may need to convert the data (if it's not already a string or buffer) to a buffer, which is a binary representation of the data. This is important for encrypting non-string data types like images or files.

Example:

```
const buffer = Buffer.from('Hello, World!', 'utf-8');
```

5. Encrypt the Data

Using the cipher object created earlier, you can now encrypt the data. The result will be in hexadecimal format, which is commonly used to represent encrypted data in a more readable form.

Example:

```
const encryptedData = encrypt(buffer.toString());
console.log('Encrypted Data:', encryptedData);
```

6. Store the Encrypted Data

After encrypting the data, you'll typically need to store it securely in a database or file system. Ensure that the storage medium is also secure and follows best practices for protecting sensitive data.

Example:

```
const fs = require('fs');
fs.writeFileSync('encryptedData.txt', encryptedData, 'utf-8');
console.log('Encrypted data saved to file');
```

Full Example

Here’s how all the steps come together in a full example:

```
const crypto = require('crypto');
const fs = require('fs');
// 1. Generate encryption key and initialization vector
const key = crypto.randomBytes(32);  // 256-bit key
const iv = crypto.randomBytes(16);   // Initialization vector
// 2. Function to encrypt data
const encrypt = (text) => {
   const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);
   let encrypted = cipher.update(text, 'utf-8', 'hex');
   encrypted += cipher.final('hex');
   return encrypted;
};
// 3. Convert data to buffer
const buffer = Buffer.from('Hello, World!', 'utf-8');
// 4. Encrypt the data
const encryptedData = encrypt(buffer.toString());
console.log('Encrypted Data:', encryptedData);
// 5. Store the encrypted data
fs.writeFileSync('encryptedData.txt', encryptedData, 'utf-8');
console.log('Encrypted data saved to file');
// To decrypt the data, use a similar process with `crypto.createDecipheriv`
```

```
const decrypt = (encryptedText) => {
  const decipher = crypto.createDecipheriv('aes-256-cbc', key, iv);
  let decrypted = decipher.update(encryptedText, 'hex', 'utf-8');
  decrypted += decipher.final('utf-8');
  return decrypted;
};
const decryptedData = decrypt(encryptedData);
console.log('Decrypted Data:', decryptedData);
```

Explanation

Key Generation: A 256-bit key and a 16-byte initialization vector (IV) are generated for AES-256 encryption.

Buffer Conversion: Data is converted to a buffer to ensure it's in the correct format for encryption.

Encryption: Data is encrypted using AES-256-CBC and stored in hexadecimal format.

Storage: The encrypted data is saved to a file named encryptedData.txt.

Decryption (Optional): Decryption can be done using the same key and IV to verify that the data was encrypted and decrypted correctly.

Points to Remember

- Data encryption is essential in Node.js applications for safeguarding sensitive information such as user credentials and payment data.
- Encryption transforms plaintext into ciphertext, which is unreadable without the proper decryption key.
- There are three main types of encryptions used in Node.js: symmetric encryption (e.g., AES), which uses the same key for both encryption and decryption; asymmetric encryption (e.g., RSA), which involves a public and private key pair; and hashing (e.g., SHA-256), which is a one-way encryption technique.
- Node.js provides built-in support for these through its crypto module, allowing developers to implement secure encryption and hashing techniques. Securing RESTful APIs involves using the crypto module to generate encryption keys, encrypt data, and store it securely.

- By implementing robust encryption practices, Node.js developers can protect data from unauthorized access, ensure compliance with regulations, and build user trust.

Application of learning 2.1.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients.

As a newly hired backend developer using Node.js, you have given developed APIs and your primary responsibility is to secure provided APIs by installing the crypto module, creating a key for encryption, Use the key to encrypt data, Convert the data to a buffer and Encrypt the data Store the encrypted data**.**

Indicative content 2.2: Integrating and Using Third-Party Libraries

Duration: 3 hrs

Practical Activity 2.2.1: Installing Node Js Package Manager (NPM)

Task:

1. Read key reading 2.2.1.

2. Referring to the previous theoretical activities (2.2.1) you are requested to go to the computer lab to perform all steps to install node js package manager (NPM)

3. Apply safety precautions

5. Refer to the work provided in key readings, install node js package manager (NPM).

4. Present out the steps for installation of node js package manager (NPM).

Key readings 2.2.1: Installing Node Js Package Manager (NPM)

Node.js Package Manager (NPM) is a powerful tool used for managing and installing libraries (packages) in Node.js projects. Below are the steps to install NPM along with Node.js.

Steps to Install Node.js and NPM:

1. **Download Node.js**
   - ✓ Visit the official Node.js website.
   - ✓ Download the appropriate installer for your operating system (Windows, macOS, or Linux).
2. **Install Node.js**
   - ✓ Run the downloaded installer.
   - ✓ Follow the installation prompts. Ensure that the option to install NPM is selected (NPM is bundled with Node.js by default).
   - ✓ Complete the installation.
3. **Verify Installation**
   - ✓ Open your terminal (Command Prompt for Windows, Terminal for macOS/Linux).
   - ✓ Type the following command to verify the installation of Node.js:

```
node -v
```

✓ Type the following command to verify the installation of NPM:

```
npm -v
```

4. **Updating NPM (Optional)**

If needed, update NPM to the latest version using the following command:

```
npm install -g npm@latest
```

5. **Initialize a Node.js Project**

✓ Navigate to your project directory in the terminal.

✓ Run the following command to initialize a new Node.js project:

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

✓ Follow the prompts to create a package.json file.

Practical Activity 2.2.2: Incorporating Common Node.js Third-Party Libraries.

Task:

1. Read key reading 2.2.2.

2. Referring to the previous practical activities (2.2.1) you are requested to go to the computer lab to incorporate common Node.js third-party libraries like Express, Lodash and Moment.js.

3. Apply safety precautions

4. Refer to the work provided in key readings, incorporate common Node.js third-party libraries.

5. Present out the incorporate common Node.js third-party libraries.

Key readings 2.2.2: Incorporating Common Node.js Third-Party Libraries.

Node.js projects often rely on third-party libraries to extend functionality. To incorporate three common libraries: Express, Lodash, and Moment.js.

1. Installing Express

Express is a web application framework for Node.js, designed to simplify the process of building web servers.

- Install Express using NPM:

```
npm install express
```

2. Installing Lodash

Lodash is a utility library that provides useful functions for common programming tasks, such as working with arrays, objects, and strings.

- Install Lodash using NPM:

```
npm install lodash
```

3. Installing Moment.js

Moment.js is a library for parsing, validating, manipulating, and displaying dates and times in JavaScript.

- Install Moment.js using NPM:

```
npm install moment
```

Verification

- After installing each library, check the package.json file to ensure they are listed under dependencies.
- Example structure in package.json:

```
{
"dependencies": {
"express": "^4.17.1",
"lodash": "^4.17.21",
"moment": "^2.29.1"
}
}
```

Practical Activity 2.2.3: Interacting with third-party libraries.

Task:

1. Read key reading 2.2.3.

2. Referring to the previous practical activities (2.2.2) you are requested to go to the computer lab to interact with third-party libraries like: Callbacks, Promises and async/await.

3. Apply safety precautions

4. Refer to the work provided in key readings, and then interact with third-party libraries.

5. Present out the interaction with third-party libraries.

Key readings 2.2.3: Interacting with third-party libraries.

Interacting with third-party libraries in Node.js often involves using callbacks, promises, and async/await for handling asynchronous operations.

1. Using Callbacks

Callback functions are passed into another function as an argument and are executed after an asynchronous operation is completed.

Example using Express

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

const app = express();

app.get('/', (req, res) => {

res.send('Hello World!');

});

app.listen(3000, () => {

console.log('Server is running on port 3000');

});
```

2. Using Promises

Promises provide a more flexible and readable way to handle asynchronous operations compared to callbacks.

Example using Lodash

```
const _ = require('lodash');

const data = [1, 2, 3, 4, 5];

const promise = new Promise((resolve, reject) => {

const result = _.shuffle(data);

if (result) {

resolve(result);

} else {

reject('Error shuffling data');

}

});

promise

  .then(result => console.log(result))

  .catch(error => console.log(error));
```

3. Using Async/Await

Async/Await is a modern syntax for working with promises in a more synchronous-looking manner.

Example using Moment.js

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

async function displayCurrentTime() {

const currentTime = await moment().format('YYYY-MM-DD HH:mm:ss');

console.log(`Current Time: ${currentTime}`);

}

displayCurrentTime();
```

Points to Remember

- Integrating third-party libraries in Node.js involves using NPM to manage and install packages like Express, Lodash, and Moment.js, which extend the functionality of applications.
- Developers interact with these libraries through asynchronous operations handled via callbacks, promises, or the more modern async/await syntax.

Application of learning 2.2.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients. As a newly hired backend developer using Node.js, you have given developed APIs and your primary responsibility is to Integrate third-party libraries in Node.js by using NPM to manage and install packages like Express, Lodash, and Moment.js, which extend the functionality of applications. And also interact with these libraries through asynchronous operations handled via callbacks, promises and async/await in a created project.

Indicative content 2.3: Maintaining and Updating Third-Party Libraries

Duration: 3 hrs

Practical Activity 2.3.1: Monitoring of library dependencies and version numbers

Task:

1. Read key reading 2.3.1.

2. Referring to the previous practical activities (2.2.3) you are requested to go to the computer lab to monitor library dependencies and version numbers in Package. Json and Npm-shrinkwrap. Json.

3. Apply safety precautions

5. Refer to the steps provided in key readings, monitor library dependencies and version numbers.

4. Present out how to monitor library dependencies and version numbers.

Key readings 2.3.1: Monitoring Library Dependencies and Version Numbers

Package.json and npm-shrinkwrap.json

Creating a package.json File:

Suppose you're starting a new Node.js project and need to add some dependencies:

```
npm init –y or npm init

npm install express lodash moment
```

This generates a package.json file like this:

```
{

"name": "my-nodejs-app",

"version": "1.0.0",

"dependencies": {
```

```
"express": "^4.17.3",

"lodash": "^4.17.21",

"moment": "^2.29.1"

}

}
```

Here, express, lodash, and moment are listed as dependencies. The caret (^) indicates that minor and patch versions will be updated but not major versions.

Using npm-shrinkwrap.json:

After ensuring that your project is working fine in development, you want to lock down the exact versions for production:

```
npm shrinkwrap
```

This generates an npm-shrinkwrap.json file, which might look like this:

```
{

"name": "my-nodejs-app",

"version": "1.0.0",

"lockfileVersion": 1,

"dependencies": {

"express": {

"version": "4.17.3",

"resolved": "https://registry.npmjs.org/express/-/express-4.17.3.tgz",

"integrity": "sha512-xyz...",

"requires": {

"debug": "~2.6.9",

"body-parser": "1.19.0"

}

},

"lodash": {

"version": "4.17.21",
"resolved": "https://registry.npmjs.org/lodash/-/lodash-4.17.21.tgz",
"integrity": "sha512-abc..."
},
"moment": {
"version": "2.29.1",
"resolved": "https://registry.npmjs.org/moment/-/moment-2.29.1.tgz",
"integrity": "sha512-def..."
}
}
}
```

This file locks down the specific versions of each dependency, ensuring the same versions are installed in any environment.

Practical Activity 2.3.2: Checking for library updates and security vulnerabilities

Task:

1. Read key reading 2.3.2.

2. Referring to the previous practical activities (2.2.3) you are requested to go to the computer lab to Check library updates and security vulnerabilities using tools like NPM outdated, NPM audit and Snyk.

3. Apply safety precautions.

5. Refer to the steps provided in key readings, Check library updates and security vulnerabilities.

4. Present out steps to check library updates and security vulnerabilities.

Key readings 2.3.2: Checking for Library Updates and Security Vulnerabilities

Once checking for libraly updates and security vulnerability you can use tools like npm outdated, npm audit and snyk.

1. **Checking for Outdated Packages**

   After some time, you might want to check if there are newer versions of your dependencies:

   ```
   npm outdated
   ```

   This command outputs something like:

   ```
   Package  Current  Wanted  Latest  Location
   express  4.17.3   4.17.4  5.0.0   node_modules/express
   lodash   4.17.21  4.17.21 4.17.21 node_modules/lodash
   moment   2.29.1   2.29.1  2.29.2  node_modules/moment
   ```

   Here, express has a new major version 5.0.0, but you might only want to update to 4.17.4 for now, as it's safer and backward compatible.

2. **Checking for Security Vulnerabilities:**

   To ensure that your dependencies are secure, run:

   ```
   npm audit
   ```

   This command might show:

   ```
   found 2 vulnerabilities (1 low, 1 moderate) in 1504 scanned packages
    2 vulnerabilities require manual review. See the full report for details.
   ```

   You can attempt to fix these vulnerabilities by running:

   ```
   npm audit fix
   ```

   If npm audit fix can't resolve everything, you'll need to investigate and manually update or replace the vulnerable packages.

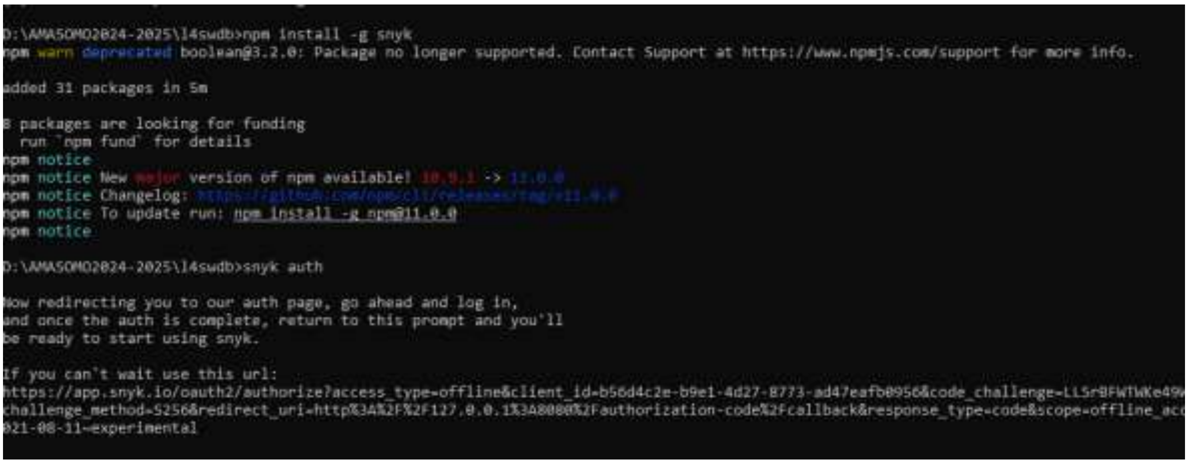
3. **Using Snyk**

   Snyk offers deeper security scanning. First, install it:

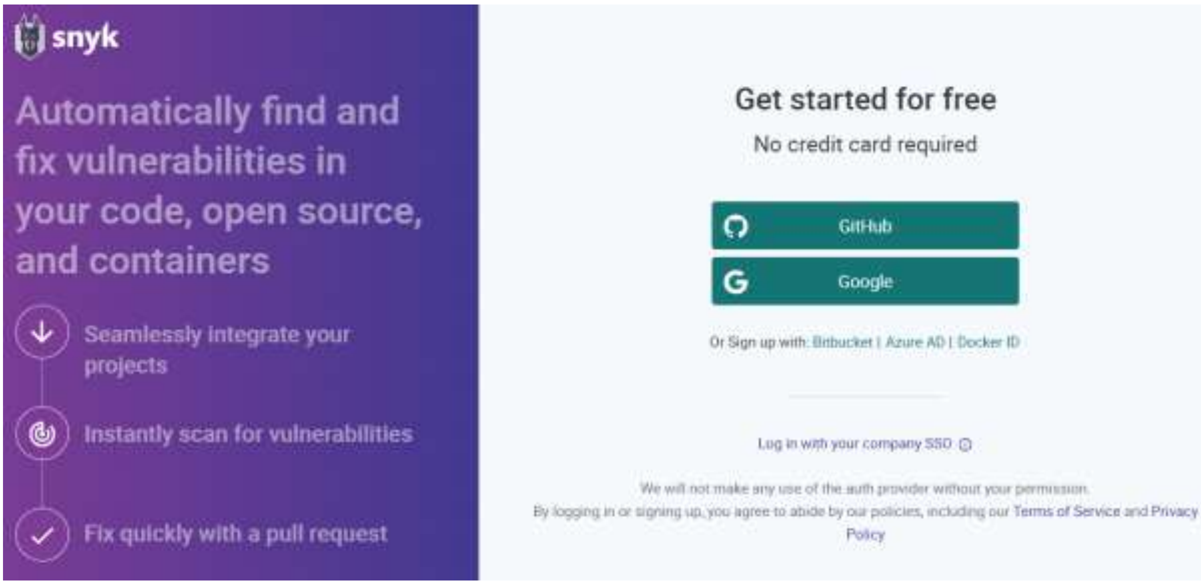
   ```
   npm install -g snyk
   ```

Authenticate (you'll need an account):

snyk auth



```
D:\AMASOMO2024-2025\l4swdb>npm install -g snyk
npm warn deprecated boolean@3.2.0: Package no longer supported. Contact Support at https://www.npmjs.com/support for more info.

added 31 packages in 5m

8 packages are looking for funding
  run `npm fund` for details
npm notice
npm notice New major version of npm available! 10.9.1 -> 11.0.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.0.0
npm notice To update run: npm install -g npm@11.0.0
npm notice

D:\AMASOMO2024-2025\l4swdb>snyk auth

Now redirecting you to our auth page, go ahead and log in,
and once the auth is complete, return to this prompt and you'll
be ready to start using snyk.

If you can't wait use this url:
https://app.snyk.io/oauth2/authorize?access_type=offline&client_id=b56d4c2e-b9e1-4d27-8773-ad47eafb0956&code_challenge=LLSrBFWTWKe49W
challenge_method=S256&redirect_uri=http%3A%2F%2F127.0.0.1%3A8080%2Fauthorization-code%2Fcallback&response_type=code&scope=offline_acc
021-08-11~experimental
```



Automatically find and fix vulnerabilities in your code, open source, and containers

Seamlessly integrate your projects

Instantly scan for vulnerabilities

Fix quickly with a pull request

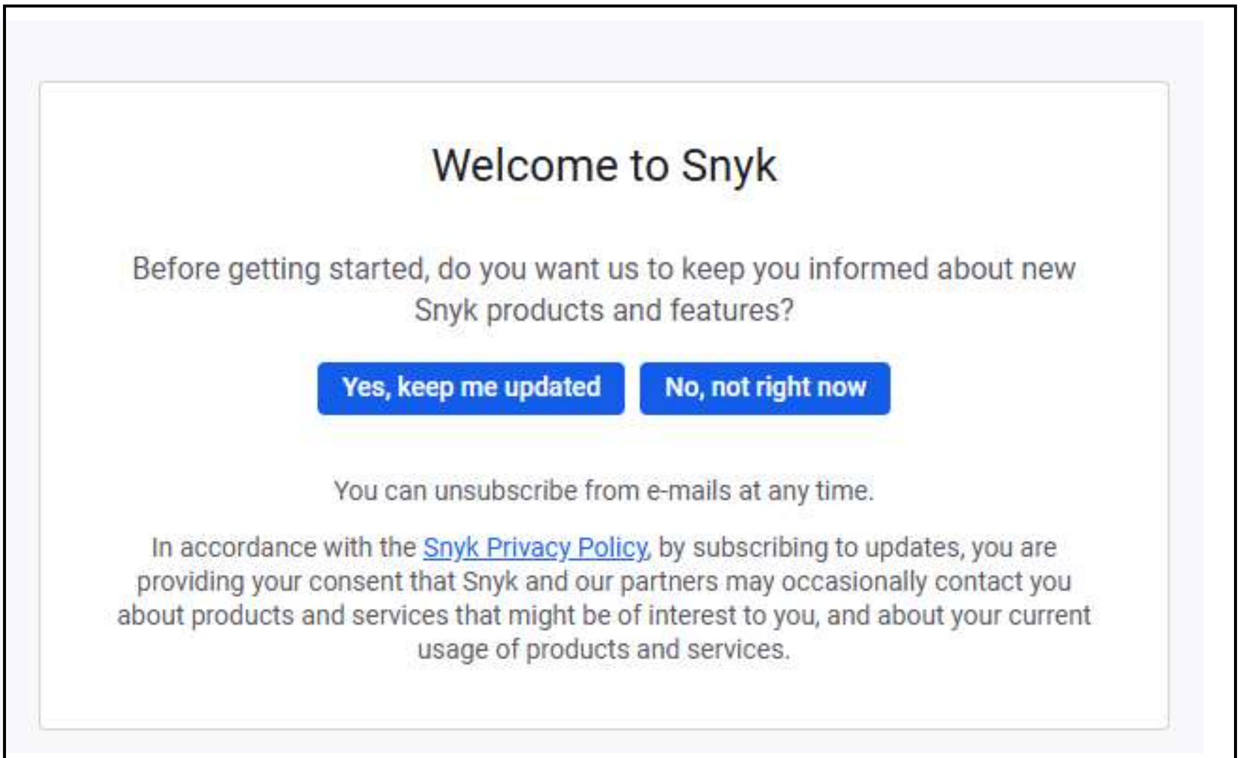
Get started for free

No credit card required

Or Sign up with: Bitbucket | Azure AD | Docker ID

Log in with your company SSO

We will not make any use of the auth provider without your permission.
By logging in or signing up, you agree to abide by our policies, including our Terms of Service and Privacy Policy

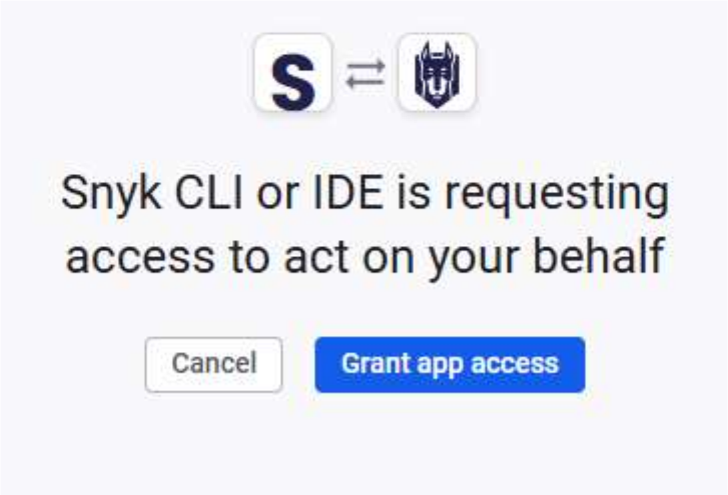


Welcome to Snyk

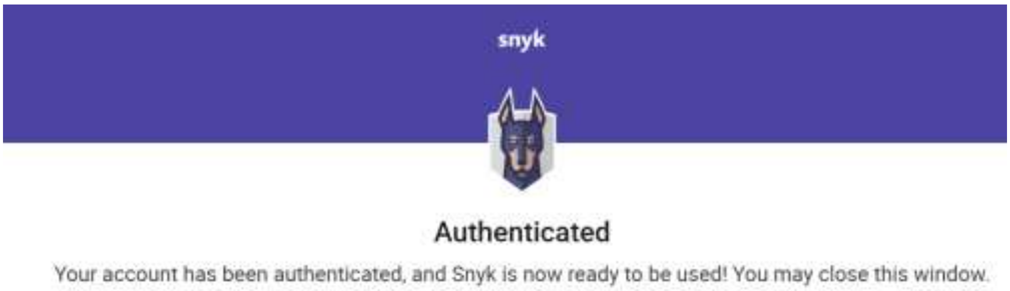
Before getting started, do you want us to keep you informed about new Snyk products and features?

You can unsubscribe from e-mails at any time.

In accordance with the Snyk Privacy Policy, by subscribing to updates, you are providing your consent that Snyk and our partners may occasionally contact you about products and services that might be of interest to you, and about your current usage of products and services.



Snyk CLI or IDE is requesting access to act on your behalf

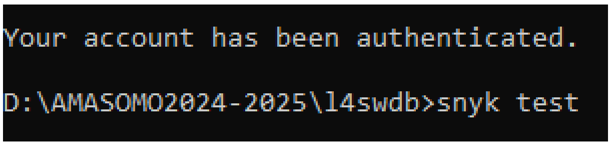


Authenticated

Your account has been authenticated, and Snyk is now ready to be used! You may close this window.

Then, run a security test:

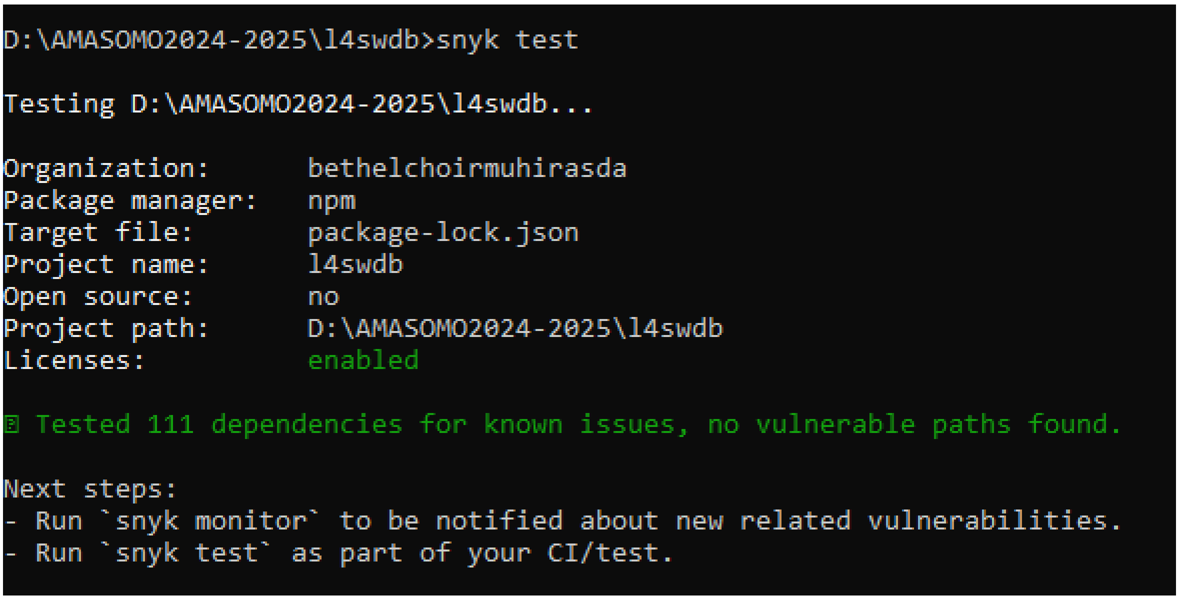
snyk test



```
Your account has been authenticated.

D:\AMASOMO2024-2025\l4swdb>snyk test
```

Snyk will provide detailed security reports and suggest fixes, even offering to open pull requests to fix vulnerabilities.

After checking all vurnerabilities it has to display the report, once any issues is observed it privides the way of making correction.



```
D:\AMASOMO2024-2025\l4swdb>snyk test

Testing D:\AMASOMO2024-2025\l4swdb...

Organization:      bethelchoirmuhirasda
Package manager:   npm
Target file:       package-lock.json
Project name:      l4swdb
Open source:       no
Project path:      D:\AMASOMO2024-2025\l4swdb
Licenses:          enabled

✓ Tested 111 dependencies for known issues, no vulnerable paths found.

Next steps:
- Run `snyk monitor` to be notified about new related vulnerabilities.
- Run `snyk test` as part of your CI/test.
```

Practical Activity 2.3.3: Updating third-party libraries

Task:

1. Read key reading 2.3.3.

2. Referring to the previous practical activities (2.3.2) you are requested to go to the computer lab to Update Third-Party Libraries.

3. Apply safety precautions

4. Refer to the steps provided in key readings, Update Third-Party Libraries

5. Present out the Steps in Update Third-Party Libraries.

Key readings 2.3.3: Updating Third-Party Libraries

When updating third-party libraries in Node.js, understanding versioning and the Semantic Versioning (semver) rules is crucial for maintaining the stability of your application.

1. **Versioning and Semver Rules**

Semantic Versioning (semver) is a convention used to indicate the types of changes in a version of a library or package. A semver version is typically expressed as MAJOR.MINOR. PATCH, like 1.2.3.

MAJOR version: Introduces incompatible API changes. Updating to a new major version may require code changes in your application.

Example: 1.0.0 to 2.0.0

MINOR version: Adds functionality in a backward-compatible manner. Updating to a new minor version should not break existing code.

Example: 1.2.0 to 1.3.0

PATCH version: Makes backward-compatible bug fixes. Updating to a new patch version is typically safe and should not introduce any breaking changes.

Example: 1.2.3 to 1.2.4

2. **Understanding Version Ranges in package. json**

In the package.json file, the version numbers of dependencies often use special characters to define ranges:

Caret (^): Allows updates that do not change the left-most non-zero digit (major version).

Example: "express": "^4.17.1" allows any 4.x.x version, such as 4.18.0 but not 5.0.0.

Tilde (~): Allows updates to the patch version but not the minor version.

Example: "express": "~4.17.1" allows updates to 4.17.x, such as 4.17.2, but not to 4.18.0.

Exact Version: Specifies an exact version, ensuring no updates.

Example: "express": "4.17.1" allows only version 4.17.1 and nothing else.

Range (>=, <=): Allows specifying a range of versions.

Example: "express": ">=4.16.0 <5.0.0" allows any version from 4.16.0 to the latest 4.x.x, but not 5.x.x.

3. **Versioning Guidelines When Updating Libraries**

   Minor and Patch Updates:

   If the update is within the same major version (e.g., from 4.17.1 to 4.18.0), it is generally safe to update because semver guarantees backward compatibility.

4. **To update within the minor or patch versions:**

   ```
   npm update
   ```

   Major Updates:

   Major updates (e.g., from 4.x.x to 5.x.x) often introduce breaking changes. It’s essential to read the release notes or changelog before upgrading.

   Major versions should be updated with caution:

   ```
   npm install express@latest
   ```

   After updating, thoroughly test your application to ensure compatibility.

   Locking Versions:

   For production environments where stability is crucial, you might prefer to lock the versions of dependencies to avoid unexpected changes.

   Use the --save-exact flag to install the exact version of a package:

   ```
   npm install express@4.17.1 --save-exact
   ```

   Use of package-lock.json:

   The package-lock.json file locks the exact versions of all dependencies (and their dependencies) used in your project. This ensures that everyone using your project installs the exact same versions of all packages, preventing discrepancies across different environments.

5. **Upgrading Guidelines**

   Regularly Review Updates: Regularly check for updates using npm outdated and decide whether to upgrade based on the impact.

   Automated Tools: Consider using tools like npm-check-updates to help manage and upgrade dependencies.

Example of Versioning and Semver Rules:

- **Understanding Semantic Versioning:**

Let's say your package. json initially has:

```
"dependencies": {
"express": "^4.17.3"
}
```

The caret (^) means that npm install will update to any newer minor or patch version but will not install 5.x.x versions, which could have breaking changes.

To update express to the latest compatible version:

```
npm update express
```

This might change your package. json to:

```
"dependencies": {
"express": "^4.17.4"
}
```

If you're confident in upgrading to the latest major version:

```
npm install express@latest
```

This updates express to 5.0.0, with potential breaking changes.

Theoretical Activity 2.3.4: Description of Strategies for managing and testing updates

Tasks:

1: You are requested to answer the following questions related to the Description of Strategies for managing and testing library:

i. How do you understand by the followings?

a) staging environments
b) Version control systems.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class and trainer.

4: For more clarification, read the key readings 2.3.4. In addition, ask questions where necessary.

Key readings 2.3.4.: Description of Strategies for managing and testing updates

When managing and testing library updates in Node.js applications, it is crucial to adopt strategies that ensure stability and reliability. Two key strategies include using staging environments and version control systems.

1. **Staging Environments**

A **staging environment** is a pre-production environment that replicates the production environment as closely as possible. It allows developers to test updates, including library updates, in a controlled setting before deploying them to the live application. By using a staging environment:

Risk Reduction: Potential issues caused by new library versions can be identified and fixed without impacting users.

Validation: New features and bug fixes introduced by the updates can be thoroughly tested, ensuring compatibility with the existing codebase.

2. **Version Control Systems**

Version control systems (VCS) like Git are essential for managing changes in a codebase, including library updates. They provide several benefits:

Branching and Merging: Developers can create separate branches for testing library updates, allowing them to isolate changes and avoid disrupting the main codebase.

Reversion: If a library update causes issues, VCS enables easy reversion to a previous stable state.

Collaboration: Multiple developers can work on different aspects of the update simultaneously, with the VCS managing changes and resolving conflicts.

Example with Staging Environments and Version Control:

- **Using a Staging Environment:**

  Suppose you have a staging environment set up where you can test updates before deploying to production. After updating a package, deploy your application to staging:

```
git push staging-branch
```

  Your CI/CD pipeline deploys this to a staging server. Here, you can run your full test suite and do manual testing to ensure nothing breaks.

- **Using Version Control Systems:**

  Imagine you've updated express in a new branch:

```
git checkout -b update-express
npm install express@latest
git add package.json package-lock.json
git commit -m "Updated express to latest version"
```

  Push the branch and create a pull request for review:

```
git push origin update-express
```

  After reviewing and testing, merge the pull request into your main branch:

```
git checkout main
git merge update-express
git push origin main
```

  If anything goes wrong, you can easily revert to a previous commit:

```
git revert <commit_hash>
```

Points to Remember

- Monitoring library dependencies in Node.js involves using package.json to define and npm-shrinkwrap.json to lock exact versions of dependencies.

- Tools like npm outdated and npm audit help identify outdated packages and security vulnerabilities, with Snyk providing deeper security checks.
- Safe updates are managed using semantic versioning (semver), ensuring minor updates don’t introduce breaking changes.
- Testing updates in staging environments and managing them through Git branches and pull requests ensure changes are stable before reaching production, minimizing the risk of issues.

Application of learning 2.3.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients. As a newly hired backend developer using Node.js, you are requested to Incorporate common Node.js third-party libraries like Express, Lodash and Moment.js and Interacting with these third-party libraries with Callbacks, Promises and async/await then Monitor library dependencies and version numbers in Package. Json and Npm-shrinkwrap. Json and also Check for library updates and security vulnerabilities using tools NPM outdated, NPM audit and Snyk and next Update third-party libraries by Versions and semver rules

Indicative content 2.4: Implementation of Authentication.

Duration: 3 hrs

Theoretical Activity 2.4.1: Description of authentication.

Tasks:

1: You are requested to answer the following questions related to the description of authentication:

i. What is Authentication?

ii. What are the principles of authentication?

iii. Explain the role of authentication in system security

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class and the trainer

4: For more clarification, read the key readings 2.4.1. In addition, ask questions where necessary.

Key readings 2.4.1: Description of authentication

1. Description of authentication

Authentication is a critical process in Node.js applications, ensuring that users are who they claim to be before granting access to sensitive resources. It plays a vital role in maintaining the security and integrity of a system.

2. Principles of Authentication

The **principles of authentication** revolve around validating the identity of a user through various methods:

- **Credentials Verification:** Users typically provide credentials, such as a username and password, which are then verified against stored records.
- **Multi-Factor Authentication (MFA):** Enhances security by requiring multiple forms of verification, such as a password and a one-time code sent to the user's phone.
- **Token-Based Authentication:** After successful authentication, the server generates a token (e.g., JWT) that the user must include in subsequent requests.

This token is verified to maintain the user session without repeatedly asking for credentials.

3. **Role of Authentication in System Security**

   Authentication is essential in system security for the following reasons:

- **Access Control:** It restricts access to resources, ensuring that only authorized users can perform certain actions or view sensitive data.
- **User Accountability:** By confirming the identity of users, the system can track actions taken by specific individuals, enhancing auditability and accountability.
- **Protection Against Unauthorized Access:** Proper authentication mechanisms prevent unauthorized users from accessing the system, thereby protecting data and resources from malicious actors.

Practical Activity 2.4.2: Implementing user authentication

Task:

1. Read key reading 2.4.2.

2. Referring to the previous theoretical activities (2.4.1) you are requested to go to the computer lab to Implement Authentication on developed project by using frameworks like Passport, JWT (JSON Web Tokens) and perform Social Authentication.

3. Apply safety precautions

4. Refer to the steps provided on task three, then Implement Authentication in an existing project.

5. Present out the Steps in Implement Authentication.

Key readings 2.4.2: Implementing user authentication

1. **Implementing Authentication with Passport.js**

   Passport.js is a popular authentication middleware for Node.js that supports various authentication strategies, including username/password, OAuth, and more.

Steps:

1. **Install Passport and Related Modules:**

```
npm install passport passport-local express-session
```

2. **Configure Passport:**

Create a passport.js configuration file to define the local strategy.

```
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
passport.use(new LocalStrategy(
function(username, password, done) {
// Here, you would check the username and password against your database
if (username === 'admin' && password === 'password') {
return done(null, { id: 1, username: 'admin' });
} else {
return done(null, false, { message: 'Incorrect credentials.' });
}
}
));

passport.serializeUser(function(user, done) {
done(null, user.id);
});

passport.deserializeUser(function(id, done) {
// Retrieve user information from the database
done(null, { id: 1, username: 'admin' });
});
```

3. **Integrate Passport into Your Application:**

```
const express = require('express');
const passport = require('passport');
const session = require('express-session');
const app = express();
app.use(session({ secret: 'secret', resave: false, saveUninitialized: false }));
app.use(passport.initialize());
app.use(passport.session());
app.post('/login', passport.authenticate('local', {
successRedirect: '/dashboard',
failureRedirect: '/login',
failureFlash: true
}));
app.get('/dashboard', (req, res) => {
if (req.isAuthenticated()) {
res.send('Welcome to your dashboard!');
} else {
res.redirect('/login');
}
});
```

4. **Run the Application:**
   - Start your Node.js server and test the login functionality by visiting the /login route.

2. Implementing Authentication with JWT (JSON Web Tokens)

JWT (JSON Web Tokens) is a compact, URL-safe means of representing claims to be transferred between two parties. JWT is commonly used for stateless authentication.

Steps:

1. **Install JWT Modules:**

```
npm install jsonwebtoken bcryptjs
```

2. **Create a JWT Authentication Middleware:**

```
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const users = []; // Simulating a user database
app.post('/register', async (req, res) => {
const { username, password } = req.body;
const hashedPassword = await bcrypt.hash(password, 10);
users.push({ username, password: hashedPassword });
res.send('User registered!');
});

app.post('/login', (req, res) => {
```

```
const { username, password } = req.body;
const user = users.find(u => u.username === username);
if (user && bcrypt.compareSync(password, user.password)) {
const token = jwt.sign({ id: user.username }, 'secretkey', { expiresIn: '1h' });
res.json({ token });
} else {
res.send('Invalid credentials');
}
});

const authenticateJWT = (req, res, next) => {
const token = req.header('Authorization').split(' ')[1];
if (token) {
jwt.verify(token, 'secretkey', (err, user) => {
if (err) return res.sendStatus(403);
req.user = user;
next();
});
} else {
res.sendStatus(401);
}
};

app.get('/dashboard', authenticateJWT, (req, res) => {
res.send('Welcome to your dashboard!');
});
```

3. Run the Application:

Register a user, then log in to receive a JWT token. Use this token to access protected routes like /dashboard.

3. Implementing Social Authentication

Social Authentication allows users to log in to your application using their existing social media accounts (e.g., Google, Facebook).

Steps:

4. Install Passport.js Social Auth Modules:

```
npm install passport-google-oauth20
```

1. **Configure Google OAuth Strategy:**

In your passport.js configuration file:

```
const GoogleStrategy = require('passport-google-oauth20').Strategy;
passport.use(new GoogleStrategy({
clientID: 'YOUR_GOOGLE_CLIENT_ID',
clientSecret: 'YOUR_GOOGLE_CLIENT_SECRET',
callbackURL: "/auth/google/callback"
},
function(accessToken, refreshToken, profile, done) {
// Check if the user exists in your database and return user data
return done(null, profile);
}));
```

2. Set Up Routes for Google Authentication:

```
app.get('/auth/google',
passport.authenticate('google', { scope: ['profile', 'email'] })
);
app.get('/auth/google/callback',
passport.authenticate('google', { failureRedirect: '/login' }),
(req, res) => {
res.redirect('/dashboard');
});
```

3. Run the Application:

When a user visits the /auth/google route, they will be redirected to Google to log in. Upon successful login, they will be redirected back to your application.

If you want to implement google and facebook at the same time you follow the following steps:

Install necessary packages

```
npm install express passport passport-google-oauth20 passport-facebook
```

express-session dotenv

Express: Web framework for Node.js.

Passport: Authentication middleware for Node.js.

passport-google-oauth20: Google OAuth 2.0 authentication strategy for Passport.

passport-facebook: Facebook OAuth 2.0 authentication strategy for Passport.

express session: Middleware to manage user sessions.

dotenv: To load environment variables from a .env file.

2. Configure OAuth Apps

You need to create OAuth applications for Google and Facebook.

Google OAuth Setup

Go to Google Developer Console.

Create a new project.

Navigate to "OAuth consent screen" and configure it.

Go to "Credentials" and create OAuth 2.0 Client IDs.

Set the authorized redirect URIs (e.g., http://localhost:3000/auth/google/callback).

Note the Client ID and Client Secret.

Facebook OAuth Setup

Go to Facebook Developers.

Create a new app.

Go to "Add a Product" and select "Facebook Login".

Set the "Valid OAuth Redirect URIs" (e.g., http://localhost:3000/auth/facebook/callback).

Note the App ID and App Secret.

3. Create the server.js File

Create a server.js file that sets up Express, Passport, and the routes for Google and Facebook authentication.

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

const passport = require('passport');

const session = require('express-session');

const GoogleStrategy = require('passport-google-oauth20').Strategy;

const FacebookStrategy = require('passport-facebook').Strategy;

require('dotenv').config();

const app = express();

// Session Setup

app.use(session({ secret: 'secret', resave: false, saveUninitialized: true }));

// Passport Setup

app.use(passport.initialize());

app.use(passport.session());

// Serialize and Deserialize User

passport.serializeUser((user, done) => {

done(null, user);

});

passport.deserializeUser((user, done) => {
```

```
done(null, user);

});

// Google Strategy

passport.use(new GoogleStrategy({

clientID: process.env.GOOGLE_CLIENT_ID,

clientSecret: process.env.GOOGLE_CLIENT_SECRET,

callbackURL: '/auth/google/callback'

},

(accessToken, refreshToken, profile, done) => {

return done(null, profile);

}

));

// Facebook Strategy

passport.use(new FacebookStrategy({

clientID: process.env.FACEBOOK_APP_ID,

clientSecret: process.env.FACEBOOK_APP_SECRET,

callbackURL: '/auth/facebook/callback'

},

(accessToken, refreshToken, profile, done) => {

return done(null, profile);

}

));
```

```
// Routes
app.get('/', (req, res) => {
res.send(`<h1>Welcome</h1><a href="/auth/google">Login with Google</a><br><a href="/auth/facebook">Login with Facebook</a>`);
});
// Google Authentication Route
app.get('/auth/google',
passport.authenticate('google', { scope: ['profile', 'email'] })
);
app.get('/auth/google/callback',
passport.authenticate('google', { failureRedirect: '/' }),
 (req, res) => {
res.redirect('/profile');
}
);
// Facebook Authentication Route
app.get('/auth/facebook',
passport.authenticate('facebook')
);
app.get('/auth/facebook/callback',
passport.authenticate('facebook', { failureRedirect: '/' }),
(req, res) => {
```

```
res.redirect('/profile');
}
);
// Profile Route
app.get('/profile', (req, res) => {
res.send(`<h1>Profile</h1><p>${JSON.stringify(req.user, null, 4)}</p>`);
});
// Logout Route
app.get('/logout', (req, res) => {
req.logout(() => {
res.redirect('/');
});
});
// Start Server
app.listen(3000, () => {
console.log('Server started on http://localhost:3000');
});
```

4. Create the .env File

Create a .env file in the root directory of your project and add your OAuth credentials:

Make file

```
GOOGLE_CLIENT_ID=your-google-client-id
```

```
GOOGLE_CLIENT_SECRET=your-google-client-secret

FACEBOOK_APP_ID=your-facebook-app-id

FACEBOOK_APP_SECRET=your-facebook-app-secret
```

5. Run the Application

Start your Node.js application:

```
node server.js
```

Visit http://localhost:3000 in your browser, and you should see options to log in with Google or Facebook.

After successfully logging in, you'll be redirected to the /profile route, where you can see the user's profile information.

Important Notes

Security: Always ensure that the callbackURL in your OAuth setup matches the one in your OAuth provider settings.

HTTPS: OAuth providers often require secure (HTTPS) redirect URIs, especially in production.

Session Management: In a production environment, consider using a more robust session store, such as Redis, instead of the default memory store.

Practical Activity 2.4.3: Using authentication middleware

Task:

1. Read key reading 2.4.3.

2. Referring to the previous practical activities (2.4.2) you are requested to go to the computer lab to Use authentication middleware to protect routes and resources.

3. Apply safety precautions

4. Refer to the steps provided in key readings, then use authentication middleware to protect routes and resources.

5. Present out the Steps in using authentication middleware to protect routes and resources.

Key readings 2.4.3: Using authentication middleware

Use authentication middleware to protect routes and resources

This ensures that only authenticated users can access certain parts of the application.

1. Setting Up the Project

Before diving into middleware, ensure you have a basic Node.js application set up with authentication (e.g., using Passport.js, JWT, etc.).

1. **Install Necessary Modules:** If not already installed, install Express and any authentication-related packages you need:

```
npm install express passport jsonwebtoken
```

2. **Set Up a Basic Express Server:**

```
const express = require('express');
const app = express();
app.use(express.json());
// Your authentication setup (e.g., Passport.js, JWT) goes here
```

2. Creating Authentication Middleware

Authentication middleware is a function that runs before a request reaches a protected route, verifying whether the user is authenticated.

Example with JWT:

1. **Create the JWT Authentication Middleware:**

```
const jwt = require('jsonwebtoken');
const authenticateJWT = (req, res, next) => {
const token = req.header('Authorization')?.split(' ')[1];
if (token) {
jwt.verify(token, 'secretkey', (err, user) => {
if (err) return res.sendStatus(403);
req.user = user;
next();
});
} else {
res.sendStatus(401); // Unauthorized
}
};
```

2. **Using the Middleware to Protect Routes:**

Apply the authenticateJWT middleware to routes that should be protected:

```
app.get('/dashboard', authenticateJWT, (req, res) => {
res.send('Welcome to the protected dashboard!');
});
app.get('/profile', authenticateJWT, (req, res) => {
res.json({ user: req.user, message: 'This is your protected profile page.' });
});
```

Public routes, like login or registration, do not need this middleware:

```
app.post('/login', (req, res) => {
// Login logic and token generation
const token = jwt.sign({ username: req.body.username }, 'secretkey',
{ expiresIn: '1h' });
res.json({ token });
});
```

3. Testing Protected Routes

1. **Start the Application:**

Run your Node.js server:

```
node app.js
```

2. **Access Protected Routes:**

Use a tool like Postman or curl to send requests:

Without a token: Attempt to access /dashboard or /profile. You should receive a 401 Unauthorized status.

With a token: Send a valid JWT token in the Authorization header and access the protected routes.

Example of sending a request with a token using curl:

```
Curl -H "Authorization: Bearer <your_token_here>"
http://localhost:3000/dashbsoard
```

4. Enhancing the Middleware (Optional)

You can customize your authentication middleware to include additional checks, such as role-based access control (RBAC), where only users with specific roles (e.g., admin) can access certain routes.

Example of Role-Based Access Control:

```
const authorizeRoles = (roles) => {
return (req, res, next) => {
if (roles.includes(req.user.role)) {
next();
} else {
res.sendStatus(403); // Forbidden
}
};
};

app.get('/admin', [authenticateJWT, authorizeRoles(['admin'])], (req, res) => {
res.send('Welcome to the admin panel!');
});
```

Theoretical Activity 2.4.4: Description of best practices for password storage and handling sensitive data

Tasks:

1: You are requested to answer the following questions related to the Best practices for password storage and handling sensitive:

I. Describe the Best Practices for Password Storage
II. Describe the Best Practices for Handling Sensitive Data

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.4.4. In addition, ask questions where necessary.

Key readings 2.4.4: Description of best practices for password storage and handling sensitive data

- **Best practices for password storage and handling sensitive data**.

  Handling passwords and sensitive data securely is crucial to protecting user privacy and maintaining the integrity of your application. This activity outlines the best practices for storing passwords and managing sensitive data effectively.

  1. Best Practices for Password Storage

a. Use Strong Hashing Algorithms:

- Passwords should never be stored in plaintext. Instead, use strong, one-way hashing algorithms such as **bcrypt**, **Argon2**, or **scrypt**.
- These algorithms are designed to be computationally intensive, which slows down brute force attacks.

b. Salt Passwords Before Hashing:

- Always add a unique, random salt to each password before hashing. A salt is a random string that ensures even identical passwords produce different hashes.
- Salting prevents attackers from using precomputed hash tables (rainbow tables) to crack passwords.

c. Apply Password Stretching:

- Password stretching involves hashing the password multiple times to make the process more computationally expensive for attackers.
- bcrypt and Argon2 inherently support password stretching, enhancing security.

d. Avoid Reusing Hashing Algorithms:

- Regularly review and update your password hashing practices to ensure you're using current and secure algorithms. When updating, re-hash passwords during user login or password changes.

e. Never Store Plaintext Passwords:

- Plaintext passwords are highly vulnerable to theft and misuse. Always store hashed and salted passwords to mitigate risks in case of a database breach.

2. Best Practices for Handling Sensitive Data

a. Encrypt Sensitive Data:

- Use strong encryption methods, such as AES-256, to protect sensitive data like personal information, financial details, and authentication tokens.
- Encrypt data both at rest (when stored) and in transit (when transmitted over networks).

b. Implement Access Control:

- Restrict access to sensitive data based on the principle of least privilege—only those who need access should have it.
- Use role-based access control (RBAC) to manage permissions and enforce data security policies.

c. Secure Data Transmission:

- Ensure all sensitive data is transmitted over secure channels using protocols like TLS (Transport Layer Security) to prevent interception by unauthorized parties.
- Regularly update SSL/TLS certificates to maintain secure communication.

d. Regularly Audit and Monitor Access:

- Conduct regular audits of who accesses sensitive data and how it's handled within the system.
- Implement logging and monitoring to detect and respond to unauthorized access or suspicious activities in real-time.

e. Educate and Train Users:

- Provide guidance to users on creating strong passwords, avoiding reuse, and recognizing phishing attempts.

- Train developers on secure coding practices, particularly in handling passwords and sensitive data.

f. Implement Secure Data Disposal:

- Ensure that sensitive data is securely erased when no longer needed. This includes securely wiping disks or using cryptographic methods to render the data unreadable.

2. Handling Sensitive Data

Encryption:

Data Encryption: Encrypt sensitive data both at rest and in transit using strong encryption algorithms (e.g., AES-256).

Transport Layer Security (TLS): Use TLS to encrypt data transmitted between clients and servers.

Access Controls:

Role-Based Access Control (RBAC): Implement RBAC to restrict access to sensitive data based on user roles.

Least Privilege Principle: Limit access to sensitive data to only those who need it to perform their job functions.

Environment Variables:

Environment Variables: Store sensitive configuration values such as database credentials and API keys in environment variables, not in source code.

Secrets Management: Use a secrets management tool (e.g., AWS Secrets Manager, HashiCorp Vault) to manage sensitive information securely.

Data Masking and Anonymization:

Masking: Mask sensitive data in logs and error messages to prevent exposure.

Anonymization: Anonymize personal data where possible to protect user privacy.

Regular Security Audits:

Security Audits: Conduct regular security audits and vulnerability assessments to identify and address potential security risks.

Penetration Testing: Perform penetration testing to simulate attacks and evaluate the security of your system.

Monitoring and Logging:

Monitoring: Monitor access to sensitive data and set up alerts for suspicious activities.

Logging: Log access to sensitive data and ensure logs are protected and reviewed regularly.

User Education:

Training: Educate users on the importance of strong passwords and security practices.

Two-Factor Authentication (2FA): Encourage or mandate the use of 2FA to add an extra layer of security.

3. Compliance with Regulations

Regulatory Compliance:

Ensure compliance with data protection regulations such as GDPR, CCPA, HIPAA, and others relevant to your industry and location.

Implement measures to protect user data and respond to data breaches as required by law.

Points to Remember

- Authentication is a critical process in Node.js applications, ensuring that users are wh o they claim to be before granting access to sensitive resources.
- The principles of authentication are:

    1. Credentials Verification.

2. Multi-Factor Authentication (MFA).
3. Token-Based Authentication.

- Role of Authentication in System Security
    1. Access Control
    2. User Accountability
    3. Protection Against Unauthorized Access
- Implementing user authentication you can follow those steps:
    1. Install Passport and Related Modules
    2. Configure Passport
    3. Integrate Passport into Your Application
    4. Run the Application
- Implementing Authentication with JWT (JSON Web Tokens)
    1. Install JWT Modules
    2. Create a JWT Authentication Middleware
    3. Run the Application
- Implementing Social Authentication
    1. Install Passport.js Social Auth Modules
    2. Configure Google OAuth Strategy
    3. Set Up Routes for Google Authentication
    4. Run the Application
- Use authentication middleware to protect routes and resources
    1. Setting Up the Project
    2. Creating Authentication Middleware
    3. Testing Protected Routes
    4. Enhancing the Middleware
- Best Practices for Password Storage
    1. Use Strong Hashing Algorithms
    2. Salt Passwords Before Hashing

    3. Apply Password Stretching
    4. Role of Authentication in System Security
    5. Avoid Reusing Hashing Algorithms
    6. Never Store Plaintext Passwords

- Best Practices for Handling Sensitive Data
    1. Encrypt Sensitive Data
    2. Implement Access Control
    3. Secure Data Transmission
    4. Regularly Audit and Monitor Access
    5. Educate and Train Users

Application of learning 2.4.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients. As a newly hired backend developer using Node.js, you are requested to implement user authentication using frameworks**,** Passport**,** JWT (JSON Web Tokens), and Social Auth. (Google, Facebook, …) and then you will use authentication middleware to protect routes and resources.

Indicative content 2.5: Implementation of Authorization

Duration: 3 hrs

Theoretical Activity 2.5.1: Description of authorization.

Tasks:

1. You are requested to answer the following questions related to the description of authorization:

   i. What is Authorization?
   ii. What are the principles of authorization?
   iii. Explain the role of authorization in system security

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.5.1. In addition, ask questions where necessary.

Key readings 2.5.1: Description of authorization.

Authorization is the process of granting or denying access to resources based on the identity of the user and their assigned permissions. It ensures that users only have access to the resources they are permitted to use.

2. Principles of Authorization

1. Least Privilege

Users and processes should only be granted the minimum level of access necessary to perform their tasks.

Implementation: Assign the least amount of privilege necessary to users and processes. For example, a user who only needs to view data should not have permission to modify or delete that data.

Benefit: Reduces the risk of accidental or intentional misuse of resources, limiting potential damage if an account or process is compromised.

Separation of Duties

No single individual or process should have control over all aspects of a critical task. Tasks should be divided among multiple users or processes to prevent fraud or errors.

Implementation: For example, in a financial system, the person who approves expenses should not be the same person who processes payments.

Benefit: Helps prevent fraud and errors by ensuring that critical actions require multiple approvals or actions from different parties.

Role-Based Access Control (RBAC)

Access permissions are assigned based on roles rather than individual users. Users are assigned roles, and roles have specific permissions.

Implementation: Define roles based on job functions (e.g., admin, editor, viewer), and assign permissions to these roles. Users are then assigned to the appropriate roles.

Benefit: Simplifies management of permissions, making it easier to apply and audit access controls as users change roles within an organization.

Attribute-Based Access Control (ABAC)

Access is granted based on attributes of the user, the resource, and the environment, rather than just roles. Attributes can include user department, resource type, time of access, etc.

Implementation: Define policies that use a combination of attributes to determine whether access should be granted.

For example, access could be granted if a user is in the HR department and it is within business hours.

Benefit: Provides more granular control over access, allowing for more dynamic and context-aware authorization decisions.

Contextual and Risk-Based Access Control

Access decisions are made based on the context of the request and associated risk factors. This may include the location of the request, the device being used, or the behavior of the user.

Implementation: Implement rules that take context into account, such as denying access from unknown IP addresses or requiring additional verification for high-risk actions.

Benefit: Enhances security by adjusting access control based on the perceived risk, reducing the likelihood of unauthorized access.

Audit and Accountability

All access and authorization decisions should be logged to allow for auditing and accountability. Users should be held accountable for their actions.

Implementation: Implement logging mechanisms that record who accessed what resources, when, and what actions were performed. Ensure that logs are monitored and reviewed regularly.

Benefit: Enables detection and investigation of unauthorized access, ensuring that any misuse of privileges can be traced back to the responsible party.

Principle of Transparency

Users should be aware of what they can and cannot do based on their permissions, and organizations should be clear about how access control decisions are made.

Implementation: Provide users with clear information about their permissions and the policies that govern access control. Ensure that authorization policies are documented and accessible.

Benefit: Reduces user confusion and helps ensure that users understand their responsibilities, minimizing accidental violations of policy.

Dynamic Authorization

Authorization decisions should be dynamic and adaptable to changing conditions, such as changes in user roles, resource sensitivity, or threat levels.

Implementation: Use authorization systems that can adapt to real-time information and make decisions based on the latest data.

Benefit: Allows for flexible and responsive access control, ensuring that permissions are always aligned with current risk levels and organizational policies.

End-to-End Security

Authorization controls should be enforced at every layer of the system, from the user interface to the backend database, to ensure that access is controlled throughout the entire data flow.

Implementation: Implement access controls at the application, database, and network layers. Ensure that sensitive operations require authorization at every step.

Benefit: Provides comprehensive protection against unauthorized access, even if one layer of defense is compromised.

Continuous Review and Improvement

Authorization policies and controls should be regularly reviewed and updated to reflect changes in the organization, technology, and threat landscape.

Implementation: Conduct regular audits and reviews of access control policies, and update them as needed. Ensure that authorization mechanisms are tested for effectiveness.

Benefit: Keeps the authorization system effective and relevant, ensuring that it continues to protect against evolving threats and meets organizational needs.

3. **Role of Authorization in System Security:**

Protection of Sensitive Data: Authorization helps protect sensitive data by ensuring that only authorized users can access it.

Compliance with Regulations: Many regulations require strict control over who can access certain types of data, making proper authorization essential.

Prevention of Unauthorized Actions: By controlling access, authorization prevents unauthorized users from performing actions that could harm the system or compromise data integrity.

Access Control Authorization is the primary mechanism for controlling access to resources in a system. It dictates what users can see, modify, or execute based on their roles and permissions.

Impact: By enforcing access control, authorization ensures that only authorized users can access sensitive data or perform critical actions, reducing the risk of unauthorized access or data breaches.

Data Protection

Function: Authorization ensures that sensitive data is only accessible to users with the necessary permissions. This includes protecting data from unauthorized viewing, modification, or deletion.

Impact: Protects the confidentiality and integrity of data, ensuring that it is only accessed by those who have legitimate reasons, thereby preventing data leaks or tampering.

Enforcing Organizational Policies

Function: Authorization helps enforce organizational policies by ensuring that access to resources is granted based on predefined rules and roles. It aligns user access with business requirements and security policies.

Impact: Helps maintain compliance with internal policies, industry regulations, and legal requirements, reducing the risk of violations that could lead to fines or legal issues.

Mitigating Insider Threats

Function: By restricting access to sensitive resources based on roles and responsibilities, authorization limits the potential for insider threats. Even if an insider has access to the system, their actions are constrained by their level of authorization.

Impact: Reduces the risk of malicious or accidental misuse of data by insiders, which is often one of the most challenging security threats to mitigate.

Audit and Accountability

Function: Authorization systems often include logging and monitoring features that track who accessed what resources and when. This creates an audit trail that can be used to detect and investigate security incidents.

Impact: Enhances accountability by ensuring that actions within the system can be traced back to specific users, making it easier to identify the source of security breaches and take corrective action.

Supporting the Principle of Least Privilege

Function: Authorization enforces the principle of least privilege by granting users the minimum level of access necessary to perform their tasks. This minimizes the risk of accidental or intentional misuse of resources.

Impact: Limits the potential attack surface within the system, reducing the chances that vulnerabilities can be exploited or that unauthorized actions can be performed.

Enhancing Multi-Tier Security

Function: In systems with multiple layers (e.g., application, database, network), authorization ensures that security controls are enforced at each layer, providing comprehensive protection.

Impact: Provides defense-in-depth, making it more difficult for attackers to compromise the system by requiring them to bypass multiple levels of authorization.

Dynamic and Context-Aware Security

Function: Modern authorization mechanisms can adapt to context, such as the location of the user, the device being used, or the time of access. This allows for more flexible and dynamic security controls.

Impact: Enhances security by allowing access decisions to be made based on real-time context, reducing the risk of unauthorized access under unusual circumstances.

Integrating with Identity Management

Function: Authorization works in conjunction with identity management systems to verify the identity of users and ensure they are granted appropriate access based on their roles.

Impact: Provides a seamless and secure experience for users, while ensuring that only authenticated and authorized users can access sensitive resources.

Enforcing Segregation of Duties

Function: Authorization helps enforce segregation of duties by ensuring that critical tasks are divided among multiple users or roles, preventing any single user from having too much control.

Impact: Reduces the risk of fraud or errors by ensuring that critical actions require multiple approvals or involve multiple individuals.

Practical Activity 2.5.2 Implementation of Role-Based and Attribute-Based Access Control

Task:

1. Read key reading 2.5.2.

2. Referring to the previous theoretical activities (2.5.1) you are requested to go to the computer lab to implement Role-Based and Attribute-Based Access Control on developed project on activity 2.4.3.

3. Apply safety precautions

4. Refer to the steps provided on task 4, then Implement Authentication in an existing project.

5. Present out the Steps in Implement Authentication.

Key readings 2.5.2: Implementation of Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC)

To implement Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC), you'll follow a series of steps to define roles, attributes, permissions, and the logic for access control decisions. Below are the steps with examples in Node.js using Express and Sequelize.

1. Set Up the Project

Start by setting up a basic Node.js project with Express and Sequelize.

```
mkdir access-control-example
cd access-control-example
npm init -y
npm install express sequelize mysql2 jsonwebtoken
```

2. Configure Sequelize

Create a **dbconfig.js** file to set up Sequelize with MySQL:

```
const { Sequelize } = require('sequelize');
const sequelize = new Sequelize('accessControlDB', 'root', 'password', {
host: 'localhost',
dialect: 'mysql',
});
module.exports = sequelize;
```

3. Define Models

Role Model (RBAC)

Create a **roleModel.js** file to define the Role model:

```
const { DataTypes } = require('sequelize');
const sequelize = require('../dbconfig');
const Role = sequelize.define('Role', {
name: {
type: DataTypes.STRING,
allowNull: false,
unique: true,
},
});
module.exports = Role;
```

User Model with Roles

Create a userModel.js file to define the User model:

```
const { DataTypes } = require('sequelize');
const sequelize = require('../dbconfig');
const Role = require('./roleModel');
const User = sequelize.define('User', {
username: {
type: DataTypes.STRING,
allowNull: false,
unique: true,
},
password: {
type: DataTypes.STRING,
allowNull: false,
},
});
// Define Role association
User.belongsTo(Role);
Role.hasMany(User);
module.exports = User;
```

Attribute-Based Model (ABAC)

Create an attributeModel.js file to define an Attribute model (e.g., department):

```
const { DataTypes } = require('sequelize');
const sequelize = require('../dbconfig');
const Attribute = sequelize.define('Attribute', {
key: {
type: DataTypes.STRING,
allowNull: false,
},
value: {
type: DataTypes.STRING,
allowNull: false,
},
});
module.exports = Attribute;
```

4. Create Middleware for Access Control

RBAC Middleware

Create a rbacMiddleware.js file to handle role-based access control:

```
const User = require('../models/userModel');
async function checkRole(roleName) {
return async (req, res, next) => {
const userId = req.user.id;
const user = await User.findByPk(userId, {
include: 'Role',
});
if (user && user.Role.name === roleName) {
return next();
} else {
return res.status(403).json({ message: 'Access denied' });
}
};
}
module.exports = checkRole;
```

ABAC Middleware

Create an abacMiddleware.js file to handle attribute-based access control:

```
const User = require('../models/userModel');
const Attribute = require('../models/attributeModel');
async function checkAttribute(key, value) {
  return async (req, res, next) => {
    const userId = req.user.id;
  const user = await User.findByPk(userId);
const attribute = await Attribute.findOne({
where: { key, value },
include: 'User',
});
if (attribute && attribute.UserId === user.id) {
return next();
} else {
return res.status(403).json({ message: 'Access denied' });
}
};
}
module.exports = checkAttribute;
```

5. Create Routes

RBAC Route

In your routes.js file, create a protected route that only allows access to a specific role:

```
const express = require('express');
const checkRole = require('../middleware/rbacMiddleware');
const router = express.Router();
router.get('/admin-dashboard', checkRole('admin'), (req, res) => {
 res.json({ message: 'Welcome to the admin dashboard' });
});
module.exports = router;
```

ABAC Route

In the same routes.js file, create a protected route that only allows access based on a specific attribute:

```
const checkAttribute = require('../middleware/abacMiddleware');
router.get('/department-dashboard', checkAttribute('department', 'IT'), (req, res)
=> {
res.json({ message: 'Welcome to the IT department dashboard' });
});
module.exports = router;
```

6. Testing the Implementation

In your server.js, set up the server and include the routes:

```
const express = require('express');
const app = express();
const routes = require('./routes/routes');
app.use(express.json());
// Use the routes
```

```
app.use('/api', routes);
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

7. Database Synchronization

Sync the models with the database:

```
const sequelize = require('./dbconfig');
const Role = require('./models/roleModel');
const User = require('./models/userModel');
const Attribute = require('./models/attributeModel');
sequelize.sync({ force: true }).then(() => {
  console.log('Database & tables created!');
});
```

8. Example User Creation

Manually create users and roles for testing:

```
const User = require('./models/userModel');
const Role = require('./models/roleModel');
const Attribute = require('./models/attributeModel');
async function createTestUsers() {
const adminRole = await Role.create({ name: 'admin' });
const userRole = await Role.create({ name: 'user' });
const admin = await User.create({ username: 'admin', password: 'adminpass', RoleId: adminRole.id });
const user = await User.create({ username: 'user', password: 'userpass', RoleId: userRole.id });
await Attribute.create({ key: 'department', value: 'IT', UserId: user.id });
console.log('Test users created');
}
createTestUsers();
```

9. Testing the Endpoints

Test the /admin-dashboard endpoint by logging in as an admin.

Test the /department-dashboard endpoint by logging in as a user with the IT department attribute.

Practical Activity 2.5.3: Using Authorization Middleware

Task:

1. Read key reading 2.5.3.

2. Referring to the previous practical activities (2.5.2) you are requested to go to the computer lab to Use Authorization Middleware to Manage User Permissions.

3. Apply safety precautions

5. Refer to the steps provided in key readings, then Use Authorization Middleware to Manage User Permissions.

4. Present out the Steps in Using Authorization Middleware to Manage User Permissions.

Key readings 2.5.3: Description of authorization.

- **Authorization Middleware**

  To use authorization middleware to manage user permissions, you can follow these steps, building on the previous implementation of Role-Based Access Control (RBAC) and Attribute-Based Access Control (ABAC). The goal is to restrict access to specific routes based on user roles and attributes.

  1. Set Up Authorization Middleware

  You need to create middleware functions that will check user permissions based on their roles (RBAC) and attributes (ABAC).

  RBAC Middleware

  Create a middleware function that checks if a user has the required role:

```
// middleware/rbacMiddleware.js
const User = require('../models/userModel');

function checkRole(requiredRole) {
return async (req, res, next) => {
const userId = req.user.id;
const user = await User.findByPk(userId, {
include: 'Role',
});

if (user && user.Role.name === requiredRole) {
return next(); // User has the required role
} else {
return res.status(403).json({ message: 'Access denied. Insufficient role.' });
}
};
}
module.exports = checkRole;
```

ABAC Middleware

Create a middleware function that checks if a user has the required attribute:

```
// middleware/abacMiddleware.js
const User = require('../models/userModel');
const Attribute = require('../models/attributeModel');

function checkAttribute(key, value) {
return async (req, res, next) => {
const userId = req.user.id;
const attribute = await Attribute.findOne({
where: { key, value, UserId: userId },
});

if (attribute) {
return next(); // User has the required attribute
} else {
return res.status(403).json({ message: 'Access denied. Insufficient attributes.' });
}
};
}
module.exports = checkAttribute;
```

2. Integrate Middleware in Routes

Use the middleware in your route definitions to protect certain routes based on user roles or attributes.

RBAC Example

Create a route that only users with the "admin" role can access:

```
// routes/adminRoutes.js
const express = require('express');
const checkRole = require('../middleware/rbacMiddleware');
const router = express.Router();

router.get('/admin-dashboard', checkRole('admin'), (req, res) => {
res.json({ message: 'Welcome to the admin dashboard' });
});

module.exports = router;
```

ABAC Example

Create a route that only users with a specific attribute (e.g., department "IT") can access:

```
// routes/itRoutes.js
```

```
const express = require('express');
const checkAttribute = require('../middleware/abacMiddleware');
const router = express.Router();

router.get('/it-dashboard', checkAttribute('department', 'IT'), (req, res) => {
res.json({ message: 'Welcome to the IT department dashboard' });
});

module.exports = router;
```

3. Set Up User Authentication

Ensure that your users are authenticated before they can be authorized. You can use JSON Web Tokens (JWT) to handle authentication.

JWT Authentication Middleware

Create a middleware function that verifies the JWT and attaches the user to the request object:

```
// middleware/authMiddleware.js
const jwt = require('jsonwebtoken');

function authenticateToken(req, res, next) {
const token = req.header('Authorization') && req.header('Authorization').split('
')[1];

if (!token) {
return res.status(401).json({ message: 'Unauthorized. No token provided.' });
}

jwt.verify(token, 'your_secret_key', (err, user) => {
if (err) {
return res.status(403).json({ message: 'Forbidden. Invalid token.' });
}
req.user = user;
next();
});
}
module.exports = authenticateToken;
```

4. Apply Middleware to Routes

Integrate authentication and authorization middleware in your Express routes.

Example: Applying Both RBAC and ABAC

```
// routes/protectedRoutes.js
const express = require('express');
const authenticateToken = require('../middleware/authMiddleware');
const checkRole = require('../middleware/rbacMiddleware');
const checkAttribute = require('../middleware/abacMiddleware');
const router = express.Router();
router.use(authenticateToken); // Apply JWT authentication to all routes
router.get('/admin-dashboard', checkRole('admin'), (req, res) => {
res.json({ message: 'Admin content here' });
});
router.get('/it-dashboard', checkAttribute('department', 'IT'), (req, res) => {
res.json({ message: 'IT department content here' });
});
module.exports = router;
```

5. Start the Server and Test

Ensure your Express app uses the routes:

```
// server.js
const express = require('express');
const adminRoutes = require('./routes/adminRoutes');
const itRoutes = require('./routes/itRoutes');
const protectedRoutes = require('./routes/protectedRoutes');
const app = express();

app.use(express.json());

// Unprotected routes
app.use('/api/admin', adminRoutes);
app.use('/api/it', itRoutes);

// Protected routes (with JWT auth, RBAC, and ABAC)
app.use('/api/protected', protectedRoutes);

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
console.log(`Server running on port ${PORT}`);
});
```

6. Example JWT Token Generation

To test your setup, you need a way to generate JWT tokens. Here's a basic example of how you might generate a token after a user logs in:

```
// authController.js (Example)
const jwt = require('jsonwebtoken');
const User = require('../models/userModel');
```

```
async function login(req, res) {
const { username, password } = req.body;
const user = await User.findOne({ where: { username } });
if (!user || user.password !== password) {
return res.status(401).json({ message: 'Invalid credentials' });
}
const token = jwt.sign({ id: user.id, role: user.Role.name }, 'your_secret_key', {
expiresIn: '1h',
});
res.json({ token });
}
module.exports = { login };
```

7. Testing the Implementation

Use Postman or a similar tool to:

- ✓ Obtain a JWT by logging in with the correct credentials
- ✓ Access protected routes by passing the JWT in the Authorization header
- ✓ Verify that the RBAC and ABAC middleware correctly restrict or allow access.

Practical Activity 2.5.4: Implementing Custom Authorization Logic for Specific Use Cases

Task:

1. Read key reading 2.5.4.

2. Referring to the previous practical activities (2.5.3) you are requested to go to the computer lab to Implement Custom Authorization Logic for Specific Use Cases.

3. Apply safety precautions

4. Refer to the steps provided in key readings, then Implement Custom Authorization Logic for Specific Use Cases.

5. Present out the Steps in Implementing Custom Authorization Logic for Specific Use Cases.

Key readings 2.5.4: Implementing Custom Authorization Logic for Specific Use Cases

Implementing custom authorization logic allows you to tailor access control to fit the unique requirements of your application. This is particularly useful when Role-Based Access Control (RBAC) or Attribute-Based Access Control (ABAC) alone cannot fully address the complexities of your business logic. Here's how you can implement custom authorization logic for specific use cases.

1. Identify the Use Case

Before implementing custom authorization logic, clearly define the specific use case.

For example:

Only allow access to certain resources during business hours.

Allow users to modify records they own but restrict access to others.

Grant permissions based on a combination of user roles, attributes, and resource states.

2. Create a Custom Authorization Middleware

Start by creating a middleware function that encapsulates your custom logic. This middleware will intercept requests and apply your logic to determine whether access should be granted or denied.

Example Use Case:

Allowing Users to Edit Their Own Records

Let's say you want users to be able to edit only the records they created. The custom middleware would look like this:

```
// middleware/customAuthMiddleware.js
const Record = require('../models/recordModel');
function canEditRecord() {
return async (req, res, next) => {
const recordId = req.params.id; // Assuming the record ID is passed in the URL
const userId = req.user.id; // Assuming user ID is stored in req.user

const record = await Record.findByPk(recordId);

if (!record) {
return res.status(404).json({ message: 'Record not found' });
}

if (record.userId === userId) {
```

```
return next(); // User is authorized to edit the record
} else {
return res.status(403).json({ message: 'Access denied. You do not own this record.'
});
}
};
}
module.exports = canEditRecord;
```

3. Apply the Middleware to Relevant Routes

Apply your custom authorization middleware to the routes where you need to enforce the custom logic.

```
// routes/recordRoutes.js
const express = require('express');
const canEditRecord = require('../middleware/customAuthMiddleware');
const authenticateToken = require('../middleware/authMiddleware');
const router = express.Router();
router.use(authenticateToken); // Ensure user is authenticated
router.put('/records/:id', canEditRecord(), async (req, res) => {
// Your update logic here
const recordId = req.params.id;
const updatedData = req.body;
try {
const record = await Record.findByPk(recordId);
if (record) {
await record.update(updatedData);
res.json({ message: 'Record updated successfully', record });
} else {
res.status(404).json({ message: 'Record not found' });
}
} catch (error) {
res.status(500).json({ message: 'Error updating record', error: error.message });
  }
});

module.exports = router;
```

4. Test the Custom Logic

Testing your custom authorization logic involves verifying that the middleware works as expected under different scenarios:

Valid Case: A user tries to edit their own record and succeeds.

Invalid Case: A user tries to edit a record they don’t own and is denied access.

Edge Case: The record doesn’t exist, and the user receives a 404 error.

5. Handle Complex Scenarios

For more complex use cases, you might need to combine multiple conditions. For example, you may want to allow a user to edit their record only if it’s in a "draft" state and within their department.

Example Use Case:

Combining Role, Attribute, and Resource State

```
// middleware/complexAuthMiddleware.js
const Record = require('../models/recordModel');

function canEditDraftRecord() {
return async (req, res, next) => {
const recordId = req.params.id;
const userId = req.user.id;
const userRole = req.user.role; // Assume role is stored in req.user
const userDepartment = req.user.department; // Assume department is stored in req.user

const record = await Record.findByPk(recordId);

if (!record) {
return res.status(404).json({ message: 'Record not found' });
}

if (
record.userId === userId &&
record.state === 'draft' &&
userDepartment === record.department &&
 (userRole === 'editor' || userRole === 'admin')
) {
return next(); // User is authorized
} else {
return res.status(403).json({ message: 'Access denied. You are not authorized to edit this record.' });
}
};
}

module.exports = canEditDraftRecord;
```

6. Optimize for Performance

When implementing custom authorization logic, consider the performance impact, especially if the logic involves multiple database queries or complex computations. Optimize queries and consider caching results if applicable.

7. Document and Maintain the Custom Logic

Document your custom authorization logic clearly so that other developers can understand and maintain it. As your application evolves, revisit the logic to ensure it still aligns with business requirements.

Conclusion

Custom authorization logic provides the flexibility to implement specific access control requirements that go beyond standard RBAC and ABAC models. By creating middleware tailored to your unique use cases, you can enforce fine-grained security controls that meet your application's needs.

Custom Logic Example:

Implement custom logic, such as checking if a user owns a resource:

```
const checkOwnership = (req, res, next) => {
const resourceOwnerId = getResourceOwnerId(req.params.id);
if (req.user.id !== resourceOwnerId) {
return res.status(403).json({ message: "Forbidden" });
}
next();
};

app.get('/resource/:id', checkOwnership, (req, res) => {
res.send('Resource content');
});
```

Points to Remember

- While **implementation of Authorization** as key to system security, ensuring users only access what they’re permitted to. Techniques like Role-Based Access Control (RBAC) simplify permissions management by assigning roles rather than individual access rights.
- In Node.js, middleware checks user roles before allowing resource access, protecting system integrity.

- Custom authorization logic can further enhance security by addressing specific use cases, like verifying resource ownership.
- This fine-tunes access control, ensuring only authorized actions are taken. Proper implementation safeguards sensitive data and maintains system confidentiality.

Application of learning 2.5.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients. As a newly hired backend developer using Node.js, you are requested to **i**mplement role-based and attribute-based access control and Use authorization middleware to manage user permissions next you will Implement custom authorization logic for specific use cases.

Indicative content 2.6: Implementation of Accountability

Duration: 2 hrs

Theoretical Activity 2.6.1: Description of accountability.

Tasks:

1: You are requested to answer the following questions related to the description of accountability:

i. What is accountability?
ii. What are the principles of accountability?
iii. Explain the role of accountability in system security.

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.6.1. In addition, ask questions where necessary.

Key readings 2.6.1: Description of authorization.

4. Accountability

Accountability in system security ensures that every action in a system is attributable to a specific user, thus making users responsible for their actions.

principles

Traceability: The ability to trace actions back to the user who performed them.

Non-Repudiation: Ensuring users cannot deny their actions.

Auditing: Regularly reviewing logs to ensure that users are held accountable for their actions.

2. Roles of Accountability in System Security

1. Enforcing Security Policies

Accountability ensures that users adhere to these policies by tracking their actions. When users know that their activities are being monitored and recorded, they are more likely to follow the established security protocols. If a user violates a policy, the system's accountability mechanisms can identify the individual responsible, allowing for corrective actions such as warnings, retraining, or disciplinary measures. This enforcement helps maintain the integrity and security of the system.

2. **Detecting Malicious Behavior**

Malicious behavior refers to any actions taken by users or external entities that are intended to harm the system, steal data, or disrupt operations. This includes activities such as unauthorized access, data breaches, or the introduction of malware.

3. **Providing Evidence in the Event of a Security Breach**

A security breach occurs when an unauthorized entity gains access to a system or data, potentially causing harm to the organization. After a breach, it's essential to understand how it happened, who was responsible, and what data or systems were affected.

Practical Activity 2.6.2 Implementing Logging and Auditing Features

Task:

1. Read key reading 2.6.2.

2. Referring to the previous theoretical activities (2.6.1) you are requested to go to the computer lab to Implement Logging and Auditing Features in Node.js using Popular Libraries like Winston and Morgan

3. Apply safety precautions

4. Refer to the steps provided on task 4, then Implement Logging and Auditing Features in an existing project.

5. Present out the Steps in Implementing Logging and Auditing Features in Node.js using Popular.

Key readings 2.6.2: Implementing Logging and Auditing Features in Node.js using Popular Libraries

1. **Using Winston**

Setup Winston

```
npm install winston
```

Configure Winston

```
const winston = require('winston');
const logger = winston.createLogger({
level: 'info',
format: winston.format.json(),
transports: [
new winston.transports.File({ filename: 'error.log', level: 'error' }),
new winston.transports.File({ filename: 'combined.log' })
]
});
```

Log Messages

```
logger.info('Information message');
logger.error('Error message');
```

2. **Using Morgan**

Setup Morgan

```
npm install morgan
```

Configure Morgan

```
const morgan = require('morgan');
app.use(morgan('combined', { stream: logger.stream.write }));
```

Theoretical Activity 2.6.3: Best practices for storing log data and protecting it from unauthorized access

Tasks:

1: You are requested to answer the following questions related to the Best practices for securely storing log data and protecting it from unauthorized access:

I. Explain the best practices for securely storing log data and protecting it in system security

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class and the trainer

4: For more clarification, read the key readings 2.6.3. In addition, ask questions where necessary.

Key readings 2.6.3.: Best practices for storing log data and protecting it from unauthorized access

1. Use a Centralized Logging System

Implement a centralized logging solution (e.g., ELK Stack, Splunk) to aggregate logs from multiple sources. This makes it easier to manage and secure logs.

2. Restrict Access

Limit access to log data to only those who need it. Use role-based access control (RBAC) to manage permissions effectively.

3. Encrypt Log Data

Encrypt logs both in transit (using TLS/SSL) and at rest (using AES or similar encryption algorithms) to protect sensitive information.

4. Implement Log Retention Policies

Define and enforce log retention policies to determine how long logs should be stored. Regularly purge old logs to reduce exposure.

5. Mask Sensitive Information

Avoid logging sensitive information (e.g., passwords, credit card numbers). If necessary, mask or hash sensitive data before logging.

6. Use Secure Storage Solutions

Store logs in secure environments, such as dedicated logging servers or cloud services with strong security measures (e.g., AWS CloudWatch, Azure Monitor).

7. Monitor Access to Logs

Implement monitoring and alerting for any unauthorized access attempts to log data. Use intrusion detection systems (IDS) to catch suspicious activities.

8. Implement Audit Trails

Keep an audit trail of who accessed the logs and when. This can help in identifying unauthorized access and tracking down issues.

9. Regularly Update Logging Practices

Stay updated with best practices and compliance requirements (e.g., GDPR, HIPAA) that may affect how you log and store data.

10. Backup Logs Securely

Regularly back up log data and store backups in a secure location. Ensure that backups are also encrypted and access-controlled.

11. Use Log Rotation

Implement log rotation to manage log file sizes and prevent filling up storage. This also helps in maintaining performance.

12. Conduct Regular Security Audits

Perform regular security audits and reviews of your logging practices to ensure compliance and identify potential vulnerabilities.

Practical Activity 2.6.4: Detection of Security Events and System Errors

Task:

1. Read key reading 2.6.4.

2. Referring to the previous theoretical activities (2.6.3) you are requested to go to the computer lab to Audit Logs to Detect Security Events and System Errors

3. Apply safety precautions

4. Refer to the steps provided in key readings, then Audit Logs in an existing project.

5. Present out the Steps in Auditing Logs to Detect Security Events and System Errors

Key readings 2.6.4: Detection of Security Events and System Errors

Detecting security events and system errors in Node.js is crucial for maintaining application integrity and ensuring user safety. Here's a comprehensive guide on how to implement effective detection mechanisms:

1. Use Proper Logging

Implement Structured Logging: Utilize libraries like winston or bunyan to create structured logs that capture various levels of information (info, warn, error, debug).

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

const logger = winston.createLogger({

level: 'info',

format: winston.format.json(),

transports: [

new winston.transports.File({ filename: 'error.log', level: 'error' }),

new winston.transports.Console(),
],

});
```

2. Error Handling Middleware

Centralized Error Handling: Set up middleware to catch and log errors systematically.

```
app.use((err, req, res, next) => {

logger.error(err.message);

res.status(500).send('Internal Server Error');

});
```

3. Monitor Security Events

Use Security Libraries: Implement libraries like helmet to automatically set security-related HTTP headers.

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

app.use(helmet());
```

Track Authentication Events: Log authentication events, such as successful logins, failed attempts, and password resets.

4. Integrate with Monitoring Tools

Use APM Tools: Application Performance Monitoring (APM) tools like New Relic, Datadog, or Sentry can help detect anomalies and errors in real-time.

Set Up Alerts: Configure alerts for specific events, such as multiple failed login attempts or unexpected behavior.

5. Real-time Monitoring and Alerts

WebSocket for Real-time Updates: Use WebSocket to provide real-time updates on security events to the admin dashboard.

6. Use Security Auditing Libraries

Integrate Auditing Tools: Libraries like express-requests-logger can help log incoming requests, which can be reviewed for unusual patterns.

7. Implement Rate Limiting

Protect Against DDoS Attacks: Use libraries like express-rate-limit to limit the number of requests from a single IP address.

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({

windowMs: 15 * 60 * 1000, // 15 minutes

max: 100, // limit each IP to 100 requests per windowMs

});

app.use(limiter);
```

8. Perform Input Validation

Sanitize Inputs: Use libraries like express-validator to validate and sanitize inputs to prevent SQL injection and XSS attacks.

```
const { body, validationResult } = require('express-validator');

app.post('/user', [

body('email').isEmail(),

body('password').isLength({ min: 5 })

], (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}
// Continue with user creation

});
```

9. Monitor System Resource Usage

Use Node.js Modules: Monitor memory and CPU usage using built-in modules like os or external libraries like node-os-utils.

10. Regular Security Audits

Conduct Regular Audits: Regularly review code, libraries, and dependencies for vulnerabilities using tools like npm audit or Snyk.

Auditing Example

1. **Setup Alerts**

Configure Winston to send alerts for specific events:

```
logger.on('logging', (transport, level, msg, meta) => {
if (level === 'error') {
sendAlert(msg);
}
});
```

2. **Analyze Logs**

Periodically review logs for suspicious activity:

```
const logs = fs.readFileSync('combined.log', 'utf-8');
const errors = logs.split('\n').filter(log => log.includes('error'));
console.log(errors);
```

Points to Remember

- While implementing Accountability allow to ensure all system actions are traceable to specific users, holding them responsible. Principles like Traceability and Non-Repudiation are enforced in Node.js through logging, using tools like Winston.
- Secure log storage and monitoring are vital for detecting and preventing unauthorized actions.
- Best practices include encrypting log data, limiting access, and centralizing logs. Regular audits help identify security threats, reinforcing system security and ensuring compliance with security policies.

Application of learning 2.6.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients. As a newly hired backend developer using Node.js, you are requested to Implement logging and auditing features in Node.js using popular libraries like Winston and Morgan and then Audit logs to detect security events and system errors.

Indicative content 2.7: Secure Environment Variables

Duration: 1 hrs

Theoretical Activity 2.7.1: Description of environment variable.

Tasks:

1: You are requested to answer the following questions related to description of environment variable:

i. What is the difference between types of Information Stored in Environment Variables?

ii. What are the potential security risks of storing sensitive information in environment variables?

iii. Explain the best practices for managing and securing environment variables in Node.js

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.7.1. In addition, ask questions where necessary.

Key readings 2.7.1: Description of environment variable

Environment variables in Node.js are key-value pairs that store configuration settings and sensitive information, such as database credentials, API keys, and application settings. They provide a way to manage the configuration of your application without hardcoding values directly into your code, allowing for flexibility and security.

1. **Types of Information Stored in Environment Variables**

- **Database Credentials:** Usernames and passwords required to access databases.
- **API Keys:** Keys used to authenticate requests to third-party APIs.
- **Encryption Keys:** Keys used for encrypting and decrypting data.

2. Potential Security Risks

- **Exposure**: If not properly protected, environment variables can be exposed, leading to security breaches.

- **Injection Attacks:** Improper handling can lead to injection attacks where attackers manipulate environment variables.

3. **Best Practices**

- **Use .env files:** Store environment variables in a .env file and never commit this file to version control.
- **Environment-Specific Variables:** Use different variables for development, staging, and production environments.
- **Access Controls:** Limit access to environment variables to authorized users and services only.

Practical Activity 2.7.2 Protecting environment variables

Task:

1. Read key reading 2.7.2.

2. Referring to the previous theoretical activities (2.7.1) you are requested to go to the computer lab to implement security measures for protecting environment variables by encrypting secrets and decrypting secrets

3. Apply safety precautions

4. Refer to the steps provided in key readings, and then implement security measures in an existing project.

5. Present out the Steps in implementing security measures for protecting environment variables.

Key readings 2.7.2: Protecting environment variables

Protecting environment variables is crucial for maintaining the security and integrity of your application.

Best practices to safeguard your environment variables in Node.js:

1. Use a .env File for Local Development

Store Locally: Use a .env file to manage environment variables locally. This file should not be committed to version control.

Add to .gitignore: Ensure your .env file is listed in your .gitignore to prevent it from being accidentally pushed to repositories.

```
# .gitignore
.env
```

2. Limit Access

Restrict Access to Variables: Ensure that only the necessary processes and users have access to the environment variables. Use OS-level permissions to control access.

3. Use Environment-Specific Configurations

Separate Configurations: Create separate environment variable files for different environments (e.g., .env.development, .env.production) and load the appropriate one based on the environment.

4. Use Secure Storage Solutions

Cloud Providers: When deploying your application, use secure storage solutions provided by cloud platforms (e.g., AWS Secrets Manager, Azure Key Vault, Google Cloud Secret Manager) to store sensitive environment variables securely.

5. Encrypt Sensitive Data

Encryption: If you must store sensitive information in environment variables, consider encrypting the values. Ensure that your application can decrypt them at runtime.

6. Validate Environment Variables

Check for Required Variables: At application startup, validate the presence and format of critical environment variables to prevent runtime errors.

```
const requiredVars = ['DB_USER', 'DB_PASSWORD', 'API_KEY'];
requiredVars.forEach((v) => {
if (!process.env[v]) {
throw new Error(`Missing required environment variable: ${v}`);
}
});
```

7. Monitor and Audit Access

Logging and Monitoring: Implement monitoring to log access to sensitive environment variables. Regularly audit who has access to these variables.

8. Use CI/CD Secrets Management

Integrate with CI/CD Tools: Use built-in secrets management features in CI/CD tools (e.g., GitHub Actions Secrets, GitLab CI/CD Variables) to securely pass environment variables during deployment.

9. Rotate Secrets Regularly

Regular Rotation: Regularly update and rotate sensitive keys and passwords to minimize the risk of exposure.

10. Educate Your Team

Security Awareness: Train your team on the importance of protecting environment variables and the best practices for handling sensitive data.

Practical Activity 2.7.3: Storing environment variables

Task:

1. Read key reading 2.7.3.

2. Referring to the previous practical activities (2.7.2) you are requested to go to the computer lab to Store environment variables in a secure location like key management service and env file

3. Apply safety precautions

5. Refer to the steps provided in key readings, and then Store environment variables in a secure location in an existing project.

4. Present out the steps in storing environment variables in a secure location.

Key readings 2.7.3: Storing environment variables

To store environment variables securely, you can use both a Key Management Service (KMS) provided by cloud providers and .env files for local development.

Storing Environment Variables in .env Files

1. **Create a .env File**

In the root of your Node.js project, create a .env file.

Add your environment variables to this file:

```
DB_HOST=localhost
DB_USER=root
DB_PASS=s1mpl3
```

2. **Install dotenv Package**

Install the dotenv package to load environment variables from the .env file:

```
npm install dotenv
```

3. Load Environment Variables in Your Application

In your application's entry file (e.g., app.js), add the following code at the top to load the environment variables:

```
require('dotenv').config();
console.log('Database Host:', process.env.DB_HOST);
console.log('Database User:', process.env.DB_USER);
```

4. Secure Your .env File

Add .env to .gitignore: Ensure that the .env file is not included in version control:

```
# .gitignore
.env
```

Limit File Access: Set file permissions to restrict access to the .env file.

Practical Activity 2.7.4: Management and loading environment variables

Task:

1. Read key reading 2.7.4.

2. Referring to the previous practical activities (2.7.2) you are requested to go to the computer lab to Manage and load environment variables in Node.js applications using dotenv

3. Apply safety precautions

4. Refer to the steps provided in key readings, and then Manage and load environment variables in Node.js applications using dotenv in an existing project.

5. Present out the Steps in Managing and loading environment variables in Node.js applications using dotenv.

Key readings 2.7.4: Management and loading environment variables

Managing and loading environment variables in a Node.js application using the dotenv library is straightforward and highly effective for keeping your sensitive information secure and organized.

Below are the steps to set it up

1. Install the dotenv Package

First, you'll need to install the dotenv package in your Node.js project. This package will allow you to load environment variables from a .env file into process.env.

```
npm install dotenv
```

2. Create a .env File

Create a .env file in the root directory of your project. This file will hold your environment variables.

```
touch .env
```

3. Define Environment Variables in the .env File

Inside your .env file, you can define your environment variables using the KEY=VALUE format.

Here's an example

```
# .env
PORT=3000
DB_HOST=localhost
DB_USER=myuser
DB_PASS=mypassword
JWT_SECRET=mysecretkey
```

4. Load Environment Variables in Your Application

To load these environment variables into your Node.js application, require the dotenv package at the very beginning of your entry file (e.g., server.js or app.js).

```
// server.js or app.js
require('dotenv').config();
// Now you can access the variables via process.env
const port = process.env.PORT;
const dbHost = process.env.DB_HOST;
const dbUser = process.env.DB_USER;
const dbPass = process.env.DB_PASS;
const jwtSecret = process.env.JWT_SECRET;
console.log(`Server running on port ${port}`);
```

5. Access Environment Variables

You can now access the environment variables anywhere in your application using process.env.VARIABLE_NAME.

```
const express = require('express');
const app = express();
const port = process.env.PORT || 3000;
```

```
app.get('/', (req, res) => {
  res.send('Environment Variables Loaded Successfully!');
});
app.listen(port, () => {
console.log(`Server is running on port ${port}`);
});
```

6. Add .env to .gitignore

To ensure that your .env file isn’t committed to your version control (e.g., Git), add it to your .gitignore file.

```
echo .env >> .gitignore
```

7. Create Environment-Specific .env Files (Optional)

If you have different configurations for different environments (e.g., development, testing, production), you can create multiple .env files, such as .env.development, .env.production, etc.

To load a specific .env file, you can modify your require('dotenv').config() call:

```
require('dotenv').config({ path: './.env.development' });
```

8. Use Environment Variables in Your Application

Now, you can use process.env.VARIABLE_NAME throughout your application for any configurations or sensitive information.

9. Validate Environment Variables (Optional but Recommended)

You can use a validation library like joi to validate your environment variables at the start of your application.

npm install joi

```
const Joi = require('joi');
const envVarsSchema = Joi.object({
PORT: Joi.number().required(),
DB_HOST: Joi.string().required(),
DB_USER: Joi.string().required(),
DB_PASS: Joi.string().required(),
JWT_SECRET: Joi.string().required(),
}).unknown().required();
const { error } = envVarsSchema.validate(process.env);
if (error) {
throw new Error(`Config validation error: ${error.message}`);
}
```

10. Deploying to Production

When deploying to production, you'll need to set environment variables on the production server, as the .env file should not be used in production. You can do this directly in the server environment configuration.

Theoretical Activity 2.7.5: Best practices for passing environment variables to other services

Tasks:

1: You are requested to answer the following questions related to the Best practices for safely passing environment variables to other services and applications:

i. Explain best practices for safely passing environment variables to other services and applications

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class and the trainer

4: For more clarification, read the key readings 2.7.5. In addition, ask questions where necessary.

Key readings 2.7.5: Best practices for passing environment variables to other services

Passing environment variables securely and efficiently between services is crucial for maintaining the security, scalability, and maintainability of your applications. Here are some best practices to consider:

1. Use .env File for Local Development

Store environment variables in a .env file: This file should be kept outside of version control to avoid exposing sensitive information (e.g., API keys, database credentials).

Load the .env file using a library: In Node.js, you can use the dotenv package to load environment variables from a .env file into process.env.

```
npm install dotenv
```

```
require('dotenv').config();
```

2. Use Environment-Specific Configuration Files

Separate configuration files for different environments: Use different .env files or configuration files for development, testing, and production environments (e.g., .env.development, .env.production).

Conditionally load configurations: Use a script or environment-specific logic to load the correct configuration based on the environment.

3. Securely Store Environment Variables in Production

Environment Variables Management Systems: Use secret management tools like AWS Secrets Manager, Azure Key Vault, or HashiCorp Vault to store and manage environment variables securely.

Avoid Hardcoding Sensitive Information: Never hardcode sensitive data (like API keys or passwords) directly in your source code. Always use environment variables.

4. Inject Environment Variables via CI/CD Pipelines

Set Environment Variables in CI/CD Pipelines: When deploying applications, set environment variables in the CI/CD pipeline rather than storing them in the codebase or configuration files.

Use Pipeline Secrets/Variables: Most CI/CD tools allow you to define secrets or environment variables that are only accessible during the pipeline execution.

5. Pass Environment Variables Securely to Docker Containers

Use --env-file Option: Pass environment variables to Docker containers using an env-file or directly via the -e flag.

```
docker run --env-file .env myservice
```

Avoid Committing Secrets: Ensure that any .env files used for Docker are not committed to version control.

6. Use Environment Variables for Configuration in Kubernetes

Define Environment Variables in Kubernetes ConfigMaps or Secrets: Use

Kubernetes ConfigMaps for non-sensitive data and Secrets for sensitive information.

Mount ConfigMaps or Secrets as Environment Variables: Inject these values into your containers as environment variables.

```
apiVersion: v1
kind: Secret
metadata:
name: my-secret
type: Opaque
data:
api-key: base64-encoded-value
```

7. Validate Environment Variables

Use validation libraries: Validate environment variables at the application startup to ensure all required variables are set and correctly formatted.

Fail fast on missing variables: If an expected environment variable is missing or incorrect, the startup process prevents the application from running in an unstable state.

8. Minimize Environment Variable Exposure

Limit the scope: Only pass the necessary environment variables to each service. Avoid passing variables that aren't needed by a specific service.

Use the least privilege principle: Ensure that services have access only to the variables they require and nothing more.

9. Rotate and Monitor Environment Variables

Regularly rotate secrets and credentials: Implement automated processes to rotate secrets and update them across your services.

Monitor access: Use monitoring tools to track the usage of sensitive environment variables and detect any unauthorized access.

10. Document Environment Variables

Document required environment variables: Provide clear documentation on the environment variables used by your application, including their purpose, default values, and how they should be configured for different environments.

Points to Remember

- **To secure environment variables** is crucial for protecting sensitive information. Storing variables in .env files, separated by environment, and restricting access are key practices. Avoiding exposure through version control and encrypting sensitive data adds layers of security.
- Using encryption tools in Node.js and Key Management Services (KMS) from providers like AWS ensures variables are protected even if exposed. These measures reduce risk and maintain secure application environments.

Application of learning 2.7.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients. As a newly hired backend developer using Node.js, you are requested to Implement security measures for protecting environment variables by Encrypting secrets and Decrypting secrets then Storing environment variables in a secure location like key management service and env file next you will manage and load environment variables in Node.js applications using dotenv.

Indicative content 2.8: Monitor and Manage Environment Variables

Duration: 1 hr

Practical Activity 2.8.1: Implementing Logging and Auditing Features

Task:

1. Read key reading 2.8.1.

2. Referring to the previous theoretical activities (2.7.5) you are requested to go to the computer lab to Implement Logging and Auditing Features to Detect Unauthorized Access to Environment Variables

3. Apply safety precautions

4. Present the Steps in Implementing Logging and Auditing Features to Detect Unauthorized Access to Environment Variables.

5. Refer to the steps provided in key reading, Implementing Logging and Auditing Features in an existing project.

Key readings 2.8.1: Implementing Logging and Auditing Features

Implementing logging and auditing features to detect unauthorized access to environment variables is crucial for securing your Node.js application.

Here are the steps to follow

1. Set Up Logging Framework

Start by integrating a logging framework into your Node.js application to capture and manage logs.

Choose a Logging Library: Popular logging libraries include winston, morgan, and bunyan.

```
npm install winston
```

Configure Logging: Create a logger instance that logs various levels of information (e.g., info, warn, error) to files and/or console.

```
const winston = require('winston');
const logger = winston.createLogger({
level: 'info',
format: winston.format.json(),
transports: [
new winston.transports.File({ filename: 'error.log', level: 'error' }),
new winston.transports.File({ filename: 'combined.log' }),
],
});

// If we're not in production, log to the console as well
if (process.env.NODE_ENV !== 'production') {
logger.add(new winston.transports.Console({
format: winston.format.simple(),
}));
}
module.exports = logger;
```

2. Monitor Environment Variable Access

To detect unauthorized access to environment variables, you can create a wrapper or a custom function that logs when sensitive variables are accessed.

Create an Access Logging Function: Log whenever sensitive environment variables are accessed.

```
const logger = require('./logger');
function getEnvVariable(key) {
const value = process.env[key];
if (value) {
logger.info(`Environment variable ${key} accessed.`);
} else {
logger.warn(`Attempt to access non-existing environment variable: ${key}`);
}

return value;
}

// Usage
const dbUser = getEnvVariable('DB_USER');
```

3. Implement Auditing

Auditing goes beyond logging by systematically recording access patterns and anomalies.

Log All Access Attempts: Use your logging framework to record every time an environment variable is accessed.

```
function auditEnvAccess(key) {
  logger.info(`Audit: Access to ${key} at ${new Date().toISOString()}`);
}

function getEnvVariableWithAudit(key) {
```

```
auditEnvAccess(key);
return process.env[key];
}
```

Store Logs in a Centralized System: For better audit tracking, consider using centralized logging solutions like ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, or a cloud-based solution like AWS CloudWatch.

4. Detect and Respond to Unauthorized Access

Use your logs to detect unauthorized access patterns, such as attempts to access variables that should not be accessible.

Set Up Alerts: Configure alerts to notify administrators when unauthorized access is detected.

```
const unauthorizedKeys = ['SECRET_KEY', 'JWT_SECRET'];
function alertUnauthorizedAccess(key) {
if (unauthorizedKeys.includes(key)) {
logger.error(`Unauthorized access attempt detected for ${key}`);
// Trigger alert or notify admin
}
}

function getEnvVariableWithAlert(key) {
alertUnauthorizedAccess(key);
return process.env[key];
}
```

Review Logs Regularly: Regularly review the logs to identify any suspicious activity and take action.

5. Implement Role-Based Access Control (RBAC)

Ensure that only authorized parts of your application or specific users can access certain environment variables.

Restrict Access: Implement RBAC to limit access to sensitive variables based on user roles or application components.

```
function getEnvVariableSecurely(key, userRole) {
const restrictedVariables = ['DB_PASS', 'JWT_SECRET'];
if (restrictedVariables.includes(key) && userRole !== 'admin') {
logger.warn(`Unauthorized access attempt by ${userRole} to ${key}`);
throw new Error('Access Denied');
}

return process.env[key];
}
```

6. Encrypt Sensitive Environment Variables

For additional security, encrypt sensitive environment variables and decrypt them only when needed.

Encrypt Variables: Use encryption tools to encrypt sensitive variables in the .env file.

Decrypt on Access: Decrypt them securely when accessing them in your application.

7. Regularly Audit and Rotate Secrets

Rotate Secrets Periodically: Regularly rotate API keys, passwords, and other sensitive variables to minimize exposure.

Conduct Audits: Periodically audit the environment variable access logs to ensure compliance and security.

8. Implement Secure Logging Practices

Avoid Logging Sensitive Data: Ensure that sensitive data like passwords or tokens are never logged in plaintext.

Anonymize Data: If necessary, anonymize sensitive information before logging it.

9. Integrate with Monitoring and Alerting Systems

Use Monitoring Tools: Integrate with monitoring tools like Prometheus, Grafana, or Datadog to visualize and track access patterns.

Set Up Alerts: Configure alerts to notify of any unusual access attempts or patterns.

10. Review and Update Security Policies

Regularly review and update your security policies to reflect the latest best practices and to address any new security threats.

Practical Activity 2.8.2: Monitoring Changes to Environment Variables

Task:

1. Read key reading 2.8.2.

2. Referring to the previous practical activities (2.8.1) you are requested to go to the computer lab to Monitor Changes in Environment Variables and Detecting Any Suspicious Activity

3. Apply safety precautions

4. Refer to the steps provided in key readings, and then monitoring changes to environment variables and detecting any suspicious activity in an existing project.

5. Present out the Steps in monitoring changes to environment variables and detecting any suspicious activity.

Key readings 2.8.2: Monitoring Changes to Environment Variables

Monitoring changes in environment variables and detecting suspicious activity in a Node.js application is crucial for maintaining security and ensuring the integrity of your environment.

1. Set Up a Baseline for Environment Variables

Start by establishing a baseline for your environment variables. This involves capturing the initial state of all environment variables when your application starts.

Capture Initial State

```
const initialEnv = { ...process.env };
```

2. Implement a Monitoring Mechanism

Monitor the environment variables throughout the application's runtime to detect any changes. You can create a function that compares the current state of environment variables with the initial state.

Create a Monitoring Function

```
const logger = require('./logger');
function monitorEnvChanges() {
Object.keys(process.env).forEach((key) => {
if (process.env[key] !== initialEnv[key]) {
logger.warn(`Environment variable ${key} has been modified. Original: ${initialEnv[key]}, New: ${process.env[key]}`);
// Trigger alert or take action based on the change
}
});
}
```

3. Schedule Regular Monitoring

To continuously monitor environment variables, you can use setInterval to check for changes at regular intervals.

Set Up Regular Checks:

```
const monitorInterval = 60000; // Check every 60 seconds
setInterval(monitorEnvChanges, monitorInterval);
```

4. Implement Alerting for Suspicious Activity

If an environment variable is changed unexpectedly, it’s important to alert administrators or log the suspicious activity.

Alert on Suspicious Activity:

```
function monitorEnvChanges() {
  Object.keys(process.env).forEach((key) => {
if (process.env[key] !== initialEnv[key]) {
logger.warn(`Suspicious change detected in environment variable ${key}.`);
// Send alert (email, SMS, etc.) to administrators
sendAlert(`Environment variable ${key} has been modified.`);
}
});
}

function sendAlert(message) {
// Integrate with your alerting system (e.g., email, SMS, Slack)
console.log(`Alert: ${message}`);
}
```

5. Log All Changes for Auditing

Ensure that all detected changes are logged for auditing purposes. This will allow you to review the history of changes and investigate any suspicious activities.

Log Changes:

```
function monitorEnvChanges() {
Object.keys(process.env).forEach((key) => {
if (process.env[key] !== initialEnv[key]) {
logger.warn(`Environment variable ${key} changed from ${initialEnv[key]} to ${process.env[key]} at ${new Date().toISOString()}`);
// You can store these logs in a centralized logging system for further analysis
}
});
}
```

6. Implement Role-Based Access Control (RBAC)

Ensure that only authorized users or components of your application can modify environment variables.

Restrict Access:

```
const restrictedKeys = ['DB_PASS', 'SECRET_KEY'];
function secureEnvChange(key, value, userRole) {
if (restrictedKeys.includes(key) && userRole !== 'admin') {
logger.error(`Unauthorized attempt by ${userRole} to change environment variable ${key}`);
throw new Error('Access Denied');
}
process.env[key] = value;
}
```

7. Use Immutable Variables When Possible

If certain environment variables should never change during runtime, enforce immutability for those variables.

Enforce Immutability:

```
const immutableKeys = ['DB_HOST', 'JWT_SECRET'];
function monitorEnvChanges() {
Object.keys(process.env).forEach((key) => {
if (immutableKeys.includes(key) && process.env[key] !== initialEnv[key]) {
logger.error(`Attempt to modify immutable environment variable ${key}. Original: ${initialEnv[key]}, New: ${process.env[key]}`);
// Revert to the original value
process.env[key] = initialEnv[key];
}
});
}
```

8. Integrate with a Monitoring Tool

To enhance monitoring, integrate with a monitoring tool like Prometheus, Datadog, or New Relic to track changes and performance metrics related to environment variables.

Integration Example with Prometheus:

```
const client = require('prom-client');
const Gauge = client.Gauge;
const envChangeGauge = new Gauge({
name: 'env_variable_changes',
help: 'Number of changes detected in environment variables',
});

function monitorEnvChanges() {
let changesDetected = 0;
Object.keys(process.env).forEach((key) => {
if (process.env[key] !== initialEnv[key]) {
changesDetected++;
}
});
envChangeGauge.set(changesDetected);
}
```

9. Implement Regular Audits

Regularly audit the environment variable access logs and change history to ensure compliance and detect any patterns of unauthorized access or modification.

Schedule Audits:

```
// This could be a manual or automated process
function auditEnvAccessLogs() {
// Review the logs and investigate any suspicious changes
}

// Run audit weekly
setInterval(auditEnvAccessLogs, 7 * 24 * 60 * 60 * 1000); // Every 7 days
```

10. Review and Update Security Policies

Regularly review and update your security policies to address new threats and ensure that your monitoring system is up to date with the latest best practices.

Monitoring Example

1. **Track Changes:**

Monitor changes to your .env file:

```
const fs = require('fs');
fs.watchFile('.env', (curr, prev) => {
console.log('Environment variables changed!');
});
```

Theoretical Activity 2.8.3: Best Practices for Managing and Rotating Environment Variables

Tasks:

1: You are requested to answer the following questions related to the Best Practices for Managing and Rotating Environment Variables to Prevent Data Breaches:

I. Explain best practices for Managing and Rotating Environment Variables to Prevent Data Breaches

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class and trainer

4: For more clarification, read the key readings 2.8.3. In addition, ask questions where necessary.

Key readings 2.8.3: Best Practices for Managing and Rotating Environment Variables

- **Best Practices for Managing and Rotating Environment Variables to Prevent Data Breaches**

1. **Store Environment Variables Securely:**
   Use .env files for local development but ensure they are excluded from version control (e.g., by adding. env to .gitignore).
   In production, use secure storage solutions like AWS Secrets Manager, Azure Key Vault, or environment management tools like HashiCorp Vault to store environment variables.
2. **Encrypt Sensitive Variables:**
   Encrypt sensitive environment variables, such as API keys and database credentials, both at rest and in transit. Decrypt them only when needed within your application.
3. **Implement Role-Based Access Control (RBAC):**
   Restrict access to environment variables based on roles. Ensure that only authorized personnel and components have access to modify or view sensitive environment variables.
4. **Regularly Rotate Secrets:**
   Regularly rotate environment variables, especially those related to authentication, such as API keys, passwords, and tokens. Automate this process if possible to reduce the risk of stale secrets being exploited.
5. **Monitor Access and Changes:**
   Log all access to environment variables and monitor for any unauthorized changes. Use tools like Prometheus, Datadog, or centralized logging systems to track and alert on suspicious activities.
6. **Avoid Hardcoding Secrets:**
   Never hardcode sensitive information directly in the source code. Always reference environment variables or secure vaults for fetching such information.
7. **Implement Immutability for Certain Variables:**
   For variables that should not change during runtime, enforce immutability by locking them or setting up monitoring to revert any unauthorized changes immediately.
8. **Use Environment-Specific Configuration Files:**
   Separate environment variables for different environments (development, testing, production) using environment-specific configuration files. This reduces the risk of accidentally exposing sensitive information in less secure environments.
9. **Audit and Review Regularly:**

Conduct regular audits of environment variables and their usage to ensure compliance with security policies. Update and review security practices frequently to adapt to new threats.

10. **Secure Your CI/CD Pipeline:**
    Ensure that your continuous integration and deployment pipelines handle environment variables securely, with encryption and restricted access, to prevent leaks during deployment.

Points to Remember

- While monitoring and managing environment variables helps maintain system security. Implementing logging to track changes in .env files detects unauthorized access. Regular audits and secure management practices like rotating sensitive variables further reduce risks.
- Automating rotations and limiting access to environment variables prevent unauthorized changes. These strategies ensure tight control over sensitive information, keeping systems secure and reliable.

Application of learning 2.8.

STU LTD is a software development company located in Musanze district, specializing in both frontend and backend solutions for various clients. As a newly hired backend developer using Node.js, you are requested to Monitor and manage environment variables by implementing logging to track changes in .env files detects unauthorized access.

Learning outcome 2 end assessment

Written assessment

I: Multiple Choice Questions

1.Which type of encryption uses the same key for both encryption and decryption?

A) Symmetric Encryption

B) Asymmetric Encryption

C) Hashing

D) None of the above

2. Which algorithm is commonly used for symmetric encryption in Node.js?

A) RSA

B) AES

C) SHA-256

D) bcrypt

3.In asymmetric encryption, which key is used to decrypt the data?

A) Public Key

B) Private Key

C) Symmetric Key

D) None of the above

4.Which Node.js module provides built-in support for cryptographic operations?

A) fs

B) http

C) crypto

D) path

5.Which of the following is NOT a use case for hashing?

A) Storing passwords securely

B) Encrypting large amounts of data

C) Verifying data integrity

D) Generating fixed-size hash values

6.What is the output size of a SHA-256 hash?

A) 128 bits

B) 256 bits

C) 512 bits

D) 1024 bits

7.Which of the following is a disadvantage of symmetric encryption?

A) Speed

B) Key distribution

C) Complexity

D) Security

8.What does the bcrypt library primarily do?

A) Encrypt files

B) Hash passwords

C) Generate keys

D) Create SSL certificates

9.Which encryption method is generally slower?

A) Symmetric

B) Asymmetric

C) Both are equally fast

D) Hashing

10.In the context of encryption, what does IV stand for?

A) Initialization Vector

B) Interchangeable Variable

C) Inner Value

D) Independent Verification

II: Respond to the followings by True or False

1.Asymmetric encryption uses the same key for both encryption and decryption.

2.The Moment.js library in Node.js is primarily used for date manipulation.

3.The crypto module in Node.js supports both symmetric and asymmetric encryption techniques.

4.Using bcrypt in Node.js automatically handles the addition of salt to the hashing process.

5.The package.json file in a Node.js project keeps track of dependencies and their versions.

6.Hash functions are reversible.

7.Asymmetric encryption is more suitable for bulk data encryption than symmetric encryption.

8.A public key can be shared openly while the private key must be kept secret.

9.All cryptographic algorithms are equally secure.

10.Hashing can be used to securely store passwords.

III: Fill in the Blank with correct key word from the listed ones

1.Symmetric encryption is typically used for encrypting large amounts of data ___.

A) quickly B) slowly C) securely D) publicly

2.The ___ module in Node.js supports various cryptographic functions.

A) fs B) crypto C) path D) url

3.In ____ encryption, a pair of keys (public and private) is used, with one key encrypting the data and the other decrypting it.

A) symmetric B) asymmetric C) hashing D) block

4.The bcrypt library is commonly used in Node.js for securely ___.

A) hashing passwords B) encrypting files C) generating keys D) decoding messages

5.A ___-bit key and a 16-byte initialization vector (IV) are generated for AES encryption.

A) 128 B) 256 C) 512 D) 1024

6.___ is often used to prevent brute-force attacks on hashed passwords.

A) Salting B) Encoding C) Hashing D) Encrypting

7.___ encryption is more efficient for encrypting large amounts of data.

A) Symmetric B) Asymmetric C) Hashing D) Hybrid

8.The ___ algorithm is commonly used in asymmetric encryption to generate public and private keys.

A) SHA-1 B) RSA C) AES D) MD5

9.To store passwords securely, we use ___, not encryption.

A) Encoding B) Hashing C) Public Key D) Decryption

10.The process of converting plaintext into ciphertext is known as ___.

A) Hashing B) Decryption C) Encryption D) Encoding

IV: Read the following statement related to secure backend application and write the number corresponding to the correct description

Answer	Encryption Type	Description
…………….	A) Symmetric	1) Uses a pair of keys (public and private)
…………….	B) Asymmetric	2) Uses the same key for encryption and decryption
………………	C) Hashing	3) Produces a fixed-size output from input data
……………….	D) Encryption	4) Converts plaintext into ciphertext

V: Multiple Choice Questions

1. Which of the following is a secure hash function?
   A) SHA-1
   B) MD5
   C) SHA-256
   D) All of the above
2. What is the purpose of salt in hashing?
   A) To increase speed
   B) To add randomness
   C) To encrypt data
   D) To create keys
3. Which is an example of a one-way hash function?
   A) AES
   B) RSA
   C) SHA-512
   D) Blowfish
4. What type of attack does salting help prevent?
   A) Replay attacks
   B) Man-in-the-middle attacks
   C) Rainbow table attacks
   D) Denial of service attacks
5. Which cryptographic operation is reversible?
   A) Hashing
   B) Encryption
   C) Salting
   D) None of the above
6. What is the main purpose of the initialization vector (IV)?
   A) To hash data
   B) To ensure randomness
   C) To create keys
   D) To compress data

7. Which of the following is an example of a symmetric encryption algorithm?
   A) RSA
   B) AES
   C) Diffie-Hellman
   D) DSA

8. What does SSL stand for?
   A) Secure Socket Layer
   B) Simple Security Layer
   C) Standard Security Layer
   D) Secure System Layer

11. Which of the following is NOT a feature of hashing?
   A) Deterministic
   B) Fixed size output
   C) Reversible
   D) Fast

12. Which Node.js function is used to create a hash?
   A) crypto.createCipher
   B) crypto.createHash
   C) crypto.generateKeyPair
   D) crypto.createSign

Practical assessment

Tela Tech ltd is a company that develop websites for different institutions they have tasked you to implement symmetric and asymmetric encryption using the Node.js ‘crypto’ module, employing algorithms like AES and RSA. Additionally, you will explore hashing techniques for securing passwords and verifying data integrity. You will also integrate third-party libraries such as ‘bcrypt’ for password hashing and ‘jsonwebtoken’ for secure data transmission using JWT. The exam will further involve securing RESTful APIs in Node.js, focusing on using encryption methods to protect data during transmission. In addition, you will demonstrate your understanding of integrating and managing Node.js third-party libraries, handling asynchronous operations with callbacks, promises, and async/await. Ensuring the security of dependencies by monitoring and updating them using npm, Manage and load environment variables in Node.js applications using dotenv, monitor changes to environment variables and detecting any suspicious activity, implement logging and auditing features to detect unauthorized access to environment variables.

END

References

Books

Doglio, F. (2018). Rest API Development with Node.Js. Uruguay: Apress.

Mardan, A. (2018). Practical Node.js. California: Apress.

Mike Cantelon, M. H. (2014). Node.js IN ACTION. Shelter Island, NY11964: Manning Shelter Island.

Web site links

javatpoint. (2024). nodejs-tutorial. Retrieved from javatpoint: https://www.javatpoint.com/nodejs-tutorial.

Nael, D. (2019, March 11). Retrieved from okta: https://developer.okta.com/blog/2019/03/11/node-sql-server

Palmer, S. (2024). how-to-build-a-secure-web-application-with-nodejs. Retrieved from devteam: https://www.devteam.spsace/blog/how-to-build-a-secure-web-application-with-nodejs/

Syed, B. A. (2014). Beginning Node.js. New york: Apress.

W3schools. (2024). nodejs_get_started.asp. Retrieved from w3schools: https://www.w3schools.com/nodejs/nodejs_get_started.asp

Watmore, J. (2022, January 01). nodejs-ms-sql-server-simple-api-for-authentication-registration-and-user-management. Retrieved from jasonwatmore: https://jasonwatmore.com/post/2022/07/01/nodejs-ms-sql-server-simple-api-for-authentication-registration-and-user-management.

Learning Outcome 3: Test Backend Application

Indicative contents

1.1 Implementation of Unit testing

1.2 Implementation of Usability testing

1.3 Implementation of Security Testing

Key Competencies for Learning Outcome 3: Test Backend Application

Knowledge	Skills	Attitudes
● Description of Usability tests. ● Description of Secure Coding Practices in Node.js. ● Description of Testing Techniques for Node.js Security. ● Identification of Best Practices for Node.js Security Testing.	● Using Mocha Testing Framework ● Using the Chai assertion library ● Monitoring Test results ● Using Postman Testing Tool ● Using Puppeteer Testing Tool ● Implementing Security Testing in Nodejs ● Applying Penetration Testing steps ● Performing penetration Testing using OWASP	● Having teamwork ability ● Being a critical thinker ● Being Innovative ● Being creative ● Practical oriented ● Detail oriented ● Be honesty ● Passion for Learning ● Problem-Solving Mindset ● Collaboration and Communication ● Attention to Security ● Ethical Coding

Duration:20 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Use correctly Mocha Testing Framework as tool for testing backend
2. Use correctly Chai assertion library as tool for testing backend
3. Monitor properly Test results as applied in testing backend
4. Use correctly Postman Testing Tool as tool for testing backend
5. Use correctly Puppeteer Testing Tool as tool for testing backend
6. Implement clearly Security Testing in Nodejs as applied In software development
7. Apply correctly Penetration Testing steps as applied in software testing
8. Perform clearly penetration Testing using OWASP as applied in backend testing
9. Describe correctly Usability tests as step in backend testing
10. Describe correctly Secure Coding Practices in Node.js as applied in testing the backend
11. Describe correctly Testing Techniques for Node.js Security as applied in backend testing
12. Identify clearly Best Practices for Node.js Security Testing as applied in backend

Resources

Equipment	Tools	Materials
● Computer	● Browser ● Node.Js ● Text Editor ● Express. Js ● Postman ● Mocha	● Internet

Indicative content 3.1: Implementation of Unit Testing

Duration: 5 hrs

Theoretical Activity 3.1.1: Introduction to unit tests

Tasks:

1: You are requested to answer the following questions related to introduction to unit testing:

i. Define the term unit testing

ii. Discuss on importance of Unit Testing in Node.js

iii. Explain the unit testing process in Node.js

iv. Discuss unit Testing Tools and Frameworks for Node.js

v. Explain unit testing Libraries for Node.js

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 3.1.1. In addition, ask questions where necessary.

Key readings 3.1.1: Introduction to unit tests

1. **Definition**

   In Node.js, unit tests are used to test individual pieces of code, such as functions or modules, to ensure they work as intended.

   This is crucial for maintaining the reliability of your application as it grows.

2. **Importance of Unit Testing in Node.js**

   - ✓ **Early Bug Detection:** Catch bugs early in the development cycle.
   - ✓ **Code Quality:** Promote better coding practices and cleaner code.
   - ✓ **Documentation:** Serve as living documentation for how the code is supposed to work.
   - ✓ **Refactoring Safety:** Ensure that changes don't break existing functionality.

3. **Unit Testing Process in Node.js**

   - ✓ **Identify Test Cases:** Determine the different scenarios and edge cases that need to be tested.

- ✓ **Write Test Cases:** Write the actual test code for these scenarios.
- ✓ **Run Tests:** Execute the tests to see if they pass or fail.
- ✓ **Fix Issues**: If tests fail, debug and fix the issues in the code.
- ✓ **Repeat:** Continuously run tests as new code is added or existing code is modified.

4. Unit Testing Tools for Node.js

Several tools can help you write and run unit tests in Node.js, including:

- ✓ **Mocha:** A feature-rich JavaScript test framework running on Node.js and in the browser.
- ✓ **Jest**: A delightful JavaScript Testing Framework with a focus on simplicity.
- ✓ **AVA:** A test runner for Node.js with a concise API, detailed error output, and process isolation.
- ✓ **Tape:** A tap-producing test harness for Node.js and browsers.
- ✓ **NYC:** it is a code coverage tool commonly used with testing libraries in Node.js. It works in conjunction with test runners like Mocha, Jest, or others to measure how much of your code is covered by your tests.

  nyc is used to gather coverage information while running tests. It provides insights into which parts of your code have been tested and which haven't. This is especially useful for identifying untested parts of the codebase and improving overall test coverage.

  How it Works: When you run your tests with nyc, it tracks which lines of your code are executed during the test runs and then generates a report detailing which lines, branches, and functions are covered.

  Reporting: It can generate reports in various formats, including text, HTML, and JSON, which can be integrated into CI pipelines for visibility.

Example of Using nyc with Mocha:

- ✓ **Install nyc and Mocha**
- ✓ **npm install --save-dev nyc mocha**
- ✓ **Run tests with nyc to collect coverage**
- ✓ **nyc mocha**
- ✓ **After running, you will see a coverage report showing the percentage of lines, functions, and branches covered by your tests.**
- ✓ **Unit Testing Frameworks for Node.js**

Frameworks provide the structure and guidelines for writing and running tests. **Some popular unit testing frameworks for Node.js include:**

- ✓ **Mocha: Known for its flexibility and extensive feature set.**
- ✓ **Jest: Provides a complete and easy-to-set-up JavaScript testing solution.**
- ✓ **AVA: Known for its simplicity and minimalism.**
- ✓ **Jasmine: A behavior-driven development framework for testing**

JavaScript code.

5. **Unit Testing Libraries for Node.js**

Libraries offer additional functionalities that can be used in conjunction with frameworks to make testing easier and more comprehensive.

Examples include:

- ✓ **Chai: An assertion library that works with Mocha, offering a variety of assertion styles.**
- ✓ **Sinon.js: Provides standalone test spies, stubs, and mocks.**
- ✓ **Proxyquire: Allows overriding dependencies during testing.**
- ✓ **Supertest: For testing HTTP servers.**

Practical Activity 3.1.2: Using Mocha Testing Framework

Task:

1. Read key reading 3.1.2.

2. Referring to the previous theoretical activities (3.1.1) you are requested to go to the computer lab to use Mocha Testing Framework.

3. Apply safety precautions

4. Present out the steps to use Mocha Testing Framework.

5. Referring to the steps provided on task 4, use Mocha Testing Framework.

Key readings 3.1.2: Using Mocha Testing Framework

Mocha is a popular JavaScript testing framework that runs on Node.js and in the browser, allowing you to write unit tests in a simple and flexible manner. Here's how you can get started with Mocha:

1. Installation and Configuration

a. Installation

To install Mocha, you'll need Node.js and npm (Node Package Manager) installed on your system.

Initialize your project (if you haven't already):

```
npm init -y
```

Install Mocha as a development dependency:

```
npm install mocha --save-dev or npm install --global mocha
```

Optionally, install other testing utilities like chai for assertions:

```
npm install chai --save-dev
```

b. Configuration

You can configure Mocha to recognize where your test files are located.

Create a test folder in the root of your project:

```
mkdir test
```

By default, Mocha looks for test files in the test folder. Test files should have the .test.js or .spec.js suffix, though this can be customized.

Update your package.json to add a test script:

```
"scripts": {
  "test": "mocha"
}
```

2. Writing Unit Tests

Writing tests with Mocha involves creating a test file and writing test cases using describe and it blocks.

Example:

Let’s say you have a function add in a file math.js:

```
// math.js
function add(a, b) {
  return a + b;
}
module.exports = add;
```

You can write a test for this function in test/math.test.js:

```
// test/math.test.js
const add = require('../math');
const { expect } = require('chai');
```

```
describe('Math Functions', function () {
it('should return 5 when adding 2 and 3', function () {
const result = add(2, 3);
expect(result).to.equal(5);
});
it('should return 0 when adding -2 and 2', function () {
const result = add(-2, 2);
expect(result).to.equal(0);
});
});
```

3. Running Tests

To run your tests, you can use the `mocha` command in your terminal. Make sure you are in your project's root directory, then run:

```
mocha
```

Mocha will automatically look for test files in the `test` directory and execute them.

Alternatively, you can add a script to your `package.json` to make it easier to run your tests. Open your `package.json` file and add the following under `"scripts"`:

```
"scripts": {
"test": "mocha"
}
```

Now, you can run your tests using the following command:

```
npm test
```

Mocha will execute all the tests found in the test directory and provide a summary of the results.

Additional Features in Mocha

Asynchronous Testing: Mocha supports asynchronous tests by passing a done callback or by returning a promise.

Hooks: Mocha provides hooks (before, after, beforeEach, afterEach) to run code before and after tests.

Reporters: Mocha offers various reporters (e.g., spec, dot, nyan) to customize how test results are displayed.

Example of Asynchronous Testing:

```
// asyncExample.test.js
const { expect } = require('chai');
function fetchData(callback) {
setTimeout(() => {
callback('data');
}, 1000);
}
describe('Async Function', function () {
it('should fetch data correctly', function (done) {
fetchData((data) => {
expect(data).to.equal('data');
done();
});
});
});
```

Practical Activity 3.1.3: Using Chai assertion library

Task:

1. Read key reading 3.1.3

2. As a software developer, you are asked to go the computer lab to use Chai assertion library.

3. Apply safety precautions

4. Present out the steps to use Chai assertion library.

5. Referring to the steps provided on task 4, use Chai assertion library.

Key readings 3.1.3: Using Chai assertion library

Chai is a popular assertion library that works well with Mocha to write unit tests in Node.js. It provides three different interfaces: `assert`, `expect`, and `should`. Let's go through the steps of using Chai in your unit tests.

1. Installation and Configuration

First, you'll need to install Chai. You can do this using npm. Open your terminal and run:

```
npm install --save-dev chai
```

2. Writing Assertions

Chai provides three interfaces for writing assertions: `assert`, `expect`, and `should`. You can choose the one that you are most comfortable with.

Using `assert` Interface

The `assert` interface provides a set of assertion functions for verifying different conditions.

Example:

```
const assert = require('chai').assert;

describe('Array', function() {

describe('indexOf()', function() {

it('should return -1 when the value is not present', function() {

assert.strictEqual([1, 2, 3].indexOf(4), -1);

});

it('should return the index when the value is present', function() {

assert.strictEqual([1, 2, 3].indexOf(2), 1);

});

});

});
```

Using `expect` Interface

The `expect` interface provides a more expressive way to write assertions.

Example:

```
const expect = require('chai').expect;

describe('Array', function() {

describe('indexOf()', function() {

it('should return -1 when the value is not present', function() {

expect([1, 2, 3].indexOf(4)).to.equal(-1);

});

it('should return the index when the value is present', function() {

expect([1, 2, 3].indexOf(2)).to.equal(1);

});

});

});
```

Using `should` Interface

The `should` interface extends native prototypes, making it easier to write assertions in a natural language style.

Example:

```
const should = require('chai').should();

describe('Array', function() {

describe('indexOf()', function() {

it('should return -1 when the value is not present', function() {

[1, 2, 3].indexOf(4).should.equal(-1);

});

it('should return the index when the value is present', function() {

[1, 2, 3].indexOf(2).should.equal(1);

});

});
});
```

3. Chai `expect` and `should` APIs

`expect` API

The `expect` API is designed to be expressive and readable.

Some common assertions you can make with `expect`:

Checking equality:

```
expect(2 + 2).to.equal(4);
```

Checking types:

```
expect('hello').to.be.a('string');
```

Checking properties:

```
expect({ foo: 'bar' }).to.have.property('foo');
```

Checking arrays:

```
expect([1, 2, 3]).to.include(2);
```

`should` API

The `should` API is similar to the `expect` API but uses a different syntax. Here are some common assertions you can make with `should`:

Checking equality:

```
(2 + 2).should.equal(4);
```

Checking types:

```
'hello'.should.be.a('string');
```

Checking properties:

```
 ({ foo: 'bar' }).should.have.property('foo');
```

Checking arrays:

```
 [1, 2, 3].should.include(2);
```

Running Tests

To run your tests, you can use Mocha as described earlier. Ensure you have a test script in your `package.json`:

```
"scripts": {
```

```
"test": "mocha"

}
```

Then, run your tests using:

```
npm test
```

Practical Activity 3.1.4: Monitoring Test results

Task:

1. Read key reading 3.1.4.

2. As a software developer, you are asked to go the computer lab to monitor test results of practical activity 3.1.2.

3. Apply safety precautions

4. Present out the steps to monitor test results.

5. Referring to the steps provided on task 4, monitor test results.

6. Present your work to the trainer and whole class.

Key readings 3.1.4: Monitoring Test results

Monitoring test results after unit testing in Node.js is crucial to ensure that your code behaves as expected. Here are the steps you can follow to monitor and review test results effectively:

1. Running Tests Locally

After writing your unit tests with Mocha and Chai, you can run them locally to see the results immediately.

a. Command Line Execution

Run the tests using the following command:

npm test

Mocha will execute all the tests in the test directory and output the results to the console. **The output will include:**

Number of Tests: The total number of tests executed.

Passed Tests: A list of tests that passed.

Failed Tests: A list of tests that failed, along with details of the failure.

Time Taken: The time it took to run the tests.

b. Reading the Output

Carefully review the console output:

Green Messages: Indicate that a test passed.

Red Messages: Indicate that a test failed, along with an error message and stack trace.

Summary: At the end of the test run, you'll see a summary showing the total number of tests, how many passed, and how many failed.

2. Generating Test Reports

For a more detailed review, especially in continuous integration (CI) environments, you can generate test reports.

a. Install Reporters

Mocha supports various reporters to format and export test results. A commonly used reporter for CI is mocha-junit-reporter.

Install the reporter:

```
npm install mocha-junit-reporter --save-dev
```

Configure Mocha to use the reporter:

Update your package. json or run Mocha with the reporter:

```
mocha --reporter mocha-junit-reporter
```

Review the Report:

The reporter generates a JUnit-style XML file that can be used with CI servers like Jenkins, CircleCI, or Travis CI for further analysis.

b. Custom Reporters

Other popular reporters include:

Spec: A hierarchical spec list:

```
mocha --reporter spec
```

Dot: Minimal output, useful for large test suites:

```
mocha --reporter dot
```

HTML: Generate an HTML file for a visual overview:

```
npm install mochawesome --save-dev
```

```
mocha --reporter mochawesome
```

3. Using Continuous Integration (CI) Tools

Integrate your test suite with a CI/CD pipeline to automatically run tests whenever code is pushed or merged.

a. Set Up a CI Pipeline

Most CI tools (e.g., Jenkins, GitHub Actions, GitLab CI, CircleCI) can be configured to:

Run Tests: Automatically execute your test suite on every commit or pull request.

Monitor Test Results: Display test results directly in the CI dashboard, showing which tests passed or failed.

Generate and Archive Reports: Store test results for later review and analysis.

b. Configure Notifications

Set up notifications to alert you (via email, Slack, etc.) when tests fail. This ensures that you can quickly address any issues.

4. Analyzing Test Coverage

Test coverage tools help you understand how much of your code is being tested.

a. Install a Coverage Tool

Use a tool like nyc (Istanbul) to measure code coverage:

```
npm install nyc --save-dev
```

b. Run Tests with Coverage

Execute your tests with coverage:

```
nyc npm test
```

This will output a coverage report showing the percentage of your codebase that is covered by tests, helping you identify untested code.

c. Review Coverage Reports

Coverage reports include detailed information on which lines of code were executed during the tests and which were missed. The report is typically available in the coverage/ directory.

5. Automated Monitoring with Dashboards

For ongoing monitoring, you can use tools like SonarQube or Coveralls to continuously track test results and code quality.

a. Integrate with SonarQube

Set up SonarQube on your CI server.

Configure your project to send test results and coverage reports to SonarQube.

b. Review Dashboards

SonarQube provides a dashboard with real-time metrics on test success rates, code coverage, and code quality, allowing you to monitor trends and catch issues early.

Points to Remember

- Unit testing in Node.js involves testing individual components or functions of an application to ensure they work as expected. It is crucial for identifying bugs early, improving code quality, and ensuring reliable, maintainable applications.
- The process typically includes writing tests for each unit, running them, and verifying the outcomes.
- Popular tools and frameworks for unit testing in Node.js include Mocha, Jest, and Jasmine, while libraries like Chai and Sinon provide utilities for assertions and mocking during tests.
- **While using the mocha testing framework you follow the steps:**

  1. Installation and Configuration

  2. Writing Unit Tests

  3. Running Tests
- Using Chai assertion library, you follow the following steps:
  1. Installation and Configuration
  2. Writing Assertions
  3. Chai `expect` and `should` APIs
- **While monitoring test results you can perform the following steps:**
  1. Running Tests Locally

2. Generating Test Reports
3. Using Continuous Integration (CI) Tools
4. Analysing Test Coverage
5. Automated Monitoring with Dashboards

Application of learning 3.1.

TelaTech ltd is a company that develop websites for different institutions they have provided you the developed database (KigaliinnovationDB) and inside created a table called clients with the following fields ID, Names, Sex, Address, Phone and Email in node js and developed APIs that will be used to insert, update, select and delete client’s records via frontend where you will be provide the developed APIs to be integrated with front end.

They have included http status codes in order to overcome error handling issues.

You have been tasked for:

- ✓ Using Mocha Testing Framework for unit testing
- ✓ Monitoring Test results

Indicative content 3.2: Implementation of Usability Testing

Duration: 8 hrs

Theoretical Activity 3.2.1: Introduction to Usability tests

Tasks:

1: In small groups, you are requested to answer the following questions related to the introduction to usability testing:

i. Define the term Usability testing

ii. Discuss on importance of Usability Testing

iii. Explain the usability testing process

iv. Identify usability Testing tools

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 3.2.1. In addition, ask questions where necessary.

Key readings 3.2.1.: Introduction to Usability tests

1. **Definition**

   Usability testing is a technique used to evaluate a product or service by testing it with real users.

   The goal is to observe how easily and effectively users can interact with the product, identify any usability issues, and gather qualitative and quantitative data to improve the overall user experience.

2. **Importance of Usability Testing**

   User-Centered Design: Usability testing ensures that the product is designed with the end user in mind, leading to a more intuitive and satisfying user experience.

   Identifying Issues Early: By testing with actual users, you can identify usability issues early in the development process, which can save time and money on costly redesigns later.

   Improving User Satisfaction: Products that are easy to use and meet user needs are more likely to succeed in the market. Usability testing helps improve user satisfaction by refining the product based on user feedback.

Validating Design Decisions: Usability testing provides evidence-based insights that validate design decisions, ensuring that the product meets user expectations and business goals.

Competitive Advantage: Products with superior usability stand out in the market, giving companies a competitive edge by delivering a better user experience than their competitors.

3. **Usability Testing Process**

1. **Planning:**

   Define Objectives: Determine the goals of the usability test. What specific aspects of the product are you evaluating?

   Select Participants: Choose a representative sample of users who reflect your target audience.

   Choose a Method: Decide on the type of usability test (e.g., moderated vs. unmoderated, in-person vs. remote).

2. **Designing the Test:**

   Create Scenarios: Develop realistic scenarios that users will perform during the test. These should be based on common tasks the product is designed to support.

   Prepare Materials: This includes any scripts for moderators, instructions for participants, and tools for recording data.

3. **Conducting the Test:**

   Facilitate the Test: Guide participants through the test scenarios, either in person or remotely. Observe their interactions without intervening too much.

   Collect Data: Record qualitative data (e.g., user comments, behavior) and quantitative data (e.g., task completion time, error rates).

4. **Analyzing Results:**

   Identify Patterns: Look for common issues or patterns in user behavior. What tasks were challenging? Where did users struggle?

   Prioritize Issues: Based on severity and impact, prioritize the usability issues that need to be addressed.

5. **Reporting and Recommendations**

   Document Findings: Create a report summarizing the test results, including key findings, issues identified, and recommended improvements.

   Implement Changes: Work with the design and development teams to address the identified issues and make necessary changes to the product.

6. **Iterate:**

   Usability testing is an iterative process. After making improvements, conduct additional tests to ensure the changes have effectively resolved the issues.

4. **Usability Testing tools**

Nightwatch.js

Nightwatch.js is an end-to-end testing framework for Node.js that can be used to automate browser-based testing.

While it's primarily focused on functional testing, it can also be adapted for usability testing by simulating user interactions.

Features:

- ✓ Simple syntax for writing tests.
- ✓ Integrates with Selenium WebDriver to automate interactions with the browser.
- ✓ Can be used to verify UI components and user flows.

Puppeteer

Puppeteer is a Node.js library that provides a high-level API to control Chrome or Chromium over the DevTools Protocol. It's often used for automated testing of web pages, including usability testing scenarios.

Features:

- ✓ Headless browser testing.
- ✓ Simulate user interactions like clicks, form submissions, and navigation.
- ✓ Capture screenshots or PDFs to visually inspect the UI.

Cypress

Cypress is a modern, front-end testing tool built for the web. It allows developers to write end-to-end tests and is particularly well-suited for testing user interactions, making it a good choice for usability testing in a Node.js project.

Features:

- ✓ Real-time reloading during test writing
- ✓ Easy-to-use API for interacting with web elements.
- ✓ Time-travel debugging to inspect the state of the application at any point in time.

WebdriverIO

WebdriverIO is a popular testing framework that can be used to automate browser testing. It supports a variety of browsers and platforms, making it a versatile tool for usability testing.

Features:

- ✓ Cross-browser testing.
- ✓ Integration with testing frameworks like Mocha and Jasmine.
- ✓ Can run tests on cloud services like BrowserStack or Sauce Labs.

TestCafe

TestCafe is a Node.js tool for automated end-to-end testing. It’s designed for testing web applications and is known for its simplicity and powerful API.

Features:

- ✓ No need for browser plugins.
- ✓ Runs tests in any browser.
- ✓ Supports asynchronous testing.

Practical Activity 3.2.2: Using Postman Testing Tool

Task:

1. Referring to the previous theoretical activities (3.2.1) you are requested to go to the computer lab to use postman-testing tool.

2. Apply safety precautions

3. Read key reading 3.2.2 and ask clarification where necessary

4. Present out the steps to use postman-testing tool.

5. Referring to the steps provided on task 4, use postman-testing tool.

6. Present your work to the trainer and whole class.

Key readings 3.2.2: Using Postman Testing Tool

Using the Postman testing tool, covering everything from installation to iterating and improving your tests.

1. **Installation of Postman**

   See practical activity 1.1.3

2. **Create a Collection**

   A collection in Postman is a way to organize your API requests into folders. This makes it easier to manage and reuse them for testing.

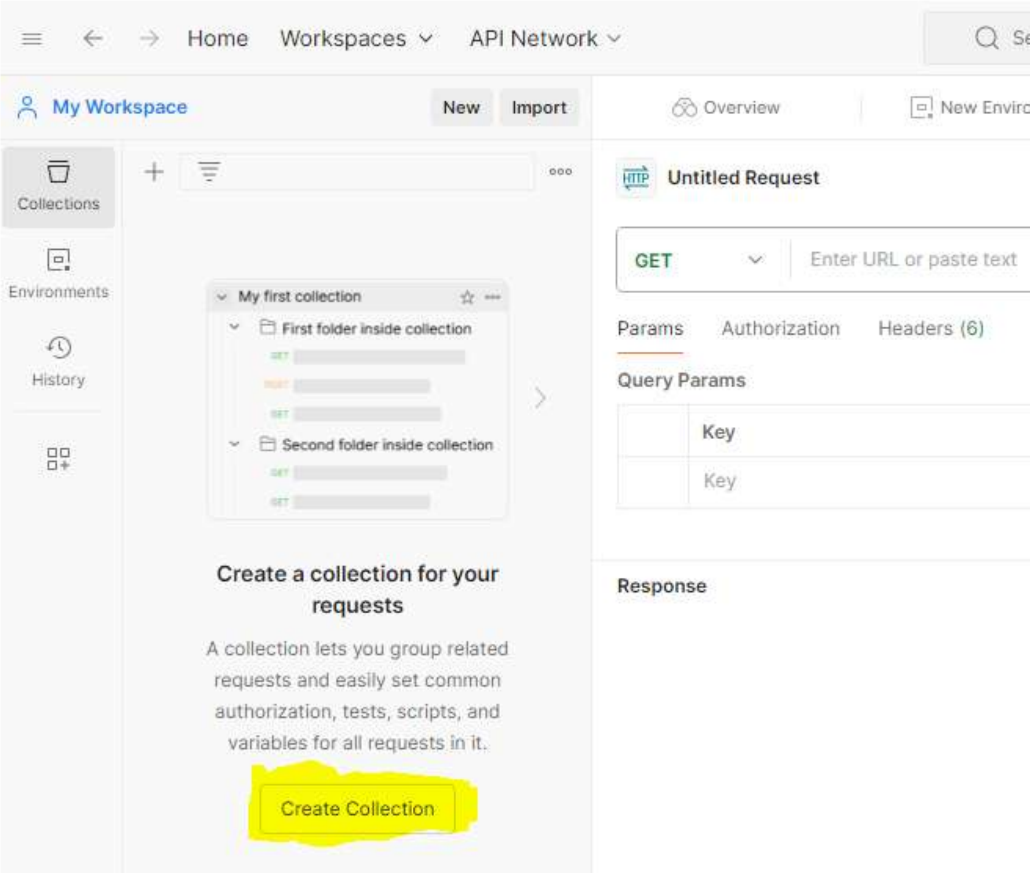
   Steps to Create a Collection

a. **Open Postman:**

   Launch Postman and sign in if you have an account.

b. **Create a New Collection:**

- ✓ Click on the Collections tab on the left sidebar.


Home
Workspaces
API Network
My Workspace
New
Import
Overview
New Enviro
Collections
Environments
History
Untitled Request
GET
Enter URL or paste text
Params
Authorization
Headers (6)
Query Params
Key
Key
Response
My first collection
First folder inside collection
Second folder inside collection
Create a collection for your requests
A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.
Create Collection

- ✓ Click the New Collection button.
- ✓ Name your collection (e.g., "Student API Tests") and add a description if needed.
- ✓ Click Create to finalize the collection.

3. **Define Request**

   A request in Postman is an API call that you define, specifying the endpoint, HTTP method (GET, POST, PUT, DELETE, etc.), headers, parameters, and body data if needed.

   Steps to Define a Request:

a. **Add a Request to a Collection:**

   In your newly created collection, click on Add Request.

   Name your request (e.g., "Get All Students").

b. **Define the Request:**

   Enter the URL: In the URL field, type the endpoint you want to test (e.g., http://localhost:3000/api/students).

   Select the HTTP Method: Choose GET, POST, PUT, DELETE, etc., from the dropdown next to the URL.

Add Parameters (Optional): If your request requires query parameters, click the Params tab and add them.

Set Headers (Optional): Click the Headers tab and add any required headers (e.g., Authorization tokens).

Body Data (Optional for POST/PUT): If you're sending data, click the Body tab and select the appropriate format (e.g., JSON) to input your data.

4. **Write Test Cases**

Test cases in Postman are written in JavaScript and are used to validate the responses of your API requests. You can check status codes, response times, data structures, and more.

Steps to Write Test Cases:

Navigate to the Tests Tab:

Once your request is set up, click on the Tests tab below the request.

Write Your Test Cases:

Use Postman's built-in snippets or write custom JavaScript code. Here are some examples:

```
// Check if the status code is 200
pm.test("Status code is 200", function () {
pm.response.to.have.status(200);
});
// Check if response time is less than 200ms
pm.test("Response time is less than 200ms", function () {
pm.expect(pm.response.responseTime).to.be.below(200);
});
// Validate the structure of the JSON response
pm.test("Response has required fields", function () {
const jsonData = pm.response.json();
pm.expect(jsonData).to.have.property('name');
```

pm.expect(jsonData).to.have.property('age');

});

5. **Run Tests**

How to Run Tests in Postman:

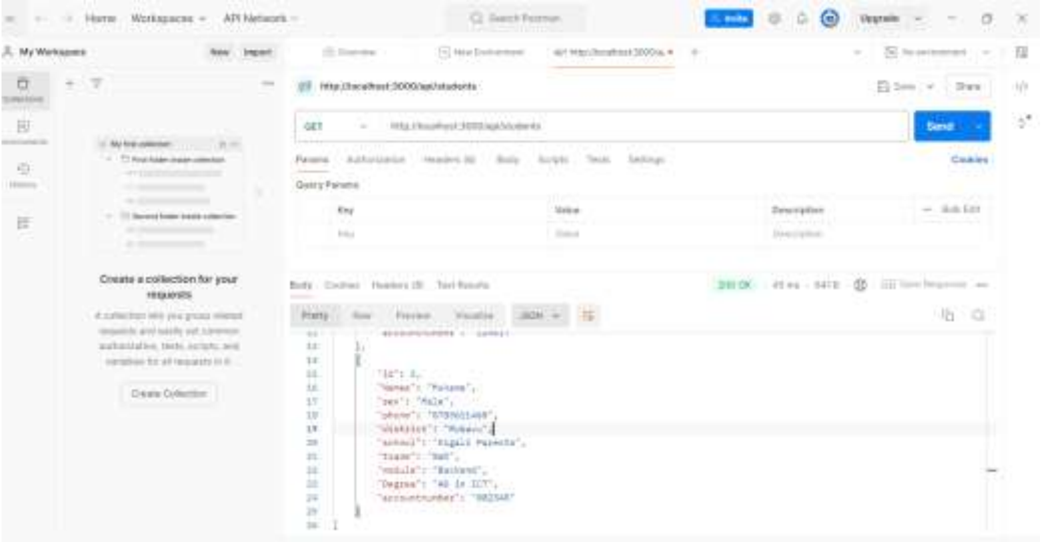
Run Individual Requests:

Click the Send button to run the request. If you have written tests, they will execute automatically, and you can view the results in the Tests tab of the response section.

Run a Collection:

To run all requests in a collection or folder, click the Runner tab on the top left of Postman.

Select your collection and configure the run (e.g., number of iterations, delay between requests).



GET http://localhost:3000/api/students

```
},
{
  "id": 2,
  "names": "Fatuma",
  "sex": "Male",
  "phone": "0789021488",
  "district": "Rubavu",
  "school": "Kigali Parents",
  "trade": "SWD",
  "module": "Backend",
  "Degree": "A0 in ICT",
  "accountnumber": "982548"
}
]
```

Click Start Run to execute the requests and tests.

6. Iterate and Improve

After running your tests, you may need to refine them based on the results, add new test cases, or modify requests to handle new scenarios or edge cases.

Steps to Iterate and Improve:

Analyze Test Results:

Review the test results in Postman. Identify any failures or performance issues.

Modify Test Cases:

Update or add test cases to cover any gaps or handle new edge cases. For example, you might add tests to check for proper error handling.

Enhance Request Definitions:

Adjust your requests to test different scenarios (e.g., different input data, headers, or authentication mechanisms).

Run Tests Again:

Rerun your tests to validate the changes. Ensure that all tests pass, and that the application behaves as expected.

Document and Share:

Use Postman's documentation feature to document your requests, test cases, and results. Share the collection with your team if needed for collaboration.

Practical Activity 3.2.3: Using Puppeteer Testing Tool

Task:

1. Read key reading 3.2.3 and ask clarification where necessary

2. Referring to the previous theoretical activities (3.2.1) you are requested to go to the computer lab to use Puppeteer testing tool.

3. Apply safety precautions

4. Present out the steps to use Puppeteer testing tool.

5. Referring to the steps provided on task 4, use Puppeteer testing tool.

6. Present your work to the trainer and whole class.

Key readings 3.2.3: Using Puppeteer Testing Tool

Using the Puppeteer testing tool, covering everything from installation to generating reports.

1. Installation of Puppeteer

Steps to Install Puppeteer:

Initialize a Node.js Project:

If you don't already have a Node.js project, create one by running:

```
mkdir puppeteer-tests

cd puppeteer-tests

npm init -y
```

Install Puppeteer:

Install Puppeteer using npm. This will download the Puppeteer package along with a version of Chromium.

```
npm install puppeteer --save-dev
```

2. Define Test Scenarios

A test scenario outlines the specific behavior you want to test, such as verifying that a user can successfully log in, navigate to a specific page, or interact with an element on the page.

Steps to Define Test Scenarios:

Identify the Scenarios:

Determine the key interactions and features you need to test. For example, "User logs in and navigates to the dashboard."

Write the Test Script:

Create a JavaScript file (e.g., login-test.js) and define the scenario using Puppeteer's API.

Example: Testing a login scenario.

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

(async () => {

const browser = await puppeteer.launch();

const page = await browser.newPage();

// Go to the login page

await page.goto('http://localhost:3000/login');

// Enter username and password
await page.type('#username', 'testuser');

await page.type('#password', 'password123');

  // Click the login button

  await page.click('#loginButton');

  // Wait for navigation to the dashboard

await page.waitForNavigation();

// Verify the user is on the dashboard

const title = await page.title();

console.log(title); // Expected: 'Dashboard'

await browser.close();

})();
```

3. Automate User Interaction

Automating user interaction involves simulating actions that a user would take, such as clicking buttons, filling out forms, or navigating through pages.

Steps to Automate User Interaction:

Simulate Form Filling:

Use Puppeteer's page.type() method to simulate typing text into input fields.

```
await page.type('#username', 'testuser');

await page.type('#password', 'password123');
```

Simulate Button Clicks:

Use page.click() to simulate clicking buttons or links.

```
await page.click('#loginButton');
```

Handle Navigation:

Use page.waitForNavigation() to ensure the script waits for the page to load after an action like clicking a link.

```
await page.waitForNavigation();
```

Interact with Other Elements:

Puppeteer allows interactions with various elements, like selecting from dropdowns, hovering over elements, or checking checkboxes.

```
await page.select('#dropdown', 'option1');

await page.hover('#hoverElement');

await page.click('#checkbox');
```

4. Measure Page Performance

Measuring page performance involves assessing how fast your web page loads, how long it takes for the user to interact with the page, and other critical performance metrics.

Steps to Measure Page Performance:

Use Puppeteer’s Built-in Performance Metrics:

Puppeteer provides access to performance metrics like First Meaningful Paint and Time to Interactive.

```
const metrics = await page.metrics();

console.log(metrics);
```

Capture a Performance Trace:

You can generate a performance trace and analyze it using Chrome DevTools.

```
await page.tracing.start({ path: 'trace.json' });

await page.goto('http://localhost:3000');

await page.tracing.stop();
```

Measure Time to Load:

Use performance.timing to measure the time it takes for the page to load.

```
const performanceTiming = JSON.parse(await page.evaluate(() => JSON.stringify(window.performance.timing)));

console.log('Load time:', performanceTiming.loadEventEnd - performanceTiming.navigationStart);
```

5. Test Accessibility

Accessibility testing ensures that your web application is usable by people with various disabilities, complying with standards like WCAG.

Steps to Test Accessibility:

Install the Axe-core Library:

Integrate axe-core, a popular accessibility testing library, with Puppeteer.

```
npm install axe-core puppeteer --save-dev
```

Run Accessibility Tests:

Inject and run axe-core in your Puppeteer script to check for accessibility issues.

```
const { AxePuppeteer } = require('axe-puppeteer');

const results = await new AxePuppeteer(page).analyze();

console.log(results.violations); // Log accessibility violations
```

6. Generate Report

Generating a report involves collecting test results, including performance metrics, accessibility issues, and other findings, and presenting them in a readable format.

Steps to Generate a Report:

Using Console Logs:

Start by logging the test results and performance metrics directly in the console.

```
console.log('Performance Metrics:', metrics);

console.log('Accessibility Violations:', results.violations);
```

Exporting to a File:

Save the results to a file (e.g., JSON) for later analysis or to generate more detailed reports.

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

fs.writeFileSync('results.json', JSON.stringify(results, null, 2));
```

Integrate with Reporting Tools:

You can integrate Puppeteer with third-party reporting tools to generate more sophisticated reports. For example, use Allure or Mochawesome for creating HTML reports from your Puppeteer tests.

Automated Reports in CI/CD:

If you’re running tests in a CI/CD pipeline, configure your pipeline to generate and store reports automatically after each test run.

Points to Remember

- Usability testing is a technique used to evaluate a product or service by testing it with real users.
- There are different importance of usability testing including User-Centered Design, Identifying Issues Early, Improving User Satisfaction, Validating Design Decisions, and Competitive Advantage
- Once performing Usability testing you follow different Processes starting from Planning to reporting and Recommendations and you have to use testing tools like Nightwatch.js, Puppeteer, Cypress, WebdriverIO, and TestCafe.
- Once testing usability for the developed project in node js using postman you pass through the following steps:
    1. Installation of Postman
    2. Create a Collection
    3. Define Request
    4. Write Test Cases
    5. Run Tests
    6. Iterate and Improve
- Using the Puppeteer testing tool, covering everything from installation to generating reports you follow the following steps:
    1. Installation of Puppeteer
    2. Define Test Scenarios
    3. Automate User Interaction
    4. Measure Page Performance
    5. Test Accessibility
    6. Generate Report

Application of learning 3.2.

TelaTech ltd is a company that develop websites for different institutions they have provided you the developed database (KigaliinnovationDB) and inside created a table called clients with the following fields ID, Names, Sex, Address, Phone and Email in node js and developed APIs that will be used to insert, update, select and delete client's records via frontend where you will be provide the developed APIs to be integrated with front end.

They have included http status codes in order to overcome error handling issues.

You have been tasked for:

- ✓ Use Postman testing tool for usability testing on APIs as provided.
- ✓ Use Puppeteer testing tool for usability testing.

Indicative content 3.3: Implementation of Security Testing

Duration: hrs

Theoretical Activity 3.3.1: Introduction Node.js Security testing

Tasks:

1: You are requested to describe key security issues in software development.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.3.1. ask questions where necessary.

Key readings 3.3.1.: Introduction Node.js Security testing

Node.js is a popular runtime environment for building scalable network applications. However, with its popularity, it becomes a target for various security threats.

Here's an introduction to key security issues in Node.js:

1. Injection Attacks

Description:

Injection attacks occur when an attacker is able to insert or "inject" malicious code into a program, typically through input fields or query parameters. The most common type is SQL Injection, where malicious SQL code is inserted into an SQL query.

Impact:

- ✓ Data theft or corruption.
- ✓ Unauthorized access to data.
- ✓ Compromise of the entire system.

Mitigation:

- ✓ Use parameterized queries or prepared statements.
- ✓ Validate and sanitize all user inputs.
- ✓ Employ Object-Relational Mapping (ORM) libraries to interact with the database.

2. Broken Authentication and Session Management

Description:

This occurs when authentication mechanisms are poorly implemented, leading to vulnerabilities in login procedures or session handling. Attackers can exploit these weaknesses to impersonate users.

Impact:

- ✓ Unauthorized access to sensitive data.
- ✓ Session hijacking.
- ✓ User impersonation.

Mitigation:

- ✓ Implement strong password policies
- ✓ Use multi-factor authentication (MFA).
- ✓ Secure session cookies (HTTPOnly, Secure flags).
- ✓ Rotate session IDs after login.

3. Cross-Site Scripting (XSS)

Description:

XSS is an attack where an attacker injects malicious scripts into webpages viewed by other users. These scripts can then execute in the context of the user's browser, potentially leading to session hijacking, defacement, or phishing.

Impact:

- ✓ Execution of unauthorized JavaScript in the user's browser
- ✓ Theft of session tokens or cookies.
- ✓ Defacement or redirection of users to malicious sites.

Mitigation:

- ✓ Sanitize and encode user input.
- ✓ Use Content Security Policy (CSP) headers.
- ✓ Escape data in HTML contexts.

4. Cross-Site Request Forgery (CSRF)

Description:

CSRF is an attack that tricks a user into performing an action on a web application without their consent, by exploiting the user's authenticated session with the application.

Impact:

- ✓ Unauthorized actions performed on behalf of the user.
- ✓ Data alteration or deletion.
- ✓ Fraudulent transactions.

Mitigation:

- ✓ Use Anti-CSRF tokens.
- ✓ Ensure that state-changing requests are performed via POST requests.
- ✓ Check the Referer or Origin headers.

5. Security Misconfiguration

Description:

This vulnerability arises when security settings are not defined, implemented, or maintained, leaving the application, servers, or network vulnerable to attacks.

Impact:

- ✓ Unauthorized access to configuration files or administrative functions.
- ✓ Exposure of sensitive data.
- ✓ Compromise of the entire system.

Mitigation:

- ✓ Regularly review and update configurations.
- ✓ Disable unnecessary features, modules, or services
- ✓ Enforce least privilege principles.

6. Insecure Cryptographic Storage

Description

This issue arises when sensitive data, such as passwords, credit card numbers, or personal information, is not securely encrypted or is improperly encrypted.

Impact

- ✓ Data theft or exposure
- ✓ Identity theft or financial loss
- ✓ Legal and compliance violations.

Mitigation

- ✓ Use strong encryption algorithms (e.g., AES-256) for sensitive data
- ✓ Securely manage and store cryptographic keys
- ✓ Avoid storing sensitive data unless absolutely necessary.

7. Insufficient Authorization

Description

Insufficient authorization occurs when users can access resources or perform actions they are not authorized to. This often results from improper role management or access control implementation.

Impact:

- ✓ Unauthorized access to sensitive data or functions.
- ✓ Data manipulation or deletion.
- ✓ Compromise of the application's integrity.

Mitigation

- ✓ Implement Role-Based Access Control (RBAC).
- ✓ Enforce the principle of least privilege.
- ✓ Regularly audit and review access controls.

8. Insufficient Logging and Monitoring

Description:

Without proper logging and monitoring, malicious activities can go undetected. Insufficient logging can prevent timely detection of attacks, while poor monitoring can lead to delayed responses.

Impact

- ✓ Delayed detection of security breaches.
- ✓ Inability to trace the attacker's actions.
- ✓ Increased damage from undetected threats.

Mitigation

- ✓ Implement comprehensive logging for all critical actions.
- ✓ Regularly monitor logs for suspicious activity.
- ✓ Set up alerts for anomalous behavior.
- ✓ Ensure logs are stored securely and are tamper-proof.

Theoretical Activity 3.3.2: Identification of Tools for Security Testing in Node.js

Tasks:

1: You are requested to identify tools for security testing in node js.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.3.2. ask questions where necessary.

Key readings 3.3.2: Identification of Tools for Security Testing in Node.js

Security testing in Node.js involves using various tools to identify vulnerabilities and ensure that your application is secure. Here’s an overview of different types of security testing tools and frameworks

1. Overview of Security Testing Tools

Security Testing Tools are designed to analyze software applications for vulnerabilities, weaknesses, and compliance with security standards. These tools can be classified into several categories

- ✓ **Static Analysis Tools:** Analyze the source code or binaries of an application without executing it. They are used to detect issues such as code vulnerabilities, insecure coding practices, and other security flaws.
- ✓ **Dynamic Analysis Tools:** Test an application while it is running to identify runtime vulnerabilities, including issues that may only surface during execution, such as configuration problems or security flaws that arise from user interactions.

- ✓ **Testing Frameworks:** Frameworks provide a structure for writing and executing tests, including security-focused tests. They often integrate with other security tools to automate testing processes.

2. Static Analysis Tools

Static Analysis Tools examine code for security vulnerabilities without running the program. They are helpful in identifying issues early in the development process.

a. ESLint (with Security Plugins):

Description: A popular linting tool that can be extended with security-focused plugins (e.g., eslint-plugin-security) to identify potential security issues in JavaScript code.

Installation:

```
npm install eslint eslint-plugin-security --save-dev
```

Configuration: Add eslint-plugin-security to your .eslintrc configuration file.

b. SonarQube:

Description: An open-source platform for continuous inspection of code quality that includes security vulnerability detection.

Installation: Requires setting up a SonarQube server and integrating it with your build process.

Configuration: Integrate with Node.js projects using the SonarQube scanner.

c. Snyk:

Description: A tool for finding and fixing vulnerabilities in your dependencies and code.

Installation:

```
npm install -g snyk
```

Usage:

```
snyk test
```

3. Dynamic Analysis Tools

Dynamic Analysis Tools test the application during runtime, focusing on how the application behaves under different conditions and user inputs.

a. OWASP ZAP (Zed Attack Proxy):

Description: An open-source security scanner that helps find vulnerabilities in web applications during runtime.

Installation: Download from the OWASP ZAP website.

Usage: Set up ZAP as a proxy for your application and perform automated scans.

b. Burp Suite:

Description: A comprehensive web application security testing tool that provides features for scanning, crawling, and analyzing web applications.

Installation: Download from the Burp Suite website.

Usage: Configure your browser to use Burp Suite as a proxy and perform scans and manual testing.

c. Arachni

Description: A feature-rich, modular, high-performance security scanner for web applications.

Installation: Follow instructions on the Arachni website.

Usage: Run scans using the command-line interface or web interface.

4. Testing Frameworks

Testing Frameworks provide the infrastructure for writing and executing automated tests, including security tests.

a. Open Worldwide Application Security Project (OWASP):

Description: Provides a variety of resources and tools for application security testing, including guidelines and checklists for secure development practices.

Usage: Utilize OWASP resources and tools like OWASP Dependency-Check and OWASP Dependency-Track for security analysis.

b. Mocha:

Description: A widely-used JavaScript test framework that supports various types of tests, including integration and unit tests.

Installation:

```
npm install mocha --save-dev
```

Usage: Write test scripts to include security-related tests and run tests with:

```
npx mocha
```

c. Chai:

Description: An assertion library often used with Mocha for writing tests, providing various assertion styles.

Installation:

```
npm install chai --save-dev
```

Usage: Use Chai to write assertions in your Mocha tests to verify security aspects.

d. Supertest:

Description: A popular testing library for HTTP assertions, useful for testing API endpoints.

Installation:

```
npm install supertest --save-dev
```

Usage: Integrate with Mocha and Chai to test API security.

Theoretical Activity 3.3.3: Description of Secure Coding Practices in Node.js

Tasks:

1: You are requested to describe secure coding practices in node js.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.3.3. ask questions where necessary.

Key readings 3.3.3: Description of Secure Coding Practices in Node.js

Secure coding practices are essential to building robust and secure Node.js applications. These practices help prevent common vulnerabilities and ensure that your application is resilient against attacks.

Below is a description of key secure coding practices in Node.js:

1. Input Validation and Sanitization

Always validate and sanitize user inputs to prevent injection attacks, such as SQL injection, command injection, and cross-site scripting (XSS).

Practices:

- ✓ Use libraries like validator to sanitize inputs.
- ✓ Implement whitelisting to accept only expected input formats.
- ✓ Avoid relying solely on client-side validation; ensure server-side validation is enforced.

2. Use of Parameterized Queries

To protect against SQL injection attacks, use parameterized queries or prepared statements when interacting with the database.

Practices:

- ✓ Use ORM libraries like Sequelize or Mongoose that support parameterized queries by default.
- ✓ For raw SQL queries, use placeholders instead of string concatenation.

3. Authentication and Authorization

Implement strong authentication and authorization mechanisms to control access to resources and protect sensitive data.

Practices:

- ✓ Use strong, hashed passwords with a salt using libraries like bcrypt.
- ✓ Implement Role-Based Access Control (RBAC) to enforce authorization rules.
- ✓ Use OAuth2 or JWT for secure authentication and session management.
- ✓ Rotate and invalidate tokens or session IDs after sensitive operations.

4. **Secure Error Handling**

Avoid exposing detailed error messages to users, as they can reveal sensitive information that could be exploited by attackers.

Practices:

- ✓ Log detailed errors on the server-side but return generic error messages to the client.
- ✓ Implement a global error handler in Express to manage and format error responses securely.

5. Avoid Hardcoding Secrets

Never hardcode secrets, such as API keys, passwords, or cryptographic keys, in your source code.

Practices:

- ✓ Use environment variables to store secrets securely
- ✓ Use tools like dotenv to manage environment variables securely.
- ✓ Consider using secret management services, like AWS Secrets Manager or HashiCorp Vault.

6. Secure Dependencies

Regularly check and update dependencies to ensure that your application does not use vulnerable packages.

Practices:

- ✓ Use tools like npm audit, Snyk, or Dependabot to monitor and fix vulnerabilities in your dependencies
- ✓ Regularly update Node.js and its dependencies to the latest secure versions
- ✓ Avoid using outdated or unmaintained libraries.

7. Implement HTTPS/TLS

Always use HTTPS/TLS to encrypt data in transit, ensuring that communication between clients and servers is secure.

Practices:

- ✓ Use libraries like helmet to automatically configure secure HTTP headers
- ✓ Obtain and renew SSL/TLS certificates using services like Let's Encrypt
- ✓ Enforce HTTPS across your application using HSTS headers.

8. Protect Against Cross-Site Scripting (XSS)

XSS attacks occur when an attacker injects malicious scripts into webpages that are viewed by other users.

Practices:

- ✓ Sanitize user inputs and outputs using libraries like DOMPurify.
- ✓ Use Content Security Policy (CSP) headers to restrict the sources from which scripts can be loaded.
- ✓ Escape output data in templates to prevent execution of injected scripts.

9. Prevent Cross-Site Request Forgery (CSRF)

CSRF attacks trick authenticated users into submitting malicious requests unknowingly.

Practices:

- ✓ Use CSRF tokens to validate the authenticity of requests
- ✓ Implement the SameSite attribute on cookies to prevent them from being sent in cross-origin requests.
- ✓ Use libraries like csurf to easily integrate CSRF protection into Express applications.

10. Secure Session Management

Manage user sessions securely to prevent session hijacking and other session-based attacks.

Practices:

- ✓ Use secure, HTTPOnly, and SameSite attributes on cookies to protect session data.
- ✓ Rotate session identifiers after login or other sensitive actions.
- ✓ Set session expiration and invalidate sessions after a period of inactivity.

11. **Implement Logging and Monitoring**

Keep track of security-related events through logging and monitoring to detect and respond to potential threats in real-time.

Practices:

- ✓ Log security events such as failed login attempts, authorization failures, and unusual activities.
- ✓ Use centralized logging systems like ELK Stack or Graylog
- ✓ Set up alerts for suspicious activities and regularly review logs.

12. Secure File Uploads

Protect against malicious file uploads that could lead to code execution or data breaches.

Practices:

- ✓ Limit the types and sizes of files that can be uploaded
- ✓ Use libraries like multer with file type validation
- ✓ Store uploaded files in directories outside the web root.

13. Use Security Headers

Secure your application by setting appropriate HTTP headers that help prevent common attacks.

Practices:

- ✓ Use helmet to set security-related HTTP headers, such as X-Frame-Options, X-Content-Type-Options, and Strict-Transport-Security.
- ✓ Regularly review and update your security headers based on best practices.

14. Conduct Regular Security Audits

Regularly audit your codebase, infrastructure, and processes to identify and address potential security vulnerabilities.

Practices:

- ✓ Perform code reviews with a focus on security.
- ✓ Conduct penetration testing and vulnerability assessments.
- ✓ Engage in third-party security audits to validate your security posture.

Theoretical Activity 3.3.4: Description of Testing Techniques for Node.js Security

Tasks:

1: You are requested to describe Testing Techniques for Node.js Security.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.3.4. ask questions where necessary.

Key readings 3.3.4: Description of Testing Techniques for Node.js Security

Testing techniques for Node.js security is essential for identifying and mitigating vulnerabilities in your application. These techniques ensure that your Node.js application is robust, secure, and capable of withstanding various attacks.

Below is a description of the key security testing techniques that should be employed during the development and maintenance of Node.js applications:

1. Static Application Security Testing (SAST)

SAST involves analyzing the source code, bytecode, or binaries of an application without executing it. This technique identifies security vulnerabilities early in the development cycle by examining the codebase for insecure coding practices, vulnerabilities, and compliance issues.

Techniques

Code Scanning: Use tools like ESLint (with security plugins), SonarQube, or Snyk to scan the code for vulnerabilities such as insecure input handling, improper use of APIs, and potential injection points.

Dependency Analysis: Regularly audit third-party dependencies using tools like npm audit or Snyk to identify and fix vulnerabilities in libraries and packages.

2. Dynamic Application Security Testing (DAST)

DAST involves testing the application while it is running, simulating attacks on the application to identify runtime vulnerabilities. This technique helps identify issues like misconfigurations, insecure authentication, and potential data leaks.

Techniques

Automated Scanning: Use tools like OWASP ZAP, Burp Suite, or Arachni to perform automated scans of your application. These tools simulate various attacks, such as SQL injection, XSS, and CSRF, to identify vulnerabilities.

Manual Penetration Testing: Conduct manual tests to simulate real-world attacks that may not be covered by automated tools. This includes exploring potential security gaps by testing authentication, session management, and access controls.

3. Interactive Application Security Testing (IAST)

IAST combines elements of both SAST and DAST, analyzing code in real-time as the application is running. It provides contextual information about vulnerabilities and allows developers to identify and fix issues more accurately.

Techniques

Real-Time Analysis: Use IAST tools like Contrast Security or Seeker that integrate with the running application to monitor and analyze security during normal operation.

These tools provide insights into how data flows through the application and highlight potential vulnerabilities in real time.

Integrated Testing: Implement IAST in your CI/CD pipeline to continuously monitor and test the security of your application throughout its lifecycle.

4. Penetration Testing

Penetration testing involves simulating real-world attacks to identify and exploit vulnerabilities in your application.

This method goes beyond automated scanning to assess the overall security posture of your Node.js application.

Techniques

Black Box Testing: Test the application from an attacker's perspective without knowledge of the internal structure. This includes testing external interfaces, APIs, and user inputs.

White Box Testing: Perform tests with full knowledge of the application's internal workings, including the codebase, architecture, and environment. This helps identify hidden vulnerabilities that may not be visible through black-box testing.

Gray Box Testing: Combine both black-box and white-box testing approaches, where the tester has partial knowledge of the internal structure, focusing on specific areas of concern.

5. Fuzz Testing

Fuzz testing (or fuzzing) involves providing the application with random, unexpected, or invalid data inputs to see how it responds. This technique is useful for discovering unexpected vulnerabilities, such as memory leaks, crashes, or unhandled exceptions.

Techniques:

Input Fuzzing: Use fuzz testing tools like Jazzer or Atheris to generate and input random data into various endpoints, forms, and API requests to observe how the application handles unexpected inputs.

Protocol Fuzzing: Test the application’s ability to handle malformed or non-standard network protocol data, potentially revealing vulnerabilities in how it processes incoming traffic.

6. Security Unit Testing

Security unit testing involves writing automated tests specifically designed to check the security aspects of individual components or functions within the application.

Techniques:

Test Cases for Input Validation: Write unit tests using frameworks like Mocha and Chai to ensure that all input fields and API endpoints are properly validated and sanitized.

Test Cases for Authentication and Authorization: Create tests to verify that authentication mechanisms (e.g., JWT token validation) and authorization rules (e.g., role-based access controls) are functioning as expected.

7. Continuous Security Testing

Continuous security testing involves integrating security tests into the CI/CD pipeline to ensure that security checks are performed automatically with every code change.

Techniques:

Automated Security Scans: Configure tools like Snyk, OWASP ZAP, or SonarQube to run automated scans as part of the CI/CD process, ensuring that vulnerabilities are detected and addressed before deployment.

Test Automation Frameworks: Use tools like Jenkins, GitLab CI, or CircleCI to automate the execution of security tests, ensuring that security is continuously monitored and enforced throughout the development lifecycle.

8. Threat Modeling

Threat modeling is the process of identifying, analyzing, and mitigating potential security threats during the design and development phases of the application.

Techniques:

Data Flow Diagrams (DFDs): Create DFDs to visualize how data moves through the application and identify potential threat vectors, such as untrusted inputs or exposed APIs.

STRIDE Framework: Use the STRIDE model (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to systematically identify and address potential security threats in the application.

9. Vulnerability Scanning

Vulnerability scanning involves using automated tools to scan the application for known vulnerabilities, such as outdated dependencies, insecure configurations, or misconfigured security settings.

Techniques:

Dependency Scanning: Regularly scan dependencies using tools like npm audit, Snyk, or Dependabot to identify and update vulnerable packages.

Configuration Scanning: Use tools like lynis or ScoutSuite to scan the server environment and application configuration for common security misconfigurations.

10. Security Regression Testing

Security regression testing ensures that previously identified and fixed security issues do not reappear in future releases of the application.

Techniques:

Regression Test Suite: Maintain a suite of security regression tests that are run automatically after every code change, especially when applying patches or updates.

Automated Testing: Integrate security regression tests into your CI/CD pipeline to ensure continuous protection against reintroduced vulnerabilities.

Theoretical Activity 3.3.5: Description of best Practices for Node.js Security

Tasks:

1: You are requested to describe best Practices for Node.js Security.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.3.5. ask questions where necessary.

Key readings 3.3.5: Description of best Practices for Node.js Security

- **Best Practices for Node.js Security Testing**

1. Integrate Security Testing Early and Often

Shift Left: Incorporate security testing early in the development lifecycle (shift left) to identify and address vulnerabilities before they reach production.

Continuous Testing: Implement security tests as part of your CI/CD pipeline to ensure that every code change is evaluated for security risks.

Automated Testing: Use automated tools for static and dynamic analysis, enabling continuous monitoring and quick identification of security issues.

2. Use a Comprehensive Security Testing Suite

Static Application Security Testing (SAST): Regularly scan your codebase with SAST tools to identify vulnerabilities in the code itself.

Dynamic Application Security Testing (DAST): Test the running application for vulnerabilities such as injection attacks and insecure configurations.

Interactive Application Security Testing (IAST): Combine elements of SAST and DAST to analyze code in real time during execution.

3. Employ Threat Modeling

Identify Threats: Perform threat modeling to identify potential attack vectors and prioritize security testing efforts based on risk.

Design Security: Use the results of threat modeling to inform secure design decisions and guide the development of test cases.

4. Regularly Update and Audit Dependencies

Dependency Management: Regularly audit third-party libraries and dependencies using tools like npm audit or Snyk and apply updates to address known vulnerabilities.

Vulnerability Scanning: Integrate dependency vulnerability scanning into your CI/CD pipeline to ensure dependencies are secure with each build.

5. Perform Regular Penetration Testing

Manual Testing: Complement automated tests with manual penetration testing to uncover complex vulnerabilities that automated tools may miss.

Test for Common Vulnerabilities: Ensure that penetration tests cover common vulnerabilities, such as SQL injection, XSS, CSRF, and authentication flaws.

6. Implement Security Regression Testing

Regression Test Suite: Maintain a suite of security regression tests that are executed regularly, especially after fixing a vulnerability or applying a patch.

Automated Regression: Automate security regression tests to quickly identify reintroduced vulnerabilities.

Security Testing Lifecycle

1. Planning

Define Objectives: Set clear security objectives and requirements based on your application's risk profile and regulatory needs.

Select Tools and Frameworks: Choose appropriate security testing tools and frameworks that align with your security goals.

Develop a Test Plan: Create a detailed test plan that outlines the scope, methodologies, and resources required for security testing.

2. Implementation

Develop Test Cases: Write specific security test cases that address the identified risks and vulnerabilities from the threat model.

Integrate with CI/CD: Implement security tests in the CI/CD pipeline to ensure they are executed automatically with each code change.

Execute Tests: Perform both automated and manual security tests, including SAST, DAST, IAST, and penetration testing.

3. Analysis

Review Results: Analyze the results of security tests to identify vulnerabilities, misconfigurations, and code weaknesses.

Prioritize Issues: Prioritize identified issues based on their severity, impact, and likelihood of exploitation.

Document Findings: Thoroughly document all security issues, including steps to reproduce, affected areas, and potential impact.

4. Remediation

Fix Vulnerabilities: Work with the development team to address and fix identified vulnerabilities.

Re-Test: Re-test the application after remediation to ensure that fixes are effective and do not introduce new issues.

Update Tests: Update security test cases as needed to reflect the changes and ensure ongoing protection against similar vulnerabilities.

5. Monitoring and Maintenance

Continuous Monitoring: Implement continuous monitoring for security incidents, using tools like IDS/IPS and centralized logging systems.

Regular Audits: Conduct regular security audits and vulnerability assessments to ensure ongoing compliance and security.

Update and Patch: Regularly update and patch the application, dependencies, and infrastructure to address new vulnerabilities.

Reporting Security Vulnerabilities

1. Internal Reporting

Incident Response Plan: Establish an internal incident response plan for reporting and addressing security vulnerabilities quickly and efficiently.

Clear Communication Channels: Ensure that there are clear communication channels for developers, security teams, and stakeholders to report and discuss vulnerabilities.

Document Vulnerabilities: Create detailed reports that document each vulnerability, including its severity, potential impact, and recommended remediation steps.

2. External Reporting

Responsible Disclosure: Implement a responsible disclosure policy that encourages external security researchers to report vulnerabilities they discover in a safe and coordinated manner.

Bug Bounty Programs: Consider setting up a bug bounty program to incentivize ethical hackers to find and report vulnerabilities in your application.

Public Advisories: When necessary, issue public advisories to inform users and stakeholders of critical vulnerabilities and the steps being taken to mitigate them.

Remediation and Mitigation

1. Prioritization

Risk-Based Approach: Prioritize remediation efforts based on the severity of the vulnerability, its potential impact, and the likelihood of exploitation.

Patch Critical Issues First: Focus on fixing critical and high-risk vulnerabilities first to minimize the potential for exploitation.

2. Immediate Fixes

Apply Patches: Apply security patches and updates as soon as they are available to address known vulnerabilities.

Hotfixes: For urgent issues, deploy hotfixes to mitigate the risk while a more permanent solution is developed.

3. Long-Term Mitigation

Code Refactoring: Refactor code as needed to address underlying security weaknesses and improve overall code quality.

Security Controls: Implement additional security controls (e.g., input validation, encryption) to mitigate the impact of potential vulnerabilities.

4. Re-Testing

Verification: After remediation, re-test the application to verify that the vulnerability has been effectively addressed and that no new issues have been introduced.

Continuous Improvement: Use the insights gained from remediation to improve security practices and prevent similar issues in the future.

Compliance and Regulations

1. Understanding Relevant Regulations

Identify Applicable Laws: Understand which security regulations and standards apply to your industry and application, such as GDPR, HIPAA, PCI DSS, or SOC 2.

Map Requirements: Map regulatory requirements to specific security controls and testing practices within your Node.js application.

2. Implementing Security Controls

Data Protection: Implement strong data protection measures, including encryption and access controls, to comply with regulations like GDPR.

Audit Trails: Maintain comprehensive audit logs to track access and changes to sensitive data, ensuring compliance with standards like PCI DSS.

3. Documentation and Reporting

Compliance Documentation: Maintain detailed documentation of your security practices, test results, and remediation efforts to demonstrate compliance.

Regulatory Reporting: Prepare for and conduct regular audits, and be ready to report on your security posture and compliance status to regulatory bodies.

4. Continuous Compliance

Regular Audits: Conduct regular security and compliance audits to ensure that your application remains in compliance with relevant regulations.

Stay Informed: Keep up to date with changes in regulations and update your security practices accordingly to maintain compliance over time.

Practical Activity 3.3.6: Implementation of Security Testing in Nodejs

Task:

1. Read key reading 3.3.6

2. Referring to the previous theoretical activities (3.3.5) you are requested to go to the computer lab to implement security testing in node js.

3. Apply safety precautions

4. Present the steps to implement security testing in node js.

5. Referring to the steps provided in task 4, implement security testing in node js.

6. Present your work to the trainer and the whole class.

Key readings 3.3.6: Implementation of Security Testing in Nodejs

Implementing security testing in Node.js involves integrating various testing methodologies and tools throughout the development lifecycle to ensure that your application is secure from vulnerabilities and potential attacks. Below is a detailed guide on how to implement security testing in Node.js:

1. Setting Up the Environment

Install Node.js: Ensure that you have Node.js installed on your system. Use the latest LTS version for stability and security.

Create a Project: Set up a new Node.js project using npm init or yarn init.

Install Security Tools: Install essential security testing tools and frameworks, such as Mocha for testing, Chai for assertions, and specific security tools like ESLint, Snyk, or OWASP ZAP.

2. Static Application Security Testing (SAST)

Use ESLint with Security Plugins:

Install ESLint: npm install eslint --save-dev.

Add security plugins like eslint-plugin-security: npm install eslint-plugin-security -

-save-dev.

Configure ESLint to run on your codebase to detect insecure coding patterns and vulnerabilities.

Dependency Scanning:

Use npm audit: Automatically scan your project's dependencies for known vulnerabilities.

Integrate Snyk: Install Snyk with npm install -g snyk and run snyk test to scan dependencies and get remediation advice.

Automate SAST: Integrate SAST tools into your CI/CD pipeline to run automatically on every code push.

3. Dynamic Application Security Testing (DAST)

Set Up OWASP ZAP:

Install OWASP ZAP for dynamic testing of your running application against various attack vectors.

Use the ZAP CLI or Docker container to automate the process of scanning your Node.js application for vulnerabilities.

Automate DAST in CI/CD:

Configure ZAP to run as part of your CI/CD pipeline, triggering scans on staging environments before deployment.

4. Interactive Application Security Testing (IAST)

Implement IAST Tools:

Use tools like Contrast Security or RASP (Runtime Application Self-Protection) to monitor and protect your application at runtime.

Integrate IAST into your Node.js application to provide real-time vulnerability detection during development and testing.

5. Unit Testing for Security

Use Mocha and Chai:

Write unit tests using Mocha and Chai to test individual components for security issues, such as input validation, output encoding, and secure handling of sensitive data.

Example:

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

const expect = chai.expect;

describe('Input Validation', function() {

  it('should reject malicious input', function() {

    const input = "<script>alert('XSS')</script>";

    const sanitizedInput = sanitizeInput(input); // hypothetical function

    expect(sanitizedInput).to.not.contain('<script>');

  });

});
```

Security-Focused Tests:

Write tests that specifically check for security issues like SQL injection, XSS, CSRF, and insecure deserialization.

6. Penetration Testing

Manual Penetration Testing:

Engage in manual penetration testing to simulate real-world attacks and identify vulnerabilities that automated tools may miss.

Use tools like Burp Suite for intercepting and manipulating requests, exploring attack surfaces, and identifying security weaknesses.

Automated Pen Testing:

Use automated tools like Metasploit to conduct penetration testing and identify common vulnerabilities.

7. Security Regression Testing

Regression Test Suite:

Maintain a suite of regression tests that specifically target previously discovered vulnerabilities to ensure they don’t reappear.

Automate these tests to run whenever changes are made to the codebase.

8. Continuous Integration and Deployment (CI/CD)

Integrate Security Testing:

Ensure all security testing tools and practices are integrated into your CI/CD pipeline.

Use tools like Jenkins, GitLab CI, or CircleCI to automate the execution of SAST, DAST, IAST, and unit tests.

Automate Remediation:

Automatically trigger remediation workflows when vulnerabilities are detected, such as opening a Jira ticket or notifying the development team.

9. Logging and Monitoring

Implement Centralized Logging:

Use centralized logging solutions like ELK Stack, Graylog, or Papertrail to collect and analyze logs from your Node.js application.

Monitor for security events such as failed login attempts, unexpected access patterns, or potential data breaches.

Real-Time Alerts:

Set up real-time alerts using monitoring tools like Prometheus, Grafana, or Datadog to detect and respond to security incidents quickly.

10. Reporting and Remediation

Generate Security Reports:

After running tests, generate reports that summarize the findings, highlighting critical vulnerabilities and providing remediation steps.

Use tools like SonarQube for comprehensive reporting on code quality and security.

Remediation Workflow:

Establish a clear workflow for addressing identified vulnerabilities, including prioritization, assignment, and tracking of remediation efforts.

11. Compliance and Regular Audits

Ensure Compliance:

Implement security controls and testing procedures to comply with industry regulations (e.g., GDPR, PCI DSS).

Regular Security Audits:

Conduct regular security audits to ensure ongoing compliance and to detect any gaps in security practices.

12. Training and Awareness

Security Training:

Provide ongoing training for developers on secure coding practices and the latest security threats.

Security Awareness:

Foster a culture of security awareness within the development team, encouraging proactive identification and reporting of potential security issues.

Practical Activity 3.3.8: Perform penetration testing using OWASP

Task:

1. Read key readings 3.3.8

2. Referring to the previous theoretical activities (3.3.5) you are requested to go to the computer lab to Perform Penetration Testing using OWASP.

3. Apply safety precautions

4. Present out the steps to Perform Penetration Testing using OWASP.

5. Referring to the steps provided on task 4, perform penetration Testing using OWASP.

6. Present your work to the trainer and whole class.

Key readings 3.3.8: Perform penetration testing using OWASP

Performing penetration testing using OWASP involves several steps to ensure thorough assessment and documentation of vulnerabilities.

Steps for Penetration Testing with OWASP

a. **Planning and Preparation**

Define the scope of the penetration test (e.g., web application, API).

Obtain necessary permissions and legal agreements.

Identify the testing team and assign roles.

b. Information Gathering

Use tools like OWASP ZAP or Burp Suite to gather information about the target.

Techniques include:

- ✓ DNS enumeration
- ✓ Port scanning with Nmap
- ✓ Identifying web technologies using Wappalyzer

Example: For a sample e-commerce website, gather information about its domain, IP address, and technologies used (e.g., PHP, MySQL).

c. Threat Modeling

Identify potential threats and vulnerabilities based on the application architecture.

Use the OWASP Top Ten as a reference to understand common vulnerabilities.

Example: For the e-commerce site, threats might include SQL Injection, Cross-Site Scripting (XSS), and insecure direct object references.

d. Vulnerability Assessment

Use automated tools to scan for vulnerabilities.

Common tools include:

- ✓ OWASP ZAP
- ✓ Nessus
- ✓ Nikto

Example: Run an automated scan on the e-commerce application and identify vulnerabilities like outdated libraries or misconfigurations.

e. Exploitation

Attempt to exploit identified vulnerabilities to determine their impact.

Use manual testing techniques in addition to automated tools.

Example: If a SQL Injection vulnerability is identified, use a tool like SQLMap to exploit it and extract data from the database.

f. Post-Exploitation

Assess the level of access gained and the potential for lateral movement within the network.

Document findings and any sensitive data accessed.

Example: After exploiting SQL Injection, check for sensitive user data or admin credentials that could be accessed.

g. Reporting

Create a detailed report outlining:

- ✓ Methodologies used
- ✓ Vulnerabilities found.
- ✓ Exploits performed.
- ✓ Recommendations for remediation

Example: Document the SQL Injection vulnerability, how it was exploited, the data accessed, and recommendations for input validation and parameterized queries.

h. Remediation and Verification

Work with the development team to fix identified vulnerabilities.

Conduct a follow-up test to verify that issues have been resolved.

Example: After the e-commerce site has implemented fixes, perform another round of testing to ensure the SQL Injection vulnerability has been remediated effectively.

i. Continuous Improvement

Encourage ongoing security practices.

Suggest regular penetration testing and security assessments.

1. Installation of OWASP Tool

OWASP (Open Web Application Security Project) provides several tools for different aspects of security testing. One popular tool for penetration testing is OWASP ZAP (Zed Attack Proxy). Here’s how you can install OWASP ZAP:

OWASP ZAP Installation:

Download: Go to the OWASP ZAP official website (https://www.zaproxy.org/download/) and download the appropriate version for your operating system.

Install: Follow the installation instructions provided for your OS (usually involves running an installer or extracting files).

Setup: Configure any necessary settings during the installation process.

2. Perform Scan

Once OWASP ZAP is installed, you can start scanning your target application

Configure Target:

Open OWASP ZAP.

Set up the target URL or IP address of the application you want to test.

Start Scanning:

Initiate an active scan from OWASP ZAP.

Wait for the scan to complete, which involves OWASP ZAP probing the application for vulnerabilities.

3. Exploit Vulnerabilities

After identifying potential vulnerabilities, you may choose to exploit them to understand their impact and verify their existence:

Verify Vulnerabilities:

OWASP ZAP provides details on vulnerabilities found during the scan.

Verify each vulnerability by attempting to exploit it, simulating how an attacker might exploit it in real-world scenarios.

4. Interpret Scan Report

Once the scan is complete, you'll need to interpret the results:

Review Findings:

Analyze the scan report generated by OWASP ZAP.

Prioritize vulnerabilities based on severity (e.g., High, Medium, Low).

Understand the technical details provided about each vulnerability.

Contextualize Impact:

Consider the impact of each vulnerability in terms of potential risks to the application and its users.

Evaluate the likelihood of exploitation and potential consequences.

5. Document Results

Documenting the results of your penetration test is crucial for reporting and remediation:

Create Report:

Compile findings into a structured report.

Include details such as vulnerability descriptions, technical details, affected components, and remediation recommendations.

Use OWASP ZAP’s reporting features or export data to create customized reports.

Recommendations

Provide actionable recommendations for mitigating identified vulnerabilities.

Prioritize recommendations based on severity and potential impact.

Presentation

Present the findings to stakeholders, including technical teams, management, and developers.

Discuss implications and strategies for addressing vulnerabilities.

Points to Remember

- Node.js is a popular runtime environment for building scalable network applications. However, with its popularity, it becomes a target for various security threats like the followings:
  - ✓ Injection Attacks
  - ✓ Broken Authentication and Session Management
  - ✓ Cross-Site Scripting (XSS)
  - ✓ Cross-Site Request Forgery (CSRF)
  - ✓ Security Misconfiguration

- ✓ Insecure Cryptographic Storage
- ✓ Insufficient Authorization
- ✓ Insufficient Logging and Monitoring

- Security testing in Node.js involves using various tools to identify vulnerabilities and ensure that your application is secure. Those security testing tools and frameworks that came classified in the following types
  - ✓ Static Analysis Tools
  - ✓ Dynamic Analysis Tools
  - ✓ Testing Frameworks
- Secure coding practices are essential to building robust and secure Node.js applications. These practices help prevent common vulnerabilities and ensure that your application is resilient against attacks. Below are key secure coding practices in Node.js.
  1. Input Validation and Sanitization
  2. Use of Parameterized Queries
  3. Authentication and Authorization
  4. Secure Error Handling
  5. Avoid Hardcoding Secrets
  6. Secure Dependencies
  7. Implement HTTPS/TLS
  8. Protect Against Cross-Site Scripting (XSS)
  9. Prevent Cross-Site Request Forgery (CSRF)
  10. Secure Session Management
  11. Secure File Uploads
  12. Use Security Headers
  13. Conduct Regular Security Audits
- Testing techniques for Node.js security is essential for identifying and mitigating vulnerabilities in your application. These techniques ensure that your Node.js application is robust, secure, and capable of withstanding various attacks. Those are testing techniques in node js.
  1. Static Application Security Testing (SAST)
  2. Dynamic Application Security Testing (DAST)
  3. Interactive Application Security Testing (IAST)
  4. Penetration Testing
  5. Fuzz Testing
  6. Security Unit Testing
  7. Continuous Security Testing

8. Threat Modelling
9. Vulnerability Scanning
10. Security Regression Testing

- **Best Practices for Node.js Security Testing include the followings:**
  1. Integrate Security Testing Early and Often
  2. Use a Comprehensive Security Testing Suite
  3. Employ Threat Modelling
  4. Regularly Update and Audit Dependencies
  5. Perform Regular Penetration Testing
  6. Implement Security Regression Testing
- Implementing security testing in Node.js involves integrating various testing methodologies and tools throughout the development lifecycle to ensure that your application is secure from vulnerabilities and potential attacks. Below is a guide on how to implement security testing in Node.js:

  1. Setting Up the Environment
  2. Static Application Security Testing (SAST)
  3. Dynamic Application Security Testing (DAST)
  4. Interactive Application Security Testing (IAST)
  5. Unit Testing for Security
  6. Penetration Testing
  7. Security Regression Testing
  8. Continuous Integration and Deployment (CI/CD)
  9. Logging and Monitoring
  10. Reporting and Remediation
  11. Compliance and Regular Audits
  12. Training and Awareness
- Performing penetration testing using OWASP involves several steps to ensure thorough assessment and documentation of vulnerabilities.

  1. Installation of OWASP Tool
  2. Perform Scan
  3. Exploit Vulnerabilities
  4. Interpret Scan Report
  5. Document Results

Application of learning 3.3.

TelaTech Ltd is a company that develops websites for different institutions they have provided you the developed database (KigaliinnovationDB) and inside created a table called clients with the following fields ID, Names, Sex, Address, Phone and Email in node js and developed APIs that will be used to insert, update, select and delete client's records via frontend where you will provide the developed APIs to be integrated with front end.

They have included HTTP status codes to overcome error-handling issues.

You have been tasked for:

- ✓ Implementation of Security Testing in Nodejs
- ✓ Perform penetration Testing using OWASP

Learning outcome 3 end assessment

Written assessment

1. What is the primary purpose of unit testing?

a) To test the entire application as a whole

b) To validate individual components or functions

c) To identify security vulnerabilities

d) To test user interface responsiveness

2. Which of the following is a popular unit testing framework for Node.js?

a) Jest

b) JUnit

c) Selenium

d) Mocha

3. Which assertion library is commonly used with Mocha?

a) Jasmine

b) Chai

c) QUnit

d) RSpec

4. What command is used to install Mocha in a Node.js project?

a) npm install mocha

b) npm install chai

c) npm install jest

d) npm install selenium

5. In Chai, which API is used for behavior-driven development (BDD) style assertions?

a) Should

b) Must

c) Could

d) Shall

6. What is the main benefit of using the Expect API in Chai?

a) It is simpler to write and understand

b) It provides advanced security testing

c) It automates UI testing

d) It is faster in execution

7. What is the first step in the unit testing process?

a) Running the test cases

b) Writing the test cases

c) Analyzing test results

d) Setting up the testing environment

8. Which tool is commonly used for usability testing?

a) Postman

b) Puppeteer

c) Usabilla

d) OWASP ZAP

9. What does the acronym OWASP stand for?

a) Open Web Application Security Project

b) Online Web Application Security Platform

c) Open Web and Software Protection

d) Online Web Application Security Protection

10. Which of the following is NOT a type of security vulnerability?

a) SQL Injection

b) Cross-Site Scripting (XSS)

c) Unit Testing

d) Cross-Site Request Forgery (CSRF)

11. Which command installs the Chai assertion library?

a) npm install chai

b) npm install mocha

c) npm install jest

d) npm install puppeteer

12. What type of testing is Postman primarily used for?

a) Unit Testing

b) API Testing

c) Usability Testing

d) Load Testing

13. What is the primary function of Puppeteer?

a) Simulate user interactions in web applications

b) Secure web applications

c) Conduct static code analysis

d) Manage server configurations

14. In Mocha, which hook runs before all test cases?

a) before()

b) after()

c) beforeEach()

d) afterEach()

15. Which is NOT a key benefit of unit testing?

a) Early bug detection

b) Reduced maintenance cost

c) Increased code coverage

d) Improved network performance

16. What is the purpose of the describe() function in Mocha?

a) To group related test cases

b) To define global variables

c) To execute tests in parallel

d) To handle asynchronous operations

17. Which of the following is a feature of the Chai Expect API?

a) It allows chaining of assertions

b) It directly manipulates the DOM

c) It provides built-in logging

d) It handles network requests

18. Which method is used to run tests in Mocha?

a) mocha run

b) npm test

c) mocha --exec

d) test start

19. What is an example of a dynamic analysis tool for security testing in Node.js?

a) OWASP ZAP

b) JSHint

c) ESLint

d) GitHub Copilot

20. Which framework is commonly used to implement security tests in Node.js?

a) OWASP

b) Mocha

c) Chai

d) Cypress

21. What is a key benefit of using automated testing tools like Puppeteer?

a) Reduces human error

b) Eliminates the need for testing

c) Increases application runtime

d) Provides advanced security features

22. Which command is used to run a collection in Postman?

a) postman run

b) newman run

c) postman exec

d) newman exec

23. In Mocha, how do you skip a test case?

a) Use it.skip

b) Use it.skip()

c) Use describe.skip()

d) Use test.skip()

24. Which of the following is an example of a static analysis tool?

a) ESLint

b) Burp Suite

c) Postman

d) JMeter

25. What is a common use case for Chai's should API?

a) Writing test assertions in a more natural language style

b) Running multiple tests in parallel

c) Automating user interface tests

d) Managing database connections

II. Read the following statement related to test backend application in column A and B and write the number corresponding to the correct description

1. The Unit Testing Tools with their primary language support:

Answer	Column A	Column B
…………………	a) Mocha	1. JavaScript
………………..	b) Junit	2. Python
………………..	c) PyTest	3. Java
………………..	d) RSpec	4. Ruby

2.The Security Testing Tool with its function:

Answer	Column A	Column B
…………………	a) OWASP ZAP	1. Static code analysis
………………..	b) ESLint	2. Dynamic application security testing
………………..	c) Burp Suite	3. Automated web vulnerability scanner
………………..	d) SonarQube	4. Code quality analysis

3. The Puppeteer function with its description:

Answer	Column A	Column B
…………………	a) page.goto()	1. Simulates a browser click
………………..	b) page.click()	2. Takes a screenshot of the page
………………..	c) page.screenshot()	3. Navigates to a URL
………………..	d) page.type()	4. Enters text into an input field

4. The Chai assertion style with its description:

Answer	Column A	Column B
…………………	a) Expect	1. Classic assert style without chaining"

…………………	b) Should	2. Assert style chaining with "expect
…………………	c) Assert	3.Not a Chai assertion style
…………………	d) Chain	4. Assert style using "should"

5.The Usability Testing Tool with its feature:

Answer	Column A	Column B
…………………	a) Postman	1. Collecting user feedback on websites
…………………	b) Usabilla	2. Remote user testing and reporting
…………………	c) Optimizely	3. API testing and automation
…………………	d) Maze	4. A/B testing and experimentation

6.The type of vulnerability with its description:

Answer	Column A	Column B
…………………	a) SQL Injection	1.Executing unauthorized SQL commands
…………………	b) XSS	2. Storing sensitive data without proper encryption
…………………	c) CSRF	3. Injecting malicious scripts into web pages
…………………	d) Insecure Cryptographic Storage	4. Forcing users to perform actions they did not intend

7. The OWASP Top 10 risk with its associated attack type:

Answer	Column A	Column B
…………………	a) Injection	1. Script injection
…………………	b) Broken Authentication	2. Default passwords
…………………	c) Cross	3.Credential stuffing
…………………	d) Security Misconfiguration	4.SQL Injection

8. The Postman feature with its description:

Answer	Column A	Column B
…………………	a) Collections	1. Storing reusable data like API keys
…………………	b) Environment Variables	2. Grouping related API requests
…………………	c) Pre-request Script	3. Validating responses from API requests

…………………	d) Tests	4. Executing scripts before a request is sent

9. The Node.js Security Practice with its best practice:

Answer	Column A	Column B
………………….	a) Use of SSL/TLS	1.Keeping packages UpToDate
…………………	b) Input Validation	2. Encrypting data in transit
…………………	c)Dependency Management	3. Avoiding exposure of sensitive information
…………………	d) Error Handling	4. Sanitizing user inputs to prevent attacks

10. The unit test concept with its description

Answer	Column A	Column B
…………………	a) Test Case	1. A collection of related test cases
…………………	b) Test Suite	2. Simulating dependencies in a test
…………………	c) Mocking	3. A statement that must be true for the test to pass
…………………	d) Assertion	4. A single scenario to validate

III. Complete the following statement related to test backend application with the appropriate term used in backend application development: **Expect, SQL Injection, individual components or functions work as expected, describe (), document, automate, user experience and interface design, insecure, test, validate, before Each(),penetration testing, SSL/TLS encryption, scan, API**

1. Unit testing is primarily conducted to ensure that _______.
2. In Chai, the _______ API allows for natural language assertions.
3. The Mocha testing framework uses the _______ method to group related tests.
4. Usability testing is important because it helps identify issues with _______.
5. A common vulnerability in web applications, where an attacker can inject SQL queries, is known as _______.
6. Postman is commonly used to _______ API endpoints and automate testing.
7. Puppeteer allows developers to _______ user interactions in a headless Chrome browser.

8. In security testing, _______ is used to identify potential vulnerabilities before an attacker can exploit them.
9. Node.js applications should always use _______ to ensure data is transmitted securely.
10. A test case in unit testing is designed to _______ a specific function or feature.
11. The OWASP ZAP tool is used to _______ web applications for vulnerabilities.
12. In Mocha, the _______ hook runs before each individual test case.
13. Chai's should _______ is often used for assertions in a BDD style.
14. Security misconfigurations often arise from _______ settings or defaults.
15. After performing a penetration test, it is crucial to _______ the findings in a detailed report.

Practical assessment

TelaTech ltd is a company that develop websites for different institutions they have provided you the developed database (Kigali innovation DB) and inside created a table called clients with the following fields ID, Names, Sex, Address, Phone and Email in node js and developed APIs that will be used to insert, update, select and delete client's records via frontend where you will be provide the developed APIs to be integrated with front end.

They have included http status codes in order to overcome error handling issues.

They have created a login form that will be used once accessing the system.

You have been tasked for:

- ✓ Use Postman testing tool for usability testing on APIs as provided.
- ✓ Use Puppeteer testing tool for usability testing.
- ✓ Using Mocha Testing Framework for unit testing
- ✓ Monitoring Test results
- ✓ Implementation of Security Testing in Nodejs
- ✓ Perform penetration Testing using OWASP for security testing

All tools' materials and equipment will be provided by the company.

END

References

Books

Doglio, F. (2018). Rest API Development with Node.Js. Uruguay: Apress.

Mardan, A. (2018). Practical Node.js. California: Apress.

Mike Cantelon, M. H. (2014). Node.js IN ACTION. Shelter Island, NY11964: Manning Shelter Island.

Web site links

Adaptable. (2024). *deploy-nodejs-app*. Retrieved from adaptable: https://adaptable.io/docs/app-guides/deploy-nodejs-app

Dizdar, A. (2022, Jurly 27). *unit-testing-in-nodejs*. Retrieved from Bright: https://brightsec.com/blog/unit-testing-in-nodejs/

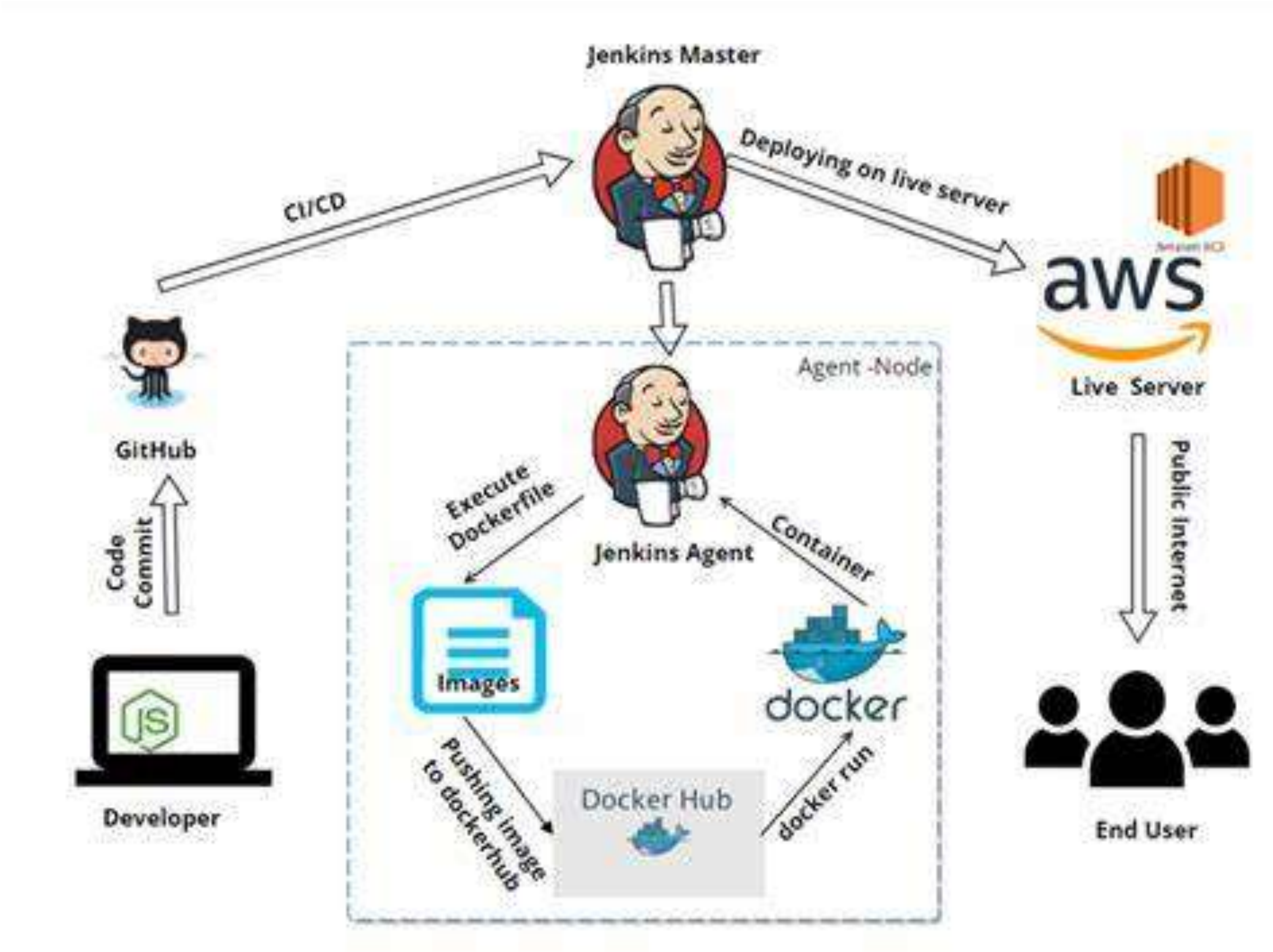
Ebere, N. S. (2022, March 3). *how-to-build-a-backend-application*. Retrieved from freecodecamp: https://www.freecodecamp.org/news/how-to-build-a-backend-application

learnthinkcreatecode. (2022, October 25). *How to test a Node.js REST API using postman.*

Nael, D. (2019, March 11). Retrieved from okta: https://developer.okta.com/blog/2019/03/11/node-sql-server

YouTube. (2020, September 13). *watch?v=VStXlFxQgZg*. Retrieved from youtube: https://www.youtube.com/watch?v=VStXlFxQgZg

YouTube. (2022, October 25). *watch?v=zLbsM_n1H9w*. Retrieved from youtube: https://www.youtube.com/watch?v=zLbsM_n1H9w


Jenkins Master
CI/CD
Deploying on live server
aws
Amazon EC2
Live Server
GitHub
Code Commit
Developer
Agent -Node
Jenkins Agent
Execute Dockerfile
Container
Images
docker
Pushing image to dockerhub
Docker Hub
docker run
Public Internet
End User

Indicative contents
4.1 Preparation of deployment Environment **4.2 Implementation of Manual Deployment of NodeJS application** **4.3 Maintenance of NodeJS application** **4.4 Application of NodeJS Documentation Tools and Frameworks**

Key Competencies for Learning Outcome 4: Manage Backend Application

Knowledge	Skills	Attitudes
● Description of NodeJS application deployment ● Description of Types of NodeJS application deployment ● Developing a maintenance plan ● Describing the Application of NodeJS Documentation Tools and Frameworks	● Maintaining NodeJS application ● Preparing NodeJS Application Deployment tools ● Publishing Documentation ● Implementing Manual Deployment of NodeJS application	● team work ● Being a critical thinker ● Being Innovative ● Being creative ● Practical oriented ● Detail oriented ● Being honesty ● Having a Passion for Learning ● Problem-Solving Mindset ● Being a Collaborator and Communicator ● Attention to Security ● Ethical Coding

Duration: 15 hrs

Learning outcome 4 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly NodeJS application deployment as used in backend application.
2. Describe correctly the Types of NodeJS application deployment as used in backend application
3. Develop effectively a maintenance plan based on deployed application.
4. Describe correctly the Application of NodeJS Documentation Tools and Frameworks as used in nodejs application deployment
5. Maintain NodeJS application deployed perfectly as used in nodejs application deployment
6. Prepare the NodeJS Application Deployment tools correctly as used in the server machine.
7. Publish documentation correctly as used in nodejs application deployment
8. Implement manual deployment of NodeJS application effectively as used in nodejs application deployment.

Resources

Equipment	Tools	Materials
● Computer	● Browser ● Node.Js ● Text Editor ● Express. Js ● Postman ● GitHub ● Swagger ● OWASP	● Internet

	● Webserver ● MySQL Workbench ● Winston ● PM2 ● Redis ● AWS Lamda	

Indicative content 4.1: Preparation of Deployment Environment

Duration: 4 hrs

Theoretical Activity 4.1.1: Description of NodeJS application deployment

Tasks:

1: You are requested to answer the following questions related to the description of NodeJS application deployment:

i. Define the following terms "Node.js application deployment"

ii. Write short notes about the following terms used in nodejs application deployment:

a) Environment Configuration
b) Dependencies
c) Build the Application
d) Testing

iii. Differentiate the types of Node.js application deployment

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 4.1.1. In addition, ask questions where necessary.

Key readings 4.1.1: Description of NodeJS Application Deployment

Node.js application deployment refers to the process of preparing, transferring, and running a Node.js application on a server or cloud environment. It involves setting up the necessary runtime environments, dependencies, configurations, and ensuring the application runs smoothly in a production environment.

The deployment process can be done manually or automated using various tools and platforms.

Preparing the Application

Environment Configuration: Ensure that your application is configured to work in different environments (development, production).

Use environment variables to manage sensitive information like API keys, database credentials, etc.

Dependencies: Make sure all necessary Node.js modules are installed. Run npm install to install dependencies listed in package.json.

Build the Application: If your application includes any frontend code (e.g., React, Angular), build the frontend assets using tools like Webpack or create a production build with commands like npm run build.

Testing: Thoroughly test your application to ensure that it behaves as expected in the production environment.

Types of NodeJS Application Deployment

1. Manual Deployment:

Manual deployment involves manually transferring the application's codebase to a server, installing dependencies, and starting the application. It requires direct interaction with the server, often through SSH, and offers full control over the deployment process.

Steps:

- ✓ Transfer the code to the server.
- ✓ Install necessary packages and dependencies.
- ✓ Start the application using Node.js.

2. Continuous Deployment:

Continuous deployment is an automated process where changes in the codebase are automatically tested, integrated, and deployed to production. It leverages Continuous Integration/Continuous Deployment (CI/CD) pipelines to streamline the process, ensuring that every update is deployed without manual intervention.

Steps:

- ✓ Set up a CI/CD pipeline using tools like Jenkins, GitHub Actions, or GitLab CI.
- ✓ Automated testing, building, and deployment of the application.
- ✓ Monitor the deployment process to ensure successful delivery.

3. Docker-based Deployment:

Docker-based deployment involves containerizing the Node.js application using Docker. Containers encapsulate the application and its dependencies, making it easier to deploy in any environment. Docker containers ensure consistency across development, testing, and production environments.

Steps:

- ✓ Create a Dockerfile to define the environment.
- ✓ Build a Docker image from the Dockerfile.
- ✓ Deploy the Docker container on a host machine or cloud service.

Points to Remember

- Node.js application deployment involves preparing, transferring, and running a Node.js app on a server or cloud environment, with proper setup for runtime environments, dependencies, and configurations.
- Deployment types include manual, continuous, and Docker-based.
- Manual deployment requires hands-on server management, while continuous deployment automates the process via CI/CD pipelines.
- Docker-based deployment uses containers for consistent environments.

Practical Activity 4.1.2: Preparation of NodeJS Application Deployment tools

Task:

1. Read key reading 4.1.2 and ask clarification where necessary

2. Referring to the previous theoretical activities (4.1.1) you are requested to go to the computer lab to Prepare NodeJS Application Deployment tools: NodeJS Runtime, Package Manager, Operating system, Webserver and Database.

3. Apply safety precautions

4. Present out the steps to Prepare NodeJS Application Deployment tools: NodeJS Runtime, Package Manager, Operating system, Webserver and Database.

5. Referring to the steps provided on task 4, Prepare NodeJS Application Deployment tools.

6. Present your work to the trainer and whole class.

Key readings 4.1.2: Preparation of NodeJS Application Deployment tools

Deploying a Node.js application on a Windows Server requires setting up several components, including the Node.js runtime, a package manager, an operating system (Windows Server), a web server (IIS), and a database (MySQL). Below is a detailed step-by-step guide to prepare the deploying your Node.js application manually on a Windows Server.

- **Set Up the Operating System (Windows Server)**

Ensure that your Windows Server is installed, updated, and configured properly. This includes setting up network settings, security configurations, and firewall settings.

1. **Install and Configure Windows Server**:

- ✓ Make sure that Windows Server is installed and running on your machine.
- ✓ Keep your server updated with the latest Windows Updates to ensure security and performance.
- ✓ Configure your firewall to allow necessary ports, especially for web traffic (usually port 80 for HTTP and port 443 for HTTPS).

2. **Install Node.js Runtime**

The Node.js runtime is necessary to run your Node.js application on the server.

- ✓ **Download Node.js**:
  - Go to the official Node.js website and download the latest LTS (Long-Term Support) version for Windows. The LTS version is recommended for production environments due to its stability
  - Choose the Windows Installer (.msi) for easy installation.
- ✓ **Install Node.js**
  - Run the downloaded installer. During the installation process:

- Accept the license agreement.
- Choose the installation directory (the default is usually fine).

- Ensure that the option to add Node.js to the system PATH is checked.

- Complete the installation by following the prompts.

✓ **Verify the Installation**

- Open a Command Prompt (you can do this by searching for cmd in the Start menu).
- Type the following commands to check that Node.js and npm (Node Package Manager) are installed correctly:

- node -v
- npm -v

  - You should see the installed versions of Node.js and npm.

3. **Install a Package Manager (npm)**

npm is the default package manager for Node.js and is automatically installed with Node.js. It allows you to manage and install packages that your Node.js application depends on.

✓ **Verify npm Installation**

- npm should have been installed with Node.js. To verify, use:

```
npm -v
```

- If npm is not installed, reinstall Node.js ensuring that npm is included.

✓ **Update npm (Optional)**

- You can update npm to the latest version using:

```
npm install -g npm
```

4. **Set Up the Web Server (IIS)**

IIS (Internet Information Services) is a web server from Microsoft, and it can be used to host your Node.js application.

✓ **Install IIS**:

- **Open Server Manager** on your Windows Server machine.
- Click on **Manage** in the top right corner and then select **Add Roles and Features**.
- In the **Add Roles and Features Wizard**, click **Next** until you reach the **Server Roles** section.
- Check the box for **Web Server (IIS)** and click **Next** until you reach the confirmation page.
- Click **Install**.

✓ **Install the IIS Node.js Module**

- IIS can host Node.js applications using the iisnode module. However, a simpler approach is to use IIS as a reverse proxy, which forwards requests to a Node.js process.
- You can also use the Application Request Routing (ARR) extension to set up reverse proxy capabilities.

✓ **Configure IIS**

- **Open IIS Manager** from the Start menu.
- **Add a New Website**:

  ○ In IIS Manager, right-click on **Sites** in the left panel and select **Add Website**.
  ○ Set up a **Site Name** (e.g., MyNodeApp).
  ○ Set the **Physical Path** to the directory where your Node.js application is stored.
  ○ Set the **Binding** information, such as the port (default is 80 for HTTP).

- **Configure the Application as a Reverse Proxy**:

  ○ Install the URL Rewrite Module to set up reverse proxy rules.
  ○ Set up a reverse proxy rule to forward all traffic to your Node.js application running on a specific port (e.g., 3000).

5. **Install MySQL Database**

MySQL is a popular relational database management system, and it will be used to store your application data.

✓ **Download MySQL Installer**

- Go to the **MySQL website** and download the MySQL Installer for Windows.

✓ **Install MySQL Server**

- Run the MySQL Installer and choose the setup type. For a server-only installation, choose **Server Only**.
- During installation, you'll be prompted to set a **root password**. Remember this password as it will be required for database access.
- Choose to configure MySQL as a Windows service so that it starts automatically with Windows.

✓ **Create a Database**

- Once MySQL is installed, create a database for your application.
- You can use **MySQL Workbench** (installed with the MySQL server) or the command line to create a new database.

- **Using Command Line**

- Open Command Prompt and log into MySQL:
- `mysql -u root -p`
- After entering your password, create a new database:

```
CREATE DATABASE mydatabase;
```

- Exit MySQL with exit.

Points to Remember

- Node.js application deployment involves preparing, transferring, and running a Node.js app on a server or cloud environment, with proper setup for runtime environments, dependencies, and configurations.
- Types of NodeJS application deployment we have:
  - Manual Deployment
  - Continuous Deployment
  - Docker-based deployment

- Deploying a Node.js application on a Windows Server with a MySQL database. You'll use IIS to serve the application, ensuring it is accessible to users via the web.
- PM2 will manage your application process, and MySQL will handle your application's data.
- Make sure to continuously monitor, update, and secure your application and server to keep your deployment stable and secure.

Application of learning 4.2.

STU LTD is a software development company located in Musanze district, it develops software both frontend and backend for different companies, due to different activities that they have, you are hired as backend developer and deployment of node js application responsible for Preparing the following NodeJS Application Deployment tools: NodeJS Runtime, Package Manager, Operating system, Webserver and Database.

Indicative content 4.2: Implementation of Manual Deployment

Duration: 4 hrs

Practical Activity 4.2.1: Implementation of Manual Deployment of NodeJS application.

Task:

1. Read key reading 4.2.1 and ask clarification where necessary

2. Referring to the previous theoretical activities (4.1.1) you are requested to go to the computer lab to deploy the project Manual Deploying of NodeJS application by copying the application source code to the server, installing of dependencies, Start the application-using command line.

3. Apply safety precautions

4. Present out the steps to deploy the project Manual Deploying of NodeJS application by copying the application source code to the server, installing of dependencies, Start the application-using command line.

5. Referring to the steps provided on task 4, Manual Deploying of NodeJS application.

6. Present your work to the trainer and whole class.

Key readings 4.2.1: Implementation of Manual Deployment of a Node.js Application on Windows

Here's a step-by-step guide to manually deploy a Node.js application on a Windows server:

1. Copy the Application Source Code to the Server

Option 1: Using Remote Desktop (RDP)

- **Connect to the Windows Server:**
  - Open Remote Desktop Connection (mstsc) on your local machine.
  - Enter the server's IP address and connect using your credentials.

- **Transfer Files:**
  - **Copy and Paste:** You can use the clipboard to copy files from your local machine and paste them into the server's directory. Drag and drop files between the local and remote desktop environments.

Option 2: Using FTP/SFTP

- **Install an FTP Client:**
  - Use an FTP client like FileZilla.
- **Connect to the Server:**
  - Enter the server's IP address, port (usually 21 for FTP or 22 for SFTP), and your credentials.
- **Transfer Files:**
  - Navigate to the destination directory on the server and upload your application files.

Option 3: Using SCP

- **Use SCP Tool:**

If SCP is available, open Git Bash or a terminal and run:

```
scp -r /path/to/local/nodejs-app username@server-ip:/path/to/server/directory
```

- **Example Command:**

Copy the directory nodejs-app from your local machine to the server:

```
scp -r C:\path\to\nodejs-app username@server-ip:C:\path\to\server\directory
```

2. Installation of Dependencies

- **Access the Windows Server:**
  - Open Command Prompt or PowerShell on the server.
- **Navigate to the Application Directory:**
  - Change to the directory where you copied your Node.js application:

```
cd C:\path\to\server\directory\nodejs-app
```

- **Install Dependencies:**
  - Ensure Node.js is installed on the server. Check with:

```
node -v
```

- Install the project dependencies specified in package.json:

```
npm install
```

3. Start the Application Using Command Line

- **Start the Application:**
  - You can start your Node.js application with:

```
node index.js
```

  - Ensure that index.js is the entry point to your application as specified in your package.json.
- **Alternative with npm start:**
  - If your package.json has a start script defined:

```
"scripts": {
  "start": "node index.js"
}
```

  - Start the application using:

```
npm start
```

4. Optional: Using PM2 for Process Management

- **Install PM2 Globally:**
  - PM2 is a process manager for Node.js applications:

```
npm install -g pm2
```

- **Start the Application with PM2:**
  - Use PM2 to start and manage your application:

```
pm2 start index.js
```

- **Manage PM2 Processes:**
  - View running processes:

```
pm2 list
```

- Stop, restart, or delete processes as needed:

```
pm2 stop index
pm2 restart index
pm2 delete index
```

- **Save PM2 Process List:**
  - To ensure processes restart on server reboot:

```
pm2 startup
pm2 save
```

Points to Remember

- **While we are implementing manual deployment, to Copy the Application we** use RDP, FTP, SFTP, or SCP to transfer files to the Windows server. To **install dependencies, we** use npm install to set up dependencies on the server.
- To **Start the Application, we** run the application using node index.js or npm start. Then **optional use PM2:** For process management and auto-restart functionality.

Application of learning 4.2.

STU LTD is a software development company located in Musanze district, it develops software both frontend and backend for different companies, due to different activities that they have, you are hired as backend developer and deployment of node js application responsible for deploying the project manually by Copying the application source code to the server, installing of dependencies, Start the application using command line.

Indicative content 4.3: Maintenance of NodeJS Application

Duration: 4 hrs

Theoretical Activity 4.3.1: Description of Best practices for maintenance

Tasks:

1: You are requested to describe best practices for maintaining nodejs application.

2: Provide the findings for write them on papers/flipchart.

3: Present the findings to the whole class and trainer.

4: For more clarification, read the key readings 4.3.1. In addition, ask questions where necessary.

Key readings 4.3.1.: Description of Best practices for maintenance

Best Practices for Maintenance of Node.js Application

1. Update Regularly

Dependencies:

Regularly check for updates to your dependencies using tools like npm outdated.

Use npm audit to identify vulnerabilities in your dependencies and address them promptly.

Consider using npm-check-updates to easily update package versions in package.json.

Node.js Version:

Keep your Node.js version up to date to benefit from performance improvements and security patches.

Use Node Version Manager (NVM) to manage multiple Node.js versions easily.

2. Monitor the Application

Performance Monitoring:

Use monitoring tools like New Relic, Datadog, or AppDynamics to track application performance and user experience.

Monitor key metrics such as response times, request rates, and error rates.

Logging:

Implement logging using libraries like Winston or Morgan. Store logs in a centralized location for easy access and analysis.

Monitor logs for errors and unusual patterns.

Health Checks:

Implement health check endpoints in your application to verify that it’s running correctly.

Use tools like Prometheus and Grafana for alerting and visualization.

3. Perform Testing

Unit Testing:

Write unit tests for individual components of your application using frameworks like Jest, Mocha, or Chai.

Integration Testing:

Perform integration tests to ensure that different parts of your application work together as expected.

End-to-End Testing:

Use tools like Cypress or Selenium to perform end-to-end testing, simulating user interactions with the application.

Continuous Integration/Continuous Deployment (CI/CD):

Set up a CI/CD pipeline using tools like Jenkins, GitHub Actions, or CircleCI to automate testing and deployment processes.

Ensure that all tests pass before deploying new changes to production.

Maintenance Plan for a Node.js Application

1. Identification of Maintenance Requirements

Assess Current State: Evaluate the application's current performance, security vulnerabilities, and technical debt.

Gather Feedback: Collect input from users and stakeholders to identify pain points and necessary improvements.

Review Logs: Analyze application logs to identify recurring issues or bottlenecks.

2. Schedule Regular Updates

Dependencies:

Set a schedule (e.g., monthly) to review and update dependencies using tools like npm outdated and npm audit.

Node.js Version:

Plan to upgrade Node.js versions at least twice a year, based on LTS (Long Term Support) releases.

Documentation Updates:

Ensure that any changes in dependencies or Node.js versions are reflected in the documentation.

3. Automate Maintenance Tasks

Use CI/CD Pipelines:

Implement CI/CD tools (e.g., GitHub Actions, Jenkins) to automate tasks such as running tests, deploying updates, and monitoring builds.

Automated Dependency Management:

Utilize tools like Dependabot or Renovate to automatically propose updates for dependencies.

Script Routine Tasks:

Write scripts for routine tasks (e.g., backups, cleanup) to minimize manual intervention.

4. Monitor Application Performance

Set Up Monitoring Tools:

Use tools like New Relic, Datadog, or Prometheus to monitor application performance metrics (e.g., response time, error rates).

Alerts and Notifications:

Configure alerts for critical performance issues or downtime to ensure timely responses.

Regular Review:

Schedule monthly reviews of performance data to identify trends and make informed decisions.

5. Test Regularly

Automated Testing:

Implement unit, integration, and end-to-end tests as part of the CI/CD pipeline.

Scheduled Testing:

Conduct performance and security testing on a quarterly basis or after significant changes.

User Acceptance Testing (UAT):

Involve users in testing new features or changes before deployment.

6. Disaster Recovery Plan

B**ackup Strategy:**

Implement regular backups of the application data and configurations, ensuring they are stored securely and can be restored quickly.

Recovery Procedures:

Document procedures for restoring the application in the event of failure, including steps for data recovery and application redeployment.

Testing the Plan:

Regularly test the disaster recovery plan to ensure that it works effectively in practice.

7. Document Changes

Change Log:

Maintain a change log for all updates, including dependency upgrades, bug fixes, and new features.

Documentation:

Ensure that all changes to code, architecture, and deployment processes are documented for future reference.

Version Control:

Utilize Git or another version control system to track changes and facilitate collaboration among team members.

Points to Remember

- While performing best practices for maintenance a deployed nodejs application we consider on:
  - Update
  - Monitor
  - Perform test
- By implementing this maintenance plan, you can ensure that your Node.js application remains reliable, secure, and performant. Regular updates, monitoring, testing, and documentation are essential components of a successful maintenance strategy, while a solid disaster recovery plan ensures that you are prepared for unexpected issues.

Practical Activity 4.3.2: Development of maintenance plan

Task:

1. Read key reading 4.3.2

2. Referring to the previous theoretical activities (4.3.1) you are requested to go in the computer lab to Update, Monitor and Perform test as practices for maintenance for deployed nodejs application.

3. Apply safety precautions

4. Present out the steps to Update, Monitor and Perform test as practices for maintenance for deployed nodejs application.

5. Referring to the steps provided on task 4, maintain deployed nodejs application.

6. Present your work to the trainer and whole class.

Key readings 4.3.2: Development of maintenance plan

1. **Identification of Maintenance Requirements**

List Dependencies and Audit for Vulnerabilities

1. Open Command Prompt or PowerShell.
2. Navigate to your project directory:

```
cd C:\path\to\your\project
```

3. List all dependencies:

```
npm list --depth=0
```

4. Audit the project for vulnerabilities:

```
npm audit
```

Use npm-check-updates to Find Outdated Packages

- Install npm-check-updates globally:

```
npm install -g npm-check-updates
```

- Check for outdated packages:

```
ncu
```

- Update the packages:

```
ncu -u
npm install
```

2. **Schedule Regular Updates**

Set Up a CI/CD Pipeline for Automated Updates

1. **Create a GitHub Actions Workflow:**

Create a .github\workflows\ci.yml file in your project directory:

```
name: Node.js CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
  schedule:
    - cron: '0 0 * * SUN'  # Run every Sunday at midnight

jobs:
  build:
    runs-on: windows-latest

    steps:
      - uses: actions/checkout@v2
      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'

      - run: npm install
      - run: npm run build --if-present
      - run: npm test
    - run: npm audit
```

This workflow runs your CI pipeline on a Windows environment using GitHub Actions.

3. **Automate Maintenance Tasks**

Automate Database Backups with Task Scheduler

1. **Create a Backup Script (backup.bat):**
   - Create a batch file backup.bat:

```
@echo off
set BACKUP_DIR=C:\path\to\backups
set MYSQL_USER=yourusername
set MYSQL_PASSWORD=yourpassword
set MYSQL_DATABASE=yourdatabase
set DATE=%date:~10,4%-%date:~4,2%-%date:~7,2%
set TIME=%time:~0,2%-%time:~3,2%

REM Create backup directory if it doesn't exist
if not exist %BACKUP_DIR% mkdir %BACKUP_DIR%

REM Backup the database
mysqldump -u %MYSQL_USER% -p%MYSQL_PASSWORD% %MYSQL_DATABASE% > %BACKUP_DIR%\backup-%DATE%-%TIME%.sql
```

2. **Schedule the Backup Script with Task Scheduler**
   - Open **Task Scheduler**.
   - Create a new task:
     - Set the trigger to run daily at a specified time.
     - Set the action to start backup.bat.
     - Make sure to run the task whether the user is logged in or not.

4. **Monitor Application Performance**

Use PM2 on Windows

1. **Install PM2:**
   - Open Command Prompt or PowerShell:

```
npm install -g pm2
```

2. **Start Your Application with PM2:**

```
pm2 start app.js
```

3. **Monitor Your Application:**

```
pm2 monit
```

PM2 will provide a command-line interface to monitor your application.

5. **Test Regularly**

Set Up Automated Tests

1. **Install Mocha and Chai:**

```
npm install --save-dev mocha chai
```

2. **Create a Test File (test\app.test.js):**

```
const assert = require('chai').assert;
const app = require('../app');

describe('App', function() {
  it('app should return hello', function() {
    assert.equal(app(), 'hello');
  });
});
```

3. **Add a Test Script to package. json:**

```
"scripts": {
  "test": "mocha"
}
```

4. **Run Tests:**

```
npm test
```

6. **Disaster Recovery Plan**

Automate and Test Backups

As shown in Step 3, use the backup.bat script and Task Scheduler for automated backups. To practice restoring a backup:

Restore the Backup:

```
mysql -u yourusername -p yourpassword yourdatabase < C:\path\to\backups\backup-YYYY-MM-DD.sql
```

7. **Document Changes**

Maintain a Changelog

1. **Create a CHANGELOG.md File:**

```
# Changelog

## [1.0.1] - 2024-08-26
### Added
- Automated backups with Task Scheduler.

### Changed
- Updated all dependencies to the latest versions.

### Fixed
- Fixed a security vulnerability in `express`.
```

2. **Use Git to Track Changes:**

```
git add CHANGELOG.md
git commit -m "Update changelog for version 1.0.1"
git push origin main
```

Practical Activity 4.3.3: Performing Continuous maintenance and

Task:

1. Read key reading 4.3.3 and ask clarification where necessary

2. Referring to the previous practical activities (4.3.2) you are requested to go to the computer lab to Upgrade and maintain previously developed functionalities, develop new functionalities, Secure new and previously developed functionalities, Test new functionalities, Deploy new changes.

3. Apply safety precautions

4. Present out the steps to Upgrade and maintain previously developed functionalities, develop new functionalities, Secure new and previously developed functionalities, Test new functionalities, Deploy new changes.

5. Referring to the steps provided on task 4, Performing Continuous maintenance and improvement.

6. Present your work to the trainer and whole class.

Key readings 4.3.3: Performing Continuous maintenance and improvement

1. Upgrade and Maintain Previously Developed Functionalities

Action Steps:

- **Run Dependency Updates:**

Install npm-check-updates globally if not already installed:

```
npm install -g npm-check-updates
```

Check for outdated dependencies and update them:

```
ncu -u
npm install
```

- **Refactor Code:**

Use tools like ESLint to identify areas for improvement:

```
npm install eslint --save-dev
npx eslint . --fix
```

- **Review Codebase:**

Regularly review code with peers.

Create a Pull Request on GitHub for code review.

2. Develop New Functionalities

Action Steps:

- **Plan and Document:**

Use a tool like Trello or Jira to create user stories or tasks.

Document requirements in a shared document or wiki.

- **Implement Features:**

Create a new feature branch:

```
git checkout -b feature/new-functionality
```

Develop the feature and commit changes:

```
git add .
git commit -m "Implement new functionality"
```

- **Test New Code:**

Write unit and integration tests:

```
npm install jest --save-dev
npm test
```

3. Secure New and Previously Developed Functionalities

Action Steps:

- **Apply Security Patches:**

Run npm audit to identify vulnerabilities:

```
npm audit
```

Fix vulnerabilities reported:

```
npm audit fix
```

- **Implement Security Measures:**

Install helmet to set HTTP security headers:

```
npm install helmet
```

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

```
app.use(helmet());
```

Use environment variables for sensitive data:

```
npm install dotenv
```

```
require('dotenv').config();
```

4. Test New Functionalities

Action Steps:

- **Run Automated Tests:**

Use Jest or Mocha for running tests:

```
npm install jest --save-dev
npm test
```

- **Perform Manual Testing:**

Create test cases and scenarios.

Test new features in a local or staging environment.

5. Deploy New Changes

Action Steps:

- **Set Up CI/CD Pipeline:**

For GitHub Actions, create a configuration file (.github/workflows/ci.yml):

```
name: CI

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - run: npm install
    - run: npm test
    - run: npm run build
```

- **Deploy to Production:**

Use deployment tools or services like Heroku, AWS, or a custom script:

```
heroku deploy:jar --app your-app-name
```

- **Monitor and Rollback:**

Implement monitoring with tools like New Relic or Datadog.

Practice rollback procedures with:

```
git revert HEAD
```

Points to Remember

- While we are Continuously maintaining and improving NodeJS applications we:
  - **Upgrade and Maintain:**
    - Use npm-check-updates for dependency updates.
    - Refactor code with ESLint and perform code reviews.
  - **Develop New Features:** Plan, implement, and test new features using Git branches.
  - **Secure Application:** Apply security patches and implement security measures with tools like helmet and dotenv.
  - **Test:** Run automated tests with Jest and perform manual testing.
  - **Deploy Changes:** Set up CI/CD pipelines, deploy to production, and monitor performance.

Application of learning 4.3.

STU LTD is a software development company located in Musanze district, it develops software both frontend and backend for different companies, due to different activities that they have, you are hired as backend developer and deployment in node js responsible for deploying the project folder (OurWebinfo) on local disk D, and Upgrade and maintain previously developed functionalities, develop new functionalities, Secure new and previously developed functionalities, Test new functionalities, Deploy new changes.

Indicative content 4.4: Application of NodeJS Documentation Tools and Frameworks

Duration: 3 hrs

3.1

Theoretical Activity 4.4.1: Description of Node.js Documentation

Tasks:

1: In small groups, you are requested to answer the following questions related to the Description of nodejs Documentation:

I. What is documentation?
II. What is the importance of documenting in Node.js application deployment?
III. What are the differences between types of documentation?

2: Provide the answer to the asked questions and write them on paper.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 4.4.1. In addition, ask questions where necessary.

Key readings 4.4.1.: Description of nodejs Documentation

1. Documentation Overview

Documentation is an essential aspect of software development, providing detailed information about the application, its functionality, and how to use it. For Node.js applications, documentation ensures that developers and users understand how the application works, how to set it up, and how to integrate with it. Good documentation is crucial for maintenance, collaboration, and user support.

2. The Importance of Documentation

Enhances Understandability: Well-documented code and APIs make it easier for developers to understand the functionality and usage of the application.

Facilitates Maintenance: Documentation helps in maintaining and updating the application by providing clear guidelines and descriptions of its components.

Improves Collaboration: Effective documentation allows team members to collaborate more efficiently, as it provides a common understanding of the application.

Supports Users: For end-users, documentation provides guidance on how to use the application or integrate it with other systems.

3. Types of Documentation

Code Documentation: Comments and annotations within the source code explaining the purpose and functionality of code blocks, functions, and classes.

API Documentation: Describes the endpoints, methods, parameters, and responses of APIs, usually generated using tools like Swagger or Postman.

User Documentation: Guides and manuals for end-users detailing how to install, configure, and use the application.

Developer Documentation: Includes setup instructions, architecture overviews, and other technical details necessary for developers working on the project.

Design Documentation: Details the application's architecture, including diagrams and design decisions.

Points to Remember

- Documentation for Node.js applications is essential for detailing functionality, setup, and integration, enhancing understandability, maintenance, and user support. Key types include code comments, API documentation (using Swagger or Postman), user guides, and design overviews.

Theoretical Activity 4.4.2: Description of nodejs documentation tools and frameworks

Tasks:

1: In small groups, you are requested to answer the following questions related to the Description of nodejs documentation tools and frameworks:

i. Differentiate popular documentation tools and frameworks you know.

ii. Describe the best practice for documentation

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 4.4.2. In addition, ask questions where necessary.

Key readings 4.4.2: Description of nodejs documentation tools and frameworks

- **Popular Documentation Tools and Frameworks**

2. **Swagger (OpenAPI)**

   Used to create interactive API documentation that helps in designing, building, and documenting RESTful APIs.

   Features: Provides a web-based UI for exploring API endpoints, automatically generates client SDKs, and offers interactive API documentation.

   Usage: Typically integrated into the development workflow using Swagger annotations in code or a Swagger YAML/JSON file.

3. **Postman**:

   Primarily used for testing APIs but also offers features for documenting APIs.

   Features: Allows you to create collections of API requests, run tests, and generate documentation from the Postman collection.

   Usage: You can generate documentation from Postman collections and share it through a public or private link.

4. **JSDoc**:

   Generates HTML documentation from JavaScript source code comments.

   Features: Parses comments in the code to create organized, readable documentation, including function descriptions, parameters, and return values.

Usage: Comments are written in JSDoc format, and the documentation is generated using the JSDoc tool.

5. **Docdash**:

A JSDoc template that creates a more visually appealing documentation layout.

Features: Enhances the default JSDoc output with a better user interface and search capabilities.

Usage: Used as a theme with JSDoc to improve the appearance and usability of the generated documentation.

Best Practices for Documentation

Write Clear and Concise Comments: Ensure comments are straightforward, avoiding jargon, and clearly explain the purpose of code blocks, functions, and classes.

Keep Documentation Up-to-Date: Regularly update documentation to reflect changes in the codebase and ensure accuracy.

Use Consistent Formatting: Follow a consistent style and format for documentation to make it easier to read and navigate.

Include Examples: Provide examples of how to use APIs and functions to help users understand their usage better.

Organize Information: Structure documentation logically with headings, tables of contents, and search functionalities to improve usability.

Leverage Documentation Tools: Use tools like Swagger, Postman, and JSDoc to automate and enhance the documentation process.

Points to Remember

- While documenting we use Tools like Swagger, Postman, JSDoc, and Docdash help in generating and presenting documentation.
- Best practices involve clear comments, consistent formatting, and regular updates.

Practical Activity 4.4.3: Publishing Documentation

Task:

1. Read key reading 4.4.3 and ask for clarification where necessary

2. Referring to the previous theoretical activities (4.4.3) you are requested to go to the computer lab to host documentation, Use GitHub for collaborative documentation, and document the Maintenance.

3. Apply safety precautions

4. Present the steps to Publish Documentation.

5. Referring to the steps provided in task 3, Publishing Documentation.

6. Present your work to the trainer and the whole class.

Key readings 4.4.3.: Publishing Documentation

Options for Hosting Documentation:

- **Static Site Generators**: Tools like Docusaurus, Jekyll, or VuePress convert Markdown files into static HTML pages. These pages can be hosted on any web server, including GitHub Pages, Netlify, or a custom server.
- **GitHub Pages**: A free service by GitHub that allows you to host static websites directly from a GitHub repository. Ideal for projects already using GitHub for version control.
- **Read the Docs**: A popular platform for hosting open-source project documentation. It integrates with version control systems like GitHub and automatically builds and hosts your documentation.
- **Custom Servers**: Hosting documentation on your own web server gives you full control over the deployment and accessibility. This is useful for internal documentation that needs to be kept private.

Host the Documentation

Option 1: Using GitHub Pages

- Initialize a GitHub repository and push your project:

```
git init
git remote add origin <your-github-repository-url>
git add .
git commit -m "Initial commit"
git push -u origin main
```

- Create a docs/ folder with a simple index.md:

```
# My Node.js Application
This is the documentation for my simple Node.js application.
## Installation
Run the following command to install the dependencies:
npm install
```

Maintaining Your Node.js Application Using GitHub

Step 1: Monitor and Update Dependencies

Use tools like Dependabot to automatically open pull requests when your dependencies need updating.

Step 2: Use Issues and Pull Requests for Maintenance

- Open GitHub Issues to track bugs, enhancements, and other maintenance tasks.
- Use Pull Requests (PRs) for code changes. This ensures all code changes are reviewed before being merged.

Step 3: Automate Tests with GitHub Actions

- Extend your deploy.yml file to include automated tests before deployment:

```
run: npm test
```

- Ensure that all PRs pass these tests before merging.

Step 4: Documentation

- Keep your documentation updated in your repository's README.md file or in a docs/ directory.
- Encourage collaboration on documentation through PRs.

Continuous Maintenance and Improvement

Step 1: Set Up Continuous Integration (CI)

- Use GitHub Actions to automate testing and deployment, ensuring that every change is automatically tested and deployed.

Step 2: Secure Your Application

- Regularly audit your dependencies using tools like npm audit and address any vulnerabilities.
- Ensure your GitHub repository has proper security settings, like enabling branch protection.

Step 3: Deploy New Features

- Use the same workflow to deploy new features. Develop features in feature branches, test them, and merge them into main via PRs.

Step 4: Regularly Review and Update

- Regularly review your code, dependencies, and documentation to ensure everything is up to date.

Points to Remember

- You can effectively deploy, maintain, and continuously improve your Node.js application using GitHub as the primary tool for collaboration, automation, and documentation.

Application of learning 4.4.

STU LTD is a software development company located in Musanze district, it develops software both frontend and backend for different companies, due to different activities that they have, you are hired as backend developer and deployment in node js responsible for Documenting published nodejs application, with GitHub facilitating collaborative updates. Regular maintenance includes updating docs, collecting feedback, and integrating updates into the CI/CD pipeline.

Learning outcome 4 end assessment

Written assessment

I. Read the following statement related to manage backend application and choose the correct answer:

1. What is the primary purpose of Node.js application deployment?

   a) Writing JavaScript code
   b) Debugging the application
   c) Preparing, transferring, and running a Node.js application on a server or cloud environment
   d) Creating a user interface

2. Which of the following tools is commonly used for containerizing a Node.js application?

   a) Jenkins
   b) Docker
   c) Nginx
   d) Git

3. Which command is used to start a Node.js application if index.js is the entry point?

   a) npm install
   b) npm run build
   c) npm start
   d) node index.js

4. Which of the following is NOT a type of Node.js application deployment?

   a) Manual Deployment
   b) Continuous Deployment
   c) Docker-based Deployment
   d) Script-based Deployment

5. Which process manager is commonly used to manage Node.js applications in production?

   a) npm
   b) PM2
   c) Yarn
   d) GitHub Actions

II. Complete the following statement related to test backend application by choosing the correct answer

1. In a CI/CD pipeline, tools like _____________ are used to automate the testing and deployment process.

   a) Jenkins
   b) FTP
   c) npm
   d) Webpack

2. To manage environment variables and sensitive information like API keys in a Node.js application, you should use _____________.

   a) `package. json`
   b) `npm install`
   c) Environment Variables
   d) `scp`

3. Before deploying a Node.js application, it is essential to install all necessary modules using _____________.

   a) npm start
   b) npm install
   c) Docker
   d) Git

4. The command pm2 start index.js is used to _____________ the application using PM2.

   a) Install
   b) Monitor
   c) Start
   d) Delete

5. For secure file transfer to a server, tools like _____________ can be used.

   a) RDP
   b) SCP
   c) FTP
   d) Git

III. Read the following statement related to manage backend application and write the number corresponding to the correct description

1. Deployment and description

Answer	**Deployment Type**	**Description**
…………….	A) Manual Deployment	1) Automated process with CI/CD pipelines
…………….	B) Continuous Deployment	2) Containerizing the application using Docker
…………	C) Docker-based Deployment	3) Manually transferring and starting the application on a server

2. **The tool with its function:**

Answer	**Tool**	**Function**
……3……..	A) Webserver	1) Stores and retrieves application data
……1……..	B) Database	2) Executes JavaScript code on the server
……2…….	C)NodeJS Runtime	3) Serves the application and handles incoming traffic

IV. Read the following statement related to managing the backend application and answer True if the statement is correct and False if the statement is wrong

1. Node.js is a runtime environment that allows you to run JavaScript on the server side.
2. Continuous deployment means that every change that passes the automated tests is deployed to production automatically.
3. Docker is a version control system used for managing Node.js application code.
4. PM2 is used for managing and monitoring Node.js applications in production.
5. SCP is used to manage Node.js dependencies and packages.
6. In a CI/CD pipeline, Jenkins can be used to automate the deployment of a Node.js application.
7. Environment variables are used to store sensitive information, such as API keys, in a Node.js application.
8. A webserver is responsible for storing and retrieving data in a Node.js application.

Practical assessment

TelaTech Ltd is a company that develops and deploys websites for different institutions they have tasked you to deploy for them a nodejs application by Preparing a deployment Environment on the Windows server: NodeJS Runtime, Package Manager, Operating system, Webserver, and Database. Implement Manual Deployment of NodeJS application: Copy the application source code to the server, install dependencies, and start the application using the command line. Develop a maintenance plan: Identify the maintenance requirements, Schedule regular updates, automate maintenance tasks, Monitor application performance, Test regularly, Disaster recovery plan, and Document changes. In addition, perform 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, and deploy new changes. Next, apply NodeJS Documentation Tools and Frameworks: Use popular documentation tools and frameworks like Swagger/Postman for API documentation, write clear and concise comments using documentation generators then after Publish Documents and Optional Use GitHub for collaborative documentation.

END

References

Books

Doglio, F. (2018). Rest API Development with Node.Js. Uruguay: Apress.

Mardan, A. (2018). Practical Node.js. California: Apress.

Mike Cantelon, M. H. (2014). Node.js IN ACTION. Shelter Island, NY11964: Manning Shelter Island.

Web site links

Nael, D. (2019, March 11). Retrieved from okta: https://developer.okta.com/blog/2019/03/11/node-sql-server

Adaptable. (2024). *deploy-nodejs-app*. Retrieved from adaptable: https://adaptable.io/docs/app-guides/deploy-nodejs-app

Dizdar, A. (2022, Jurly 27). *unit-testing-in-nodejs*. Retrieved from Bright: https://brightsec.com/blog/unit-testing-in-nodejs/

Ebere, N. S. (2022, March 3). *how-to-build-a-backend-application*. Retrieved from freecodecamp: https://www.freecodecamp.org/news/how-to-build-a-backend-application

learnthinkcreatecode. (2022, October 25). *How to test a Node.js REST API using postman.*

YouTube. (2020, September 13). *watch?v=VStXlFxQgZg*. Retrieved from youtube: https://www.youtube.com/watch?v=VStXlFxQgZg

YouTube. (2022, October 25). *watch?v=zLbsM_n1H9w*. Retrieved from youtube: https://www.youtube.com/watch?v=zLbsM_n1H9w

RTB | RWANDA TVET BOARD

October 2024