UNIT I

INTRODUCTION TO EXPRESS FRAMEWORK

1.1 INTRODUCTION

The Express framework, often simply referred to as **Express**, is a fast, unopinionated, and minimalist web framework for **Node.js**. It is widely used for building web applications and APIs due to its simplicity, flexibility, and robust set of features.

Key Features of Express:

1. **Minimalist**: Provides a thin layer of fundamental web application features without dictating the application structure or dependencies.
2. **Routing**: Offers a powerful and flexible routing mechanism to handle different HTTP methods and URLs.
3. **Middleware Support**: Middleware functions in Express are used to process requests and responses, making it easier to implement features like logging, authentication, and error handling.
4. **Template Engines**: Supports various templating engines (like Pug, EJS, Handlebars) for rendering dynamic HTML pages.
5. **Robust API**: Provides HTTP utilities such as redirection, caching, and content negotiation.
6. **Extensibility**: Easily integrates with third-party libraries and plugins for added functionality.

Why Use Express?

- **Speed and Efficiency**: Lightweight and fast, making it ideal for high-performance applications.
- **Flexibility**: Doesn't enforce a rigid structure, giving developers the freedom to design their apps as they see fit.
- **Large Ecosystem**: Supported by a vibrant community and an extensive library of middleware for extending functionality.
- **Ease of Learning**: Simpler than other frameworks, especially for developers familiar with JavaScript and Node.js.

Basic Example:

Here’s a simple Express application:

```
// Import the express module
const express = require('express');

// Create an instance of express
const app = express();

// Define a route
app.get('/', (req, res) => {
  res.send('Hello, World!');
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Explanation:

1. **require('express')**: Imports the Express library.
2. **express()**: Creates an Express application instance.
3. **app.get()**: Defines a route to handle GET requests at the root URL (/).
4. **res.send()**: Sends a response back to the client.
5. **app.listen()**: Starts the server on a specified port and listens for incoming requests.

When to Use Express:

- **Building APIs**: RESTful APIs, GraphQL, etc.
- **Web Applications**: Both server-rendered and single-page applications.
- **Middleware Applications**: Handling data streams or interfacing with other systems.

Express is a foundational framework that many developers rely on to build scalable, maintainable, and performant applications.

1.2 RESTFUL SERVICES

REST or Representational State Transfer is an architectural style that can be applied to web services to create and enhance properties like performance, scalability, and modifiability. RESTful web services are generally highly scalable, light, and maintainable and are used to create APIs for web-based applications. It exposes API from an application in a secure and stateless manner to the client. The protocol for REST is HTTP. In this architecture style, clients and servers use a standardized interface and protocol to exchange representation of resources.

REST emerged as the predominant Web service design model just a couple of years after its launch, measured by the number of Web services that use it. Owing to its more straightforward style, it has mostly displaced SOAP and WSDL-based interface design.

REST became popular due to the following reasons:

1. It allows web applications built using different programming languages to communicate with each other. Also, web applications may reside in different environments, like on Windows, or for example, Linux.
2. Mobile devices have become more popular than desktops. Using REST, you don't need to worry about the underlying layer for the device. Therefore, it saves the amount of effort it would take to code applications on mobiles to talk with normal web applications.
3. Modern applications have to be made compatible with the Cloud. As Cloud-based architectures work using the REST principle, it makes sense for web services to be programmed using the REST service-based architecture.

RESTful Architecture:

1. **Division of State and Functionality:** State and functionality are divided into distributed resources. This is because every resource has to be accessible via normal HTTP commands. That means a user should be able to issue the GET request to get a file, issue the POST or PUT request to put a file on the server, or issue the DELETE request to delete a file from the server.
2. **Stateless, Layered, Caching-Support, Client/Server Architecture:** A type of architecture where the web browser acts as the client, and the web server acts as the server hosting the application, is called a client/server

architecture. The state of the application should not be maintained by REST. The architecture should also be layered, meaning that there can be intermediate servers between the client and the end server. It should also be able to implement a well-managed caching mechanism.

Principles of RESTful applications:

1. **URI Resource Identification:** A RESTful web service should have a set of resources that can be used to select targets of interactions with clients. These resources can be identified by URI (Uniform Resource Identifiers). The URIs provide a global addressing space and help with service discovery.
2. **Uniform Interface:** Resources should have a uniform or fixed set of operations, such as PUT, GET, POST, and DELETE operations. This is a key principle that differentiates between a REST web service and a non-REST web service.
3. **Self-Descriptive Messages:** As resources are decoupled from their representation, content can be accessed through a large number of formats like HTML, PDF, JPEG, XML, plain text, JSON, etc. The metadata of the resource can be used for various purposes like control caching, detecting transmission errors, finding the appropriate representation format, and performing authentication or access control.
4. **Use of Hyperlinks for State Interactions:** In REST, interactions with a resource are stateless, that is, request messages are self-contained. So explicit state transfer concept is used to provide stateful interactions. URI rewriting, cookies, and form fields can be used to implement the exchange of state. A state can also be embedded in response messages and can be used to point to valid future states of interaction.

Advantages of RESTful web services:

1. **Speed:** As there is no strict specification, RESTful web services are faster as compared to SOAP. It also consumes fewer resources and bandwidth.
2. **Compatible with SOAP:** RESTful web services are compatible with SOAP, which can be used as the implementation.
3. **Language and Platform Independency:** RESTful web services can be written in any programming language and can be used on any platform.
4. **Supports Various Data Formats:** It permits the use of several data formats like HTML, XML, Plain Text, JSON, etc.

1. Endpoints

HTTP Method	Endpoint	Description
GET	/users	Retrieve all users.
GET	/users/{id}	Retrieve a specific user.
POST	/users	Create a new user.
PUT	/users/{id}	Update a specific user.
DELETE	/users/{id}	Delete a specific user.

2. Example Interaction

- **GET Request**:

  GET /users/123

  Response:

  ```
  {
    "id": 123,
    "name": "John Doe",
    "email": "johndoe@example.com"
  }
  ```

  POST Request:

  ```
  POST /users
  Content-Type: application/json

  {
    "name": "Jane Doe",
    "email": "janedoe@example.com"
  }
  ```

  Response (201 Created):

  ```
  {
    "id": 124,
    "name": "Jane Doe",
    "email": "janedoe@example.com"
  }

  // GET /resource/123
  // Returns the state of the resource with ID 123
  app.get('/resource/:id', function(req, res) {
  ```

```
var id = req.params.id;
var resource = findResourceById(id);
res.json(resource);
});

// POST /resource
// Creates a new resource with the state specified in the request body
app.post('/resource', function(req, res) {
var resource = req.body;
var id = createResource(resource);
res.json({ id: id });
});

// PUT /resource/123
// Updates the state of the resource with ID 123 with the state specified in the request body
app.put('/resource/:id', function(req, res) {
var id = req.params.id;
var resource = req.body;
updateResource(id, resource);
res.sendStatus(200);
});

// DELETE /resource/123
// Deletes the resource with ID 123
app.delete('/resource/:id', function(req, res) {
var id = req.params.id;
deleteResource(id);
res.sendStatus(200);
});
```

Benefits of RESTful Services

- Simplicity and ease of implementation.
- Scalability due to stateless nature.
- Language-agnostic (clients and servers can be written in different programming languages).
- Reusability and standardization.

1.3 INTRODUCTION TO EXPRESS

Express full-stack web development refers to building complete web applications (frontend and backend) using the **Express.js** framework as the server-side technology. It typically involves a combination of technologies to create robust, scalable, and dynamic web applications.

What is Express.js?

Express.js is a minimal and flexible Node.js web application framework. It provides a robust set of features for building web and mobile applications, such as:

- Middleware to handle requests and responses.
- Routing for defining application endpoints.
- Integration with various databases (e.g., MongoDB, PostgreSQL).
- Compatibility with templating engines (e.g., EJS, Pug) or APIs (JSON).

What is Full-Stack Development?

Full-stack development involves working on both the **frontend** (client-side) and **backend** (server-side) of a web application, including database management and deployment.

- **Frontend:** The user interface that interacts with the user (HTML, CSS, JavaScript, frameworks like React, Angular, or Vue.js).
- **Backend:** Handles data processing, server logic, and APIs (e.g., Express.js with Node.js).
- **Database:** Stores application data (e.g., MongoDB, MySQL, or Firebase).

Key Technologies in an Express Full-Stack Setup

1. **Frontend:**
    - **HTML**: Structure of the web page.
    - **CSS**: Styling and layout.
    - **JavaScript**: Interactivity and dynamic content.
    - **Frontend Frameworks/Libraries**: React, Angular, or Vue.js.
2. **Backend:**
    - **Node.js**: JavaScript runtime for building server-side applications.
    - **Express.js**: Framework for creating web servers and APIs.
3. **Database:**

- **MongoDB** (NoSQL) or **PostgreSQL/MySQL** (SQL) for storing and managing data.

4. **Version Control:**
   - **Git**: To manage code versions.
   - **GitHub/GitLab**: For collaboration and deployment.
5. **Tools:**
   - **Postman/Thunder Client**: For API testing.
   - **VS Code**: Popular code editor.
6. **Deployment Platforms:**
   - **Heroku**, **Vercel**, **AWS**, or **DigitalOcean** for hosting applications.

Building Blocks of an Express Full-Stack Application

1. **Frontend Development:**
   - Create a user interface using HTML, CSS, and JavaScript.
   - Use a framework (React, Vue.js) for component-based architecture.
2. **Backend Development:**
   - Use Express.js to set up a server.
   - Define API routes to handle HTTP requests (GET, POST, PUT, DELETE).

   Example Express.js Code:

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

app.use(express.json());

app.get('/api', (req, res) => {
  res.send('Welcome to the Express.js API!');
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

3. **Database Integration:**
   - Connect Express to a database (e.g., MongoDB with Mongoose).
   - Example of connecting to MongoDB:
   - const mongoose = require('mongoose');

```
mongoose.connect('mongodb://localhost:27017/mydatabase', {
    useNewUrlParser: true,
    useUnifiedTopology: true,
}).then(() => {
    console.log('Connected to MongoDB!');
}).catch(err => console.error('MongoDB connection error:', err));
```

4. **Frontend-Backend Communication:**
    - Use **fetch** or **Axios** in the frontend to make API calls to the backend.

    Example in React:

    ```
    fetch('/api')
      .then(response => response.json())
      .then(data => console.log(data))
      .catch(error => console.error('Error:', error));
    ```

5. **Deployment:**
    - Bundle the frontend and backend code.
    - Deploy to a cloud provider like Heroku or AWS.

Advantages of Using Express.js for Full-Stack Development

- **Scalability:** Lightweight and efficient, suitable for large applications.
- **Customizability:** Flexibility to integrate with different tools and frameworks.
- **Speed:** Rapid development with reusable modules.
- **Community Support:** Rich ecosystem of middleware and resources.

Conclusion

Express full-stack development is a powerful way to build web applications, offering a blend of simplicity, flexibility, and performance. By combining Express.js with modern frontend frameworks and databases, developers can create scalable and maintainable applications that deliver great user experiences.

1.4 BUILD YOUR FIRST WEB SERVER

uilding your first web server is an exciting way to dive into web development. Here's a simple guide to help you set up a basic web server. For simplicity, we'll use **Node.js** and its built-in HTTP module, as it provides an easy way to get started.

Prerequisites

1. **Install Node.js**: Download and install Node.js. This will give you access to both node and npm (Node Package Manager).
2. **Code Editor**: Use an editor like VS Code

Steps to Create a Basic Web Server

1. **Create Your Project Directory**:

```
mkdir my-web-server

cd my-web-server
```

2. **Initialize a Node.js Project**:
   npm init -y

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

3. **Create a JavaScript File**: Create a file named server.js in your project directory.

4. **Write Basic Server Code**: Open server.js in your editor and add the following code:

```
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: OK
res.setHeader('Content-Type', 'text/html');
res.end('<h1>Hello, World!</h1>');
});

server.listen(port, hostname, () => {
console.log(`Server running at http://${hostname}:${port}/`);
});
```

5. **Run Your Server**: In your terminal, run

node server.js

Server running at http://127.0.0.1:3000/

6. **Access the Server**: Open your browser and go to http://127.0.0.1:3000/.

You should see "Hello, World!" displayed

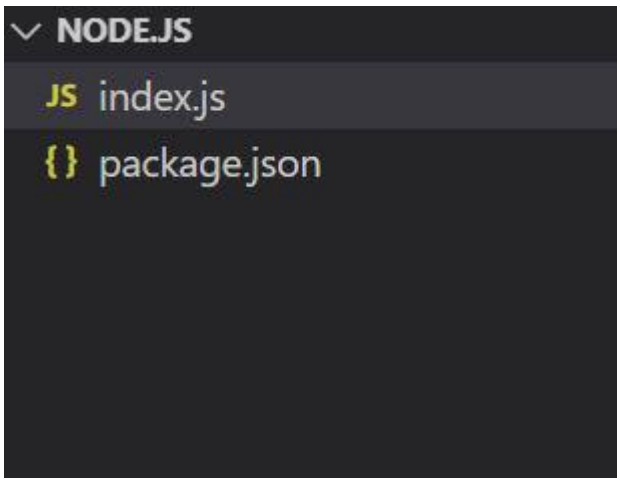
Using Built-in HTTP module

HTTP and HTTPS, these two inbuilt modules are used to create a simple server. The HTTPS module provides the feature of the encryption of communication with the help of the secure layer feature of this module. Whereas the HTTP module doesn't provide the encryption of the data.

Approach

Building a simple Node.js web server with the http module by using http.createServer(), which listens for requests, sends responses, and is ideal for understanding core server functionality.

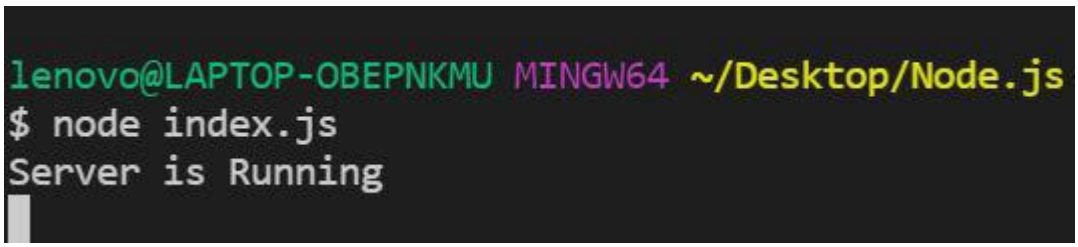
Project structure: It will look like this.



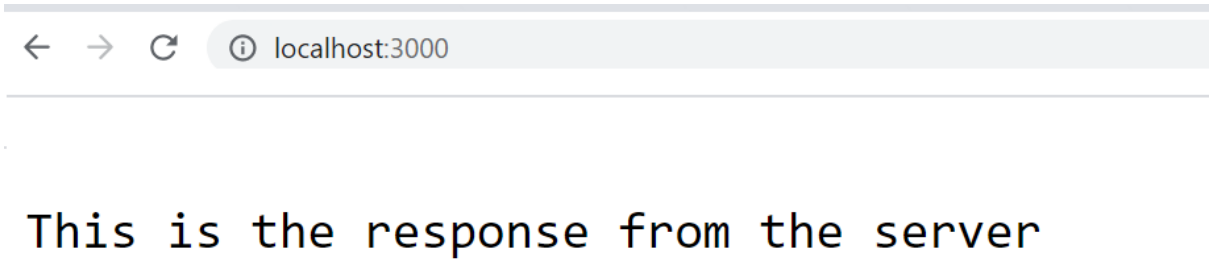

```
// Filename - index.js

// Importing the http module
const http = require("http")

// Creating server
const server = http.createServer((req, res) => {
    // Sending the response
    res.write("This is the response from the server")
    res.end();
})

// Server listening to port 3000
server.listen((3000), () => {
    console.log("Server is Running");
})
```

Run **index.js** file using below command:

node index.js



utput: Now open your browser and go to *http://localhost:3000/*, you will see the following output:



This is the response from the server

Using Express Module

The *express.js* is one of the most powerful frameworks of the node.js that works on the upper layer of the http module. The main advantage of using *express.js* server is filtering the incoming requests by clients.

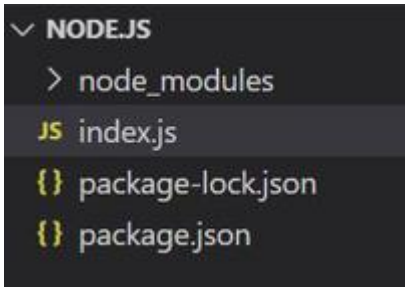
Approach

To create a web server with Express initialize an app with express(), defining routes with app.get(), and sending responses using res.send(). Express simplifies development with built-in features and middleware.

Installing module: Install the required module using the following command.

npm install express

Project structure: It will look like this.



Example:This example demonstrates creating a simple web server using express.js

```
// Filename - index.js


// Importing express module

const express = require("express")

const app = express()


// Handling GET / request

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

    res.send("This is the express server")

})


// Handling GET /hello request

app.get("/hello", (req, res, next) => {

    res.send("This is the hello response");

})


// Server setup
```

```
app.listen(3000, () => {
    console.log("Server is Running")
})
```

Run the index.js file using the below command:

```
node index.js
```

Output: Now open your browser and go to *http://localhost:3000/*, you will see the following output:

1.5 NODEMON

Nodemon is a popular tool that is used for the development of applications based on node.js. It simply restarts the node application whenever it observes the changes in the file present in the working directory of your project.

Additionally, nodemon does not need any specific modifications to code or the mode of development. It acts as a facilitator in the node by replacing the wrapper for it. To use nodemon, you will simply need to replace the word node on the CLI while you are about to execute your script.

Installation

To carry out the installation of Nodemon in your node.js-based project use the following steps for your reference.

```
npm install -g nodemon
```

Although you can clone nodemon from Git but the above is a better recommendation. The above command will install the nodemon globally into your system. Also, you can install further dependencies which are highly recommended for a smooth workflow using the below command.

```
npm install --save-dev nodemon
```

Note: Remember that installing nodemon locally, would not be available in your system path rather you need to run nodemon locally by calling it from the npm script with the command npx nodemon or npm start.

Creating a Node project

To create your node project, all you need to do is to follow the below very crucial steps.

First, make a directory with the below command as shown.

1. mkdir nodemon -exp

After making a directory, you will need to initialize the package.json using the command.

1. npm init -y

Now, all you have to do is to install express using the below command whether you have yarn or node installed in your system.

yarn add express

r

npm install express --save

The next step is to add some plain Javascript and to do that you need to use this Javascript to configure the express through node for making the port connected to the server so that results are displayed on the console. Use the below sample code to configure your server with a listener port.

```
// server.js file


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

App.use('/', (req, res) =>  {

 res.status(300).send('Hello JavaTpoint!');

});

app.listen(8080);
```

In the above code, we have established our port to listen to port number 8080. The res and req are defined as the response and the request that would be listened through the port.

The next step is to start the server using the below command.

nodemon server

On starting the server, it may look something like this.

Now, if you change inside the server.js file, the server will automatically restart, and you can get the latest output on the browser.

If you make any changes in the server.js file, it will automatically be reflected and the server will restart and the latest output will be displayed on the browser.

Conversely, nodemon will search for files driven out by your project and will look for a primary package.json file to start the script. Also, you can either opt for launching nodemon directly for your application by writing the start script in the package.json file. The package.json file may have some dependencies as shown below.

```
{
  "name": "nodemon-exp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon server"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.3"
  }
}
```

Now, all you need to do is to hit the command as shown below.

npm start

The above command will serve the same thing as discussed.

Options with Nodemon

1. **-exec**

This option is used primarily to specify the binary executable objects with the associated files. For instance, while combining the TypeScript files, it will run binary execution methods to reflect changes in the dependencies.

1. **--ignore**

This option is used to ignore files or specific patterns in your working directory.

1. **-ext**

This option is used to notify the watchable file extensions. For this, you need to specify it comma separation like --ext js, ts.

1. **--delay**

This option is used to set intervals or make delays in terms of seconds.

1. **--watch**

This option is used to objectify multiple directories associated with watch. Using this option will let you specifically watch the subdirectories or files constituting your project's data.

1. **--verbose**

This option is primarily used to display the changes that have been made while starting the nodemon server.

Nodemon is a utility that helps develop Node.js-based applications by automatically restarting the application when file changes in the directory are detected. It is particularly useful during development to streamline the workflow and avoid manual restarts of the server after each change.

Key Features

- Automatically restarts the Node.js application when changes are detected.
- Configurable to watch specific files, extensions, or directories.
- Supports custom scripts and events.
- Can be used globally or as a project dependency.

Installation

Install Globally:

npm install -g nodemon

This allows you to use the nodemon command anywhere in your system.

Install as a Project Dependency:

npm install --save-dev nodemon

This ensures nodemon is available only within the project.

Usage

Start a Node.js Application:

nodemon app.js

This runs the app.js file and monitors for changes.

Custom Script Execution:

nodemon --exec "npm run my-script"

Specify a Different File:

nodemon server.js

Watch Specific Files or Extensions:

nodemon --watch src --ext js,html

Configuration

You can create a nodemon.json file in the root of your project for custom configurations:

```
{
  "watch": ["src"],
  "ext": "js,json",
  "ignore": ["node_modules/*"],
  "exec": "node server.js"
}
```

Common Commands

- **Restart manually:** Type rs in the terminal where nodemon is running.
- **Ignore certain files or directories:**
- nodemon --ignore logs/*
- **Run with a specific delay:**
- nodemon --delay 2

Nodemon is simple to use and can significantly improve development efficiency in Node.js applications.

1.6 ENVIRONMENTAL VARIABLES

In full-stack web development, **environmental variables** are used to manage configuration settings that can change between different environments, such as development, staging, and production. They provide a secure and flexible way to store sensitive information and project-specific settings. These variables are typically stored outside the source code, so they can be easily changed without modifying the actual codebase.

Key Purposes of Environmental Variables:

1. **Configuration Management**: Different environments may have different settings, such as database URLs, API keys, or service credentials. Environmental variables allow you to switch configurations based on the environment (development, testing, production).
2. **Security**: Sensitive data like passwords, API keys, or tokens should never be hard-coded into the application code. Storing such data in environment variables helps keep it secure.
3. **Flexibility**: Instead of hard-coding values in the codebase, you can use environment variables to dynamically configure your application, making it more flexible and reusable across different systems.
4. **Environment-Specific Behavior**: Environmental variables allow you to adjust the behavior of your application in different stages of development. For example, in a production environment, you might want to enable logging, but in a development environment, you might want to suppress logs to reduce noise.

Common Use Cases:

1. **Database Connection Strings**: You might have a different database URL or credentials for development and production environments.
    - Example: DB_HOST=localhost or DB_HOST=prod.db.server
2. **API Keys**: Store keys for third-party services like Google Maps, Stripe, or SendGrid.
    - Example: GOOGLE_API_KEY=your_api_key_here
3. **Secret Keys or Tokens**: Store sensitive tokens, such as JWT secret keys.
    - Example: JWT_SECRET=mysecretkey
4. **Port Configuration**: Specify the port number the server will listen on.
    - Example: PORT=3000

Working with Environmental Variables:

1. In Node.js (JavaScript Backend):

- Use the process.env object to access environmental variables. You can load them using packages like dotenv in development.

```
require('dotenv').config(); // loads .env file
const dbHost = process.env.DB_HOST;
```

.env file:

```
DB_HOST=localhost
DB_PORT=5432
API_KEY=your_api_key_here
```

2. In React (Frontend):

- React supports environmental variables by prefixing them with REACT_APP_ for them to be embedded into the build. **.env** file:

```
REACT_APP_API_URL=https://api.example.com
REACT_APP_ENV=production
```

Then access them in the React code:

```
const apiUrl = process.env.REACT_APP_API_URL;
```

3. In Deployment:

- In production environments, such as on cloud platforms (e.g., Heroku, AWS), you can set environmental variables through the platform's dashboard or using CLI commands.

4. In Docker:

- When using Docker, you can pass environment variables through the docker-compose.yml file or directly in the Dockerfile.

```
environment:
  - DB_HOST=localhost
  - DB_PORT=5432
```

Best Practices:

1. **Never commit .env files**: Ensure .env files are listed in .gitignore to prevent them from being pushed to version control systems.

2. **Use default values**: Provide fallback values for development environments, or use default values directly in the code if the environment variable is not set.
3. **Use different configurations for each environment**: Create multiple .env files like .env.dev, .env.prod for different environments.
4. **Secure storage in production**: For production, store environmental variables securely (e.g., via environment management in cloud services or using secret management tools like AWS Secrets Manager).

By handling configurations through environmental variables, you can ensure better security, maintainability, and scalability for your web applications.

1.7 ROUTE PARAMETERS

Router parameters are dynamic values that are used to pass information through URLs, typically in web applications. They are commonly used in frameworks like Express (Node.js), React Router (for client-side routing), and others.

There are two main types of router parameters:

1. **Route Parameters (Path Parameters)**: These are part of the URL path and are typically represented by a colon (:) followed by a variable name. For example:
    - In Express (Node.js):
    - app.get('/user/:id', (req, res) => {
    - const userId = req.params.id;
    - res.send(`User ID is ${userId}`);
    - });
    - In React Router:
    - <Route path="/user/:id" component={User} />

Here, :id is a route parameter that can be accessed in your route handler.

2. **Query Parameters**: These are passed in the URL after a ? symbol and typically come in key-value pairs, like ?key=value. They are often used for filtering or pagination. For example:
    - In Express:
    - app.get('/search', (req, res) => {
    - const query = req.query.q; // Retrieves value of 'q'
    - res.send(`Searching for: ${query}`);
    - });
    - In React Router:
    - // The query string could look like /search?q=example

- const queryParams = new URLSearchParams(location.search);
- const query = queryParams.get('q');

Both types allow developers to manage dynamic content and handle user input more efficiently in their web applications.

1.8 HANDILING HTTP GET REQUEST

Handling an HTTP GET request is a common task in web development, usually performed by web servers or frameworks to respond to client requests. Here's how you can handle GET requests in various programming languages or web frameworks.

GET requests are commonly used for retrieving data from a server. In Express.js, you can handle GET requests using the app.get() method. Here's an example:

```
// Handling a GET request
app.get('/api/users', (req, res) => {
    // Simulated user data
    const users = [
        { id: 1, name: 'John' },
        { id: 2, name: 'Alice' },
        { id: 3, name: 'Bob' }
    ];

    // Sending JSON response with user data
    res.json(users);
});
```

In this example, when a GET request is made to '/api/users', the server responds with JSON data containing a list of users.

Key Steps for Handling GET Requests:

1. **Define a route** that maps to the URL where the request is made.
2. **Specify the HTTP method** (GET in this case).
3. **Return the desired response** when the route is accessed.

1.9 HANDLING AN HTTP POST REQUEST

Handling an HTTP POST request involves setting up a server that can receive, process, and respond to requests made by clients (like web browsers, applications, or other servers). Here's a simple breakdown of how to handle an HTTP POST request:

Basic Steps:

1. **Set Up a Server**: You need a web server or framework that listens for HTTP requests. In many programming languages, libraries or frameworks can make this process easier. For example, in Python, Flask or Django can be used; in Node.js, Express is commonly used.
2. **Configure the Endpoint**: You need to define an endpoint in your server that will handle the POST request. A POST request typically sends data to the server (e.g., form data, JSON, etc.).
3. **Extract the Data**: POST requests usually include data in the request body (such as form data or JSON). You'll need to extract and process this data.
4. **Respond to the Request**: After processing the POST request, you send a response back to the client, often with some kind of status message or result of the request.

Example in Python (Flask):

```
from flask import Flask, request, jsonify


app = Flask(__name__)


@app.route('/submit', methods=['POST'])
```

```
def handle_post_request():
    # Extracting JSON data from the request
    data = request.get_json()

    # Process the data (e.g., save to a database, perform calculations, etc.)
    # In this case, just print the received data
    print("Received data:", data)

    # Send a response back to the client
    return jsonify({"status": "success", "data_received": data})

if __name__ == '__main__':
    app.run(debug=True)
```

Explanation:

- @app.route('/submit', methods=['POST']): Defines the route that handles POST requests.
- request.get_json(): Extracts JSON data from the body of the POST request.
- jsonify(): Sends back a JSON response.

Example in JavaScript (Node.js with Express):

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

// Middleware to parse JSON bodies
```

```
app.use(express.json());


app.post('/submit', (req, res) => {
    const data = req.body;


    // Process the data (e.g., log it, save it to a database)
    console.log("Received data:", data);


    // Send back a JSON response
    res.json({ status: "success", data_received: data });
});


app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Explanation:

- app.use(express.json()): Middleware to parse incoming JSON data.
- req.body: Accesses the parsed body data from the POST request.
- res.json(): Sends a JSON response back to the client.

Common POST Request Body Formats:

1. **JSON**: Often used in APIs. The body contains a JSON string.
2. **Form Data**: Often used in web forms (application/x-www-form-urlencoded or multipart/form-data).

In Flask, for form data:

```
@app.route('/submit', methods=['POST'])

def handle_post_request():

    name = request.form['name']

    email = request.form['email']

    return jsonify({"name": name, "email": email})
```

In Express, for form data:

```
app.use(express.urlencoded({ extended: true }));

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

    const { name, email } = req.body;

    res.json({ name, email });

});
```

This gives a basic overview of handling POST requests. Depending on your use case, you can customize this to handle authentication, validate data, or save to a database.

1.10 CALLING ENDPOINTS USING POSTMAN

1. What is Postman?

Postman is a popular API testing tool that allows you to send HTTP requests to interact with APIs. It simplifies testing RESTful APIs by providing a user-friendly interface and various features for request creation, testing, and automation.

2. Setting Up Postman

- Download and install Postman from the official website: Postman Download.
- Open Postman after installation.

3. Making API Requests

Follow these steps to call an endpoint in Postman:

Step 1: Create a New Request

- Open Postman.
- Click the **New** button on the left sidebar.
- Choose **Request** from the available options.

Step 2: Set Request Method

- In the **Request** window, select the HTTP method you want to use (GET, POST, PUT, DELETE, PATCH, etc.) from the dropdown next to the URL bar.
    - **GET**: Retrieve data.
    - **POST**: Send data to the server.
    - **PUT**: Update data on the server.
    - **DELETE**: Remove data from the server.

Step 3: Enter URL (Endpoint)

- In the URL field, enter the API endpoint URL you want to call. For example, https://api.example.com/v1/resource.

Step 4: Add Headers (if needed)

- If your API requires headers (e.g., Content-Type, Authorization), go to the **Headers** tab.
- Add necessary headers:
    - **Content-Type**: Specifies the format of the request body (e.g., application/json).
    - **Authorization**: If the API requires a token, add the token in the format Bearer <Token>.

Step 5: Add Body (for POST, PUT, PATCH requests)

- If you're making a POST, PUT, or PATCH request that requires data, go to the **Body** tab.
- Choose the appropriate format for the body:
    - **raw**: For sending raw JSON or other text formats.
    - **form-data**: For sending form-based data (multipart).
    - **x-www-form-urlencoded**: For sending URL-encoded key-value pairs.

Example for JSON Body:

```
{
  "username": "john_doe",
  "password": "12345"
}
```

Step 6: Send the Request

- After entering the necessary information, click the **Send** button to send the request to the API.

Step 7: View the Response

- Once the request is sent, Postman will show the response in the bottom section.
    - **Status**: The HTTP status code (e.g., 200 OK, 404 Not Found).
    - **Body**: The response body, often in JSON or XML format.
    - **Headers**: The headers returned by the server.

4. Additional Postman Features

a. Environment Variables

Postman allows you to set environment variables to reuse across requests.

- Create a new environment via the gear icon in the top-right corner.
- Add variables like {{base_url}} and use them in requests, e.g., {{base_url}}/v1/resource.

b. Authentication

- **Bearer Token**: If the API requires authentication via a token, you can enter it under the **Authorization** tab in Postman.
    - Select **Bearer Token** from the dropdown and paste your token in the text field.
- **Basic Authentication**: For username and password authentication, use **Basic Auth** under the **Authorization** tab.

c. Pre-request Scripts

You can write JavaScript code under the **Pre-request Script** tab that will execute before the request is sent. This can be useful for setting dynamic variables.

d. Tests

You can write tests to validate the response of the API call.

- Go to the **Tests** tab.
- Example: Check if the response status code is 200.
- pm.test("Status code is 200", function() {
- pm.response.to.have.status(200);
- });

e. Collection Runner

You can run a collection of API requests in sequence, passing data from one request to another (via environment variables or data files). This is useful for testing multiple endpoints in an automated way.

5. Common HTTP Status Codes

- **200 OK**: The request was successful.
- **201 Created**: The resource was created successfully (usually for POST requests).
- **400 Bad Request**: The request was invalid.
- **401 Unauthorized**: The request requires authentication.
- **403 Forbidden**: The server understands the request, but refuses to authorize it.
- **404 Not Found**: The resource could not be found.
- **500 Internal Server Error**: The server encountered an error.

6. Exporting and Sharing Requests

- You can save and export your requests by clicking the **Save** button.
- Share collections by exporting them to a file and sharing them with teammates.

7. Automating Requests

- Postman provides tools for automating API requests and running tests using **Newman**, the Postman CLI tool. This allows you to execute your

Postman collections via the command line or integrate them into CI/CD pipelines.

8. Best Practices

- **Use Variables**: To keep your tests flexible and reusable, use environment or collection variables.
- **Test with Real Data**: Always use realistic test data and handle edge cases.
- **Write Tests**: Ensure your API responses match expectations by writing automated tests in Postman.

Conclusion

Postman is a versatile tool for making API requests and testing responses. By leveraging its various features like environments, authentication, and automated testing, you can streamline your API development and testing processes.

1.11 INPUT VALIDATION

Definition: Input validation refers to the process of ensuring that the data provided to a program or system is both correct and secure. This step is essential in preventing unexpected behaviors, errors, and vulnerabilities in an application. Proper input validation protects against malformed data, attacks like SQL injection, and ensures data integrity.

Types of Input Validation:

1. **Client-Side Validation:**
    - Performed in the user's browser, before the data is sent to the server.
    - It provides immediate feedback to the user but can be bypassed by malicious users.
    - Often implemented using JavaScript or HTML5 validation attributes.
    - Example: Checking if an email address is in the correct format before submitting a form.
2. **Server-Side Validation:**
    - Performed on the server after the data has been received from the client.
    - More secure than client-side validation as it cannot be bypassed by altering the client-side code.

    - Example: Ensuring that a user's password is of an acceptable length and complexity.
3. **Database-Level Validation:**
    - Enforced within the database itself, such as through constraints, triggers, or stored procedures.
    - Ensures data integrity and accuracy directly at the data storage level.

Key Techniques in Input Validation:

1. **Whitelisting (Allow-listing):**
    - Accept only data that matches a predefined set of criteria or patterns.
    - Recommended approach as it minimizes the risk of unexpected data being processed.
    - Example: If accepting an email address, ensure the string matches the pattern someone@domain.com (using regular expressions).
2. **Blacklisting (Deny-listing):**
    - Reject data that matches known bad patterns or values.
    - Not as secure as whitelisting, since new or unknown malicious inputs may not be detected.
    - Example: Rejecting inputs that contain SQL keywords (like DROP, DELETE).
3. **Length Validation:**
    - Check that the input is neither too short nor too long for the expected format.
    - Prevents buffer overflows and ensures the data fits within the expected size for efficient processing.
    - Example: Username should be at least 5 characters and no more than 20.
4. **Type Validation:**
    - Ensure the input matches the expected data type.
    - Example: If a field expects an integer, input validation checks if the value is an integer (not a string or float).
5. **Format Validation:**
    - Checks whether the input conforms to a specific pattern or format, such as email, date, phone number, etc.
    - Example: Ensuring a phone number is in the format (XXX) XXX-XXXX.
6. **Range Validation:**
    - For numerical or date values, validate that they fall within a permissible range.

    - Example: Age must be between 18 and 120 years.
7. **Cross-Site Scripting (XSS) Prevention:**
    - Validate user inputs to prevent malicious scripts from being executed on the client side.
    - Often achieved by sanitizing input, stripping or encoding harmful characters like <, >, and &.
8. **SQL Injection Prevention:**
    - Ensure that inputs do not contain SQL commands or special characters that can manipulate SQL queries.
    - Example: Using prepared statements with parameterized queries instead of directly concatenating inputs into SQL queries.
9. **Escape Special Characters:**
    - Escape characters that have special meanings in certain contexts, like HTML, JavaScript, or SQL.
    - Example: Replacing & with & in HTML to prevent XSS.

Best Practices for Input Validation:

1. **Validate on Both Client-Side and Server-Side:**
    - Client-side validation provides a good user experience but can be bypassed.
    - Server-side validation ensures security and data integrity.
2. **Avoid Relying Solely on Front-End Validation:**
    - Malicious users can disable JavaScript or modify the client-side code. Always validate on the server side as well.
3. **Use Strong Data Types and Structures:**
    - Ensure that inputs are typed properly, such as integers, booleans, etc., and are not arbitrary strings.
    - Example: Use int for age or quantity instead of allowing any input.
4. **Limit Input Length:**
    - Minimize risk by limiting input lengths to reasonable values based on the field requirements.
    - Example: A phone number should not exceed 15 characters.
5. **Apply Contextual Validation:**
    - Validate inputs based on the context they are used in.
    - Example: A ZIP code input should only accept numbers or a specific pattern depending on the country.
6. **Use Regular Expressions for Complex Pattern Matching:**
    - Regular expressions are useful for validating patterns like email addresses, URLs, phone numbers, etc.
    - Example: ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ for email.

7. **Error Handling:**
    - Ensure that validation errors are communicated clearly to the user, with appropriate error messages.
    - Example: "Please enter a valid email address" rather than a vague "Invalid input."
8. **Use Libraries and Frameworks:**
    - Many programming languages have built-in libraries for input validation.
    - Examples: Python's re module, JavaScript's built-in RegExp, or using validation libraries in web frameworks like Django or Laravel.
9. **Test Input Validation Thoroughly:**
    - Perform extensive testing to cover edge cases and unexpected inputs.
    - Examples: Empty strings, extremely long inputs, or malformed data.

Common Input Validation Vulnerabilities:

1. **Buffer Overflow Attacks:**
    - Occur when an input exceeds the allocated memory buffer and overwrites adjacent memory, potentially allowing the execution of arbitrary code.
    - Prevented by validating input size and using safe string-handling functions.
2. **SQL Injection:**
    - Occurs when user input is incorporated directly into a SQL query without proper validation or escaping.
    - Prevented using prepared statements and parameterized queries.
3. **Cross-Site Scripting (XSS):**
    - Malicious code injected into web applications, often via input fields, that gets executed on the client side.
    - Prevented by sanitizing inputs and encoding output.
4. **Command Injection:**
    - Happens when user input is executed as a system command.
    - Prevented by properly sanitizing and validating input, especially when interacting with the operating system.

Common Validation Techniques and Their Applications:

1. **Regular Expressions (Regex):**

- Regex is a powerful tool to validate inputs such as email addresses, phone numbers, etc.
- Example: Email validation with regex in Python:
- import re
- def validate_email(email):
- pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
- return re.match(pattern, email)

2. **Whitelisting with Set of Accepted Inputs:**
   - Ensure inputs match a set of predefined acceptable values.
   - Example: A country field might only accept values from a set of supported country codes.
3. **Sanitization:**
   - The process of cleaning data before using it, especially to prevent code injection attacks like XSS.
   - Example: Removing or encoding HTML special characters to prevent code execution.

Conclusion:

Input validation is a crucial aspect of secure programming, and proper validation techniques can mitigate a wide range of security risks. Both client-side and server-side validation, when implemented correctly, help ensure that only valid and secure data enters the system, protecting both users and systems from malicious activity.

1.12 HTTP PUT REQUESTS

To handle an HTTP PUT request, you typically need to set up a route in your web framework that listens for PUT requests at a specific endpoint. This request method is generally used to update an existing resource on the server. Here's a basic example of how to handle a PUT request in a few popular web frameworks.

Express (Node.js)

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

const app = express();



app.use(express.json()); // For parsing JSON payload
```

```
app.put('/update-resource/:id', (req, res) => {
    const resourceId = req.params.id; // Retrieve the resource ID from the URL
    const updatedData = req.body; // Get the updated data from the request body

    // Simulate updating a resource
    // You would typically find the resource by its ID and update it in the database
    console.log(`Updating resource with ID: ${resourceId}`);
    console.log('Updated data:', updatedData);

    res.status(200).json({ message: 'Resource updated successfully' });
});

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

Flask (Python)

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/update-resource/<int:id>', methods=['PUT'])
```

```
def update_resource(id):
    updated_data = request.json  # Get the updated data from the request body

    # Simulate updating a resource
    # You would typically find the resource by its ID and update it in the database
    print(f"Updating resource with ID: {id}")
    print('Updated data:', updated_data)

    return jsonify(message="Resource updated successfully"), 200

if __name__ == '__main__':
    app.run(debug=True)
```

Django (Python)

```
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
import json

@csrf_exempt  # Disable CSRF for this example, you should handle CSRF in production
def update_resource(request, id):
    if request.method == 'PUT':
        # Parse the JSON body
```

```
    updated_data = json.loads(request.body)

    # Simulate updating a resource
    # Typically, you would query your database and update the record here
    print(f"Updating resource with ID: {id}")
    print('Updated data:', updated_data)

    return JsonResponse({'message': 'Resource updated successfully'}, status=200)

# In your urls.py, you would map this view to a URL like:
# path('update-resource/<int:id>/', views.update_resource),
```

ASP.NET (C#)

```
using Microsoft.AspNetCore.Mvc;

[Route("api/[controller]")]
[ApiController]
public class ResourceController : ControllerBase
{
    [HttpPut("{id}")]
    public IActionResult UpdateResource(int id, [FromBody] Resource updatedData)
    {
```

```
        // Simulate updating a resource

        // Typically, you would find the resource by its ID and update it in your
database

        Console.WriteLine($"Updating resource with ID: {id}");

        Console.WriteLine($"Updated data: {updatedData}");



        return Ok(new { message = "Resource updated successfully" });

    }

}
```

Key points when handling PUT requests:

- **URL Parameters**: The resource identifier (e.g., id) is usually provided as part of the URL.
- **Request Body**: The data to update the resource with is sent in the request body (commonly in JSON format).
- **Response**: The server typically responds with a success message and a 200 OK or a similar success status. If the resource doesn't exist, a 404 Not Found response can be returned.

1.13 HANDLING A DELETE REQUEST:

To handle an HTTP DELETE request, you typically need a web framework or server-side code that can process incoming requests. Below is an example in various programming languages for handling DELETE requests.

1. Node.js (Express.js)

In Express, you can handle a DELETE request like this:

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

const app = express();
```

```
app.delete('/resource/:id', (req, res) => {
    const resourceId = req.params.id;
    // Logic to delete the resource by its ID
    console.log(`Resource with ID ${resourceId} deleted`);

    res.status(200).send(`Resource with ID ${resourceId} deleted successfully`);
});

app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

In this example:

- The DELETE method is mapped to the /resource/:id route.
- The :id parameter represents the ID of the resource to be deleted.
- The server responds with a success message once the resource is deleted.

2. Python (Flask)

In Flask, you can handle DELETE requests as follows:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/resource/<int:id>', methods=['DELETE'])
def delete_resource(id):
```

```
    # Logic to delete the resource by its ID
    print(f"Resource with ID {id} deleted")

    return jsonify(message=f"Resource with ID {id} deleted successfully"), 200

if __name__ == '__main__':
    app.run(debug=True)
```

Here:

- The DELETE method is defined for the /resource/<int:id> route, where id is the resource identifier.
- After deletion, the server returns a JSON message confirming the deletion.

3. Java (Spring Boot)

In Spring Boot, you can handle DELETE requests with a method in a controller:

```
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ResourceController {

    @DeleteMapping("/resource/{id}")
    public String deleteResource(@PathVariable("id") Long id) {
        // Logic to delete the resource
```

```
        System.out.println("Resource with ID " + id + " deleted");

        return "Resource with ID " + id + " deleted successfully";

    }

}
```

This example:

- Uses the @DeleteMapping annotation to map the HTTP DELETE request.
- @PathVariable is used to extract the id from the URL.

4. PHP

In PHP, you would handle the DELETE request manually, often using a router or framework. Here's a simple example:

```
<?php



if ($_SERVER['REQUEST_METHOD'] == 'DELETE') {

    $id = $_GET['id'];  // Assuming the ID is passed as a query parameter

    // Logic to delete the resource by its ID

    echo "Resource with ID $id deleted successfully";

}

?>
```

General Steps for Handling a DELETE Request:

1. **Define the Route**: Ensure you have a route that responds to the HTTP DELETE method.
2. **Extract Data**: Extract any necessary data (like resource ID) from the request path, query, or body.
3. **Delete Resource**: Perform the logic required to delete the resource from the database or storage.

4. **Respond**: Send an appropriate response back to the client, typically with a status code (e.g., 200 OK for success or 404 if the resource is not found).

These are basic examples, and in production environments, you would add more features like authentication, validation, and error handling to ensure your DELETE requests are secure and robust.