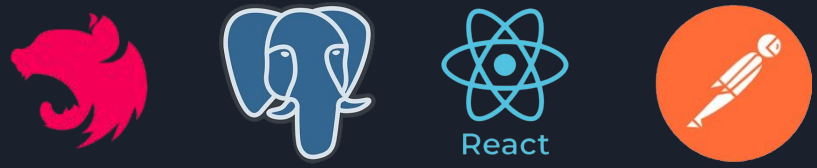
Full Stack Development BootCamp - Day 1

Build Real-World Apps from Backend to Frontend!



Trainer:

Muhammad Maaz Sheikh
Technical Team Lead

Tech Stack Covered:

- Nest Js
- PostgreSQL
- React Js
- Real world project development

Organized By:

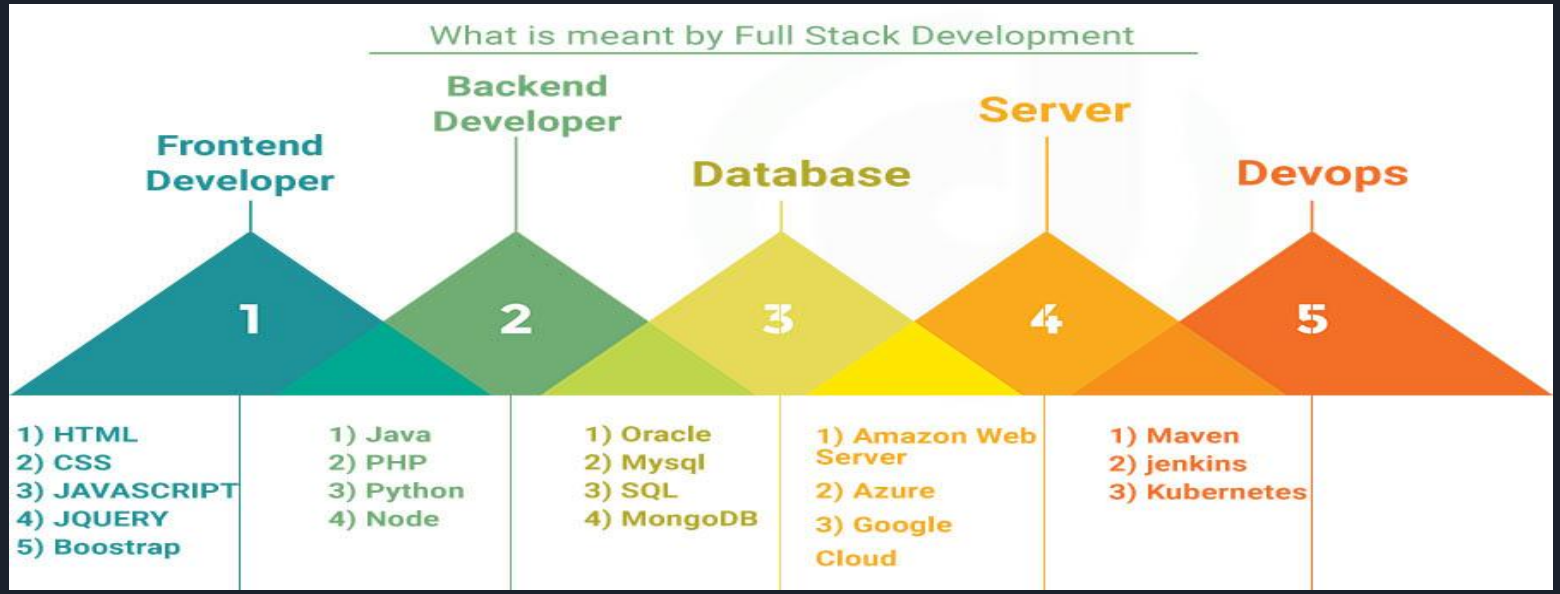
Iqra University

Introduction

Full Stack Development

What is Full Stack Development?

Full Stack Development refers to the practice of working on both the front-end and back-end aspects of a web application.



Who is Full Stack Developer?

- A Full Stack Developer is proficient in multiple programming languages and frameworks, allowing them to handle all aspects of the development process.

- Full Stack Development offers a holistic approach to development, ensuring seamless integration between different layers of an application.

- A Full Stack Developer is someone who is capable of taking an idea from inception to a usable working product, all by himself

Front End Technologies

Front-End technologies in full stack development includes:

- ➔ Html, Html5
- ➔ Css, Css3, Tailwind
- ➔ JQuery, React Js, Angular Js, Angular, Vue, Ember, Vanilla
- ➔ Bootstrap and media queries

Back End Technologies

Back-end Development focuses on the server-side logic and database management of a web application.

Back-End technologies in full stack development includes:

- ➔ Next Js/Express Js/Node Js
- ➔ .Net
- ➔ Java
- ➔ Ruby on Rails
- ➔ Django
- ➔ PHP (Laravel)

Database Management

Full Stack Developers are responsible for managing databases and ensuring efficient data storage and retrieval.

They work with database management systems such as MySQL, PostgreSQL, Redis or MongoDB.

Full Stack Developers design and optimize database structures, allowing for seamless data manipulation and retrieval.

Table of Content

Section A-1: Backend Development Nest Js

What is Nest JS?

What is Nest JS?

Nest Js is framework for building node.Js server side application

Nest Js is build with and fully support of TypeScript

Allow developers to create testable, scalable, loosely coupled and easily maintainable applications

The architecture of Nest Js is heavily inspired by the Angular

Setting up the Node Js (Installation Guide)

Setting up the Node Js (Installation Guide)

Step 1: Check to see if Node.js is already installed

To see if Node is installed, open the Windows Command Prompt, Powershell or a similar command line tool, and type “node -v”.
Note: This should print the version number so you’ll see something like this v0. 10.35.

S**tep 2:** If Node.js isn’t installed, download the windows installer from the official node.js website.

S**tep 3:** Run the Installer (the .msi file that was previously downloaded)

S**tep 4:** Follow the prompts in the Node.js Setup Wizard.

S**tep 5:** Restart your computer (you won’t be able to run node.js until computer restarts)

Setting up the Node Js (Installation Guide)

Step 6: Test Node.Js and NPM

Step 6.1: Open your command line tool and type “node -v”. If installed successfully, the current version number will be printed. (Current version: v18.16.1)

Step 6.2: Test NPM In your command line tool, type “npm -v”. If installed successfully, the current version number will be printed. (Current version: 9.5.1)

Step 6.3: **Create & Run Test File** - Create a new file called “**hello.js**”. Add the following line of code to your js file “**console.log(‘Node is installed’);**”. To run the code, open your command line program, navigate to the folder where your program is saved, and type “**node hello.js**”. This command will start node.js and run the code you saved in the file.

Setting up the Nest Js (Installation Guide)

Setting up the Nest Js (Installation Guide)

- ➔ To start a new Nest JS project, you can use the Nest CLI:
- ➔ Start by installing the Nest JS CLI globally on your local machine and also creating a new Project

```
$ npm install -g @nestjs/cli
$ nest new project-name
```

- ➔ This will create a basic project structure with a default app module.
- ➔ Open your command line tool and type “nest -v”. If installed successfully, the current version number will be printed. (Current version: v11.0.5)

```
C:\Windows\System32>nest -v
11.0.5
```

Understanding its Architecture

Understanding its architecture (Modules, Controllers, Services)

- ➔ The **project-name** directory will be created, node modules and a few other boilerplate files will be installed, and a src/ directory will be created and populated with several core files.
- ➔ Here's a brief overview of those core files:

`app.controller.ts`	A basic controller with a single route.
`app.controller.spec.ts`	The unit tests for the controller.
`app.module.ts`	The root module of the application.
`app.service.ts`	A basic service with a single method
`main.ts`	The entry file of the application which used the core function **NestFactory** to create a Nest application instance and starts the server

Understanding its architecture (Modules, Controllers, Services)

- **Modules:** Modules serve as containers for organizing code based on business domains or specific functionalities. They can be standalone or nested within other modules, creating a hierarchical structure for the application.

- **Controllers:** Controllers handle incoming requests and return an appropriate response. They can be seen as the entry point to your application, where different routes are defined and mapped to specific functions.

- **Providers:** Providers are used to inject dependencies into controllers or other providers. That allows for better code reusability and testability and promotes the principle of separation of concerns.

Understanding its architecture (Pipes, Guards, Filters)

- **Pipes:** Pipes are used for data transformation and validation, allowing for cleaner and more efficient code. They can be applied to any input or output of a controller function, making it easier to handle errors and validate user input.

- **Guards:** Guards are used for authorization and access control, allowing you to restrict access to certain routes based on user roles or permissions. They can also be used for data transformation and validation, making them a powerful tool for securing your application.

- **Filters:** Filters are used for handling exceptions and errors within your application. They can be applied globally or scoped to specific controllers or routes, providing a centralized way of handling errors and improving the overall robustness of your application.

Understanding its architecture (Interceptors, Middleware)

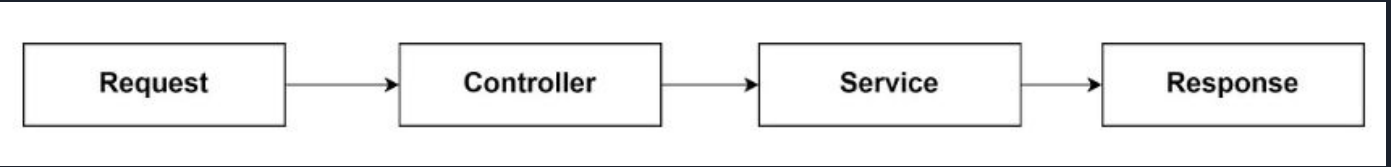
- **Interceptors:** Interceptors are used for intercepting incoming requests before they reach the controller and outgoing responses before they are sent back to the client. That allows for easy manipulation of data, logging, and error handling.

- **Middleware:** Middleware are functions that can be applied to all requests or specific routes, allowing for additional processing before or after the controller logic is executed. That can include tasks like authentication, data validation, and error handling.

Understanding Basic & Extended Flow

Understanding its Basic Flow(Request to Response)

- **Controller, Service(Provider), and Module:** These are the three basic building blocks of a Nest JS application. The controller handles incoming requests, the service provides business logic and data manipulation, and the module acts as a container for both.



This is the basic flow of data in a Nest JS application. The controller handles the request, which then calls the appropriate service to perform business logic and return a response.

Understanding its Extended Flow(Request to Response)

➔ This is the extended flow of data in a Nest JS application, showcasing all the components and their roles in processing requests and responses.

Request — -> Middleware — -> Guard — -> Interceptor — -> Pipe — ->
Controller — -> Service — -> Interceptor (Outbound) — -> Filter (Exception)

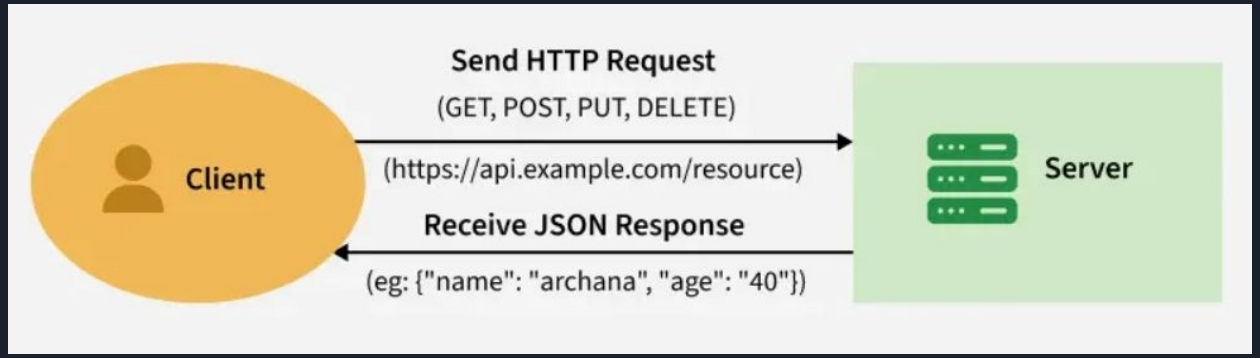
What is REST API?

What is REST API?

REST API stands for **RE**presentational **S**tate **T**ransfer **API**. It is a type of **API (Application Programming Interface)** that allows communication between different systems over the internet. **REST APIs** work by sending requests and receiving responses, typically in **JSON format**, between the client and server.

REST APIs use **HTTP methods** (such as GET, POST, PUT, DELETE) to define actions that can be performed on resources. These methods align with **CRUD (Create, Read, Update, Delete)** operations, which are used to manipulate resources over the web.

Example REST API



A request is sent from the client to the server via a web URL, using one of the HTTP methods. The server then responds with the requested resource, which could be HTML, XML, Image, or JSON, with JSON being the most commonly used format for modern web services.

Http Methods

Http Methods

The PUT, GET, POST, and DELETE methods are typically used in REST based architecture. The following table gives an explanation of these operations:

HTTP Method	CRUD Operation	Description
POST	INSERT	The POST verb is most often utilized to create new resources.
PUT	The **PUT** and **PATCH** methods are both used to update resources on a server in RESTful APIs, but they have key differences in how they work.	Used when you want to replace a resource completely. For example, replacing a user's entire profile data.
PATCH		Used when you want to update only a specific part of a resource. For example, updating just the email address in a user's profile.
GET	SELECT	Fetches a resource. The resource is never changed via Get resource.
DELETE	DELETE	Delete a resource

Http Response Status Codes

Http Response Status Codes

Http response status codes are grouped in five classes:

100 – 199	Information response code
200 – 299	Successful response code
300 – 399	Redirection response code
400 – 499	Client response code
500 – 599	Server response code

Http Response Status Codes

The status codes listed below are defined by RFC 9110

Success (2xx):

- **200 OK:** The request was successful.
- **201 Created:** The request was successful and a resource was created.
- **202 Accepted:** The request has been accepted for processing, but the processing hasn't been completed yet.
- **204 No Content:** There is no content to send for this request, but the headers are useful. The user agent may update its cached headers for this resource with the new ones.

Client Errors (4xx):

- **400 Bad Request:** The server could not understand the request due to a syntax error or missing information.
- **401 Unauthorized:** The request requires authentication, or the provided credentials are invalid.
- **403 Forbidden:** The client does not have permission to access the requested resource.

Http Response Status Codes

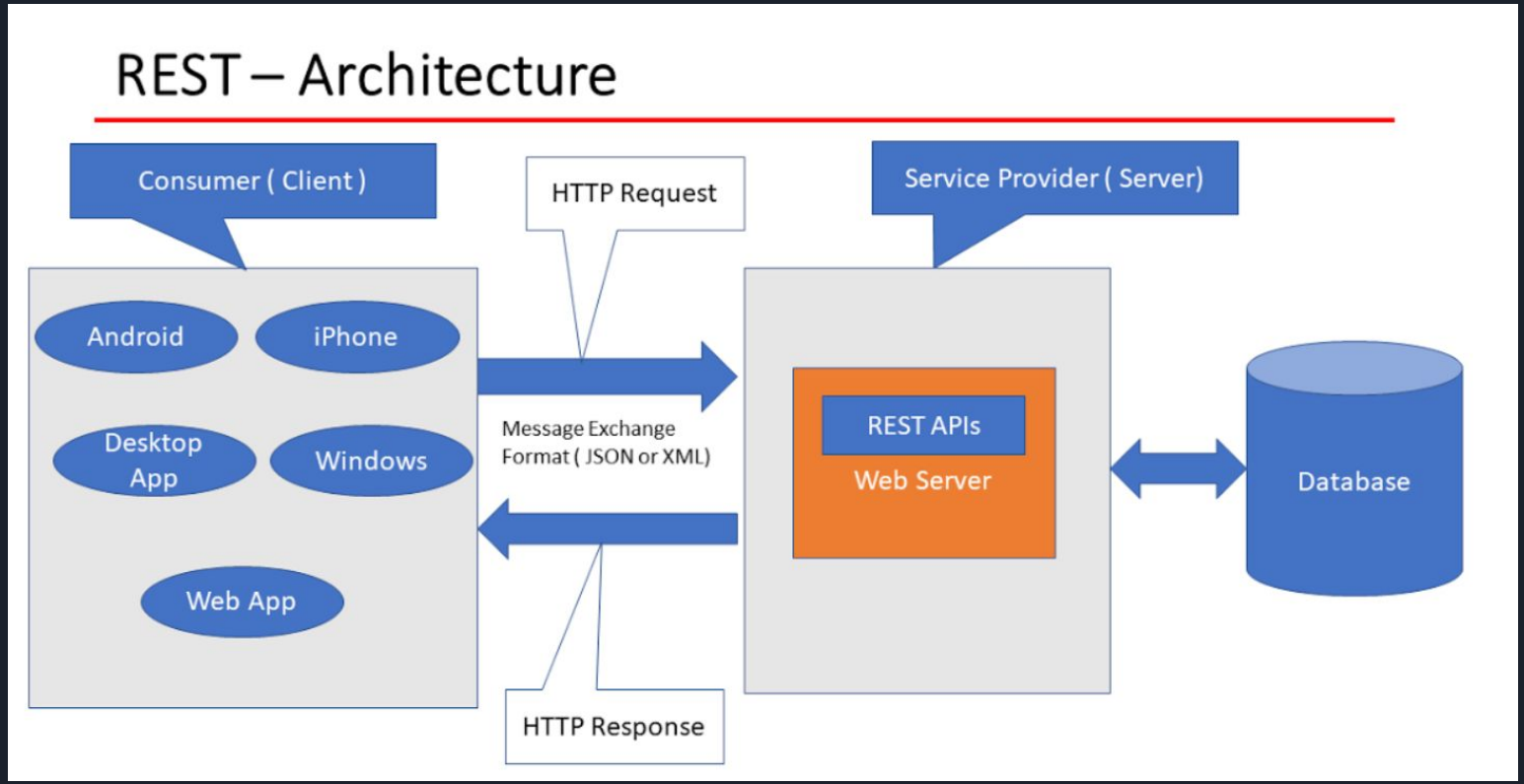
- **404 Not Found:** The requested resource could not be found.
- **405 Method Not Allowed:** The HTTP method used is not allowed for the requested resource.

Server Errors (5xx):

- **500 Internal Server Error:** A generic error indicating that the server encountered an unexpected condition.
- **501 Not Implemented:** The server does not support the functionality required to fulfill the request.
- **503 Service Unavailable:** The server is currently unavailable.

Architecture of REST API

Architecture of REST API



Implement a Simple Rest API using Nest JS

Implement a Simple Rest API using Nest JS

- Create a API Project by using below command

  Nest new user-api

- We need to implement below APIs:
  - Create a User (POST)
  - Retrieve a User (GET)
  - Update a User (PUT)
  - Partially Update a User (PATCH)
  - Delete a User (DELETE)

Implement a Simple Rest API using Nest JS

➔ **Step # 01:** Create **user.controller.ts**

This file will handle all five HTTP methods.

Command: nest g controller user

➔ **Step # 02:** Create a **user.service.ts**

This file will implement methods using the user interface

Command: nest g service user

➔ **Step # 03:** Define an Interface **user.interface.ts**

Create a file **user.interface.ts** inside the user folder to define the User interface.

Implement a Simple Rest API using Nest JS

- **Step # 04: user.interface.ts** should look like this

```
export interface User {
  id: number;
  name: string;
  email: string;
}
```

- **Step # 05:** Now open the **user.service.ts** implement the methods using the user interface
  - Define a hardcoded list in the service to perform the ALL operations

```
private users: User[] = [{ id: 1, name: 'Alice', email: 'alice@example.com' }];
```

Implement a Simple Rest API using Nest JS

➔ **Step # 05:** Continue

- Now define the GET method to retrieve the hardcoded list items

```
// GET - Retrieve user by ID
getUser(id: number): User | { message: string } {
  return this.users.find((user) => user.id === id) || { message: 'User not found' };
}
```

- Now define the POST method to save the data in the list

```
// POST - Create a new user
createUser(user: { name: string; email: string }): User {
  const newUser: User = { id: this.users.length + 1, ...user };
  this.users.push(newUser);
  return newUser;
}
```

Implement a Simple Rest API using Nest JS

➔ **Step # 05:** Continue

- Now define the PUT method to update the values available in the list items

```
// PUT - Replace a user
updateUser(id: number, userDto: { name: string; email: string }): User | { message: string
  const index = this.users.findIndex((user) => user.id === id);
  if (index !== -1) {
    this.users[index] = { id, ...userDto };
    return this.users[index];
  }
  return { message: 'User not found' };
}
```

Implement a Simple Rest API using Nest JS

➔ **Step # 05**: Continue

- Now define the PATCH method to update the values that was send in payload not all.

```
// PATCH - Partially update a user
partiallyUpdateUser(id: number, userDto: Partial<User>): User | { message: string } {
  const user = this.users.find((user) => user.id === id);
  if (!user) return { message: 'User not found' };

  Object.assign(user, userDto);
  return user;
}
```

Implement a Simple Rest API using Nest JS

➔ **Step # 05:** Continue

- Now define the **DELETE** method to delete the record from the list.

```
// DELETE a user by ID
deleteUser(id: number): string {
  const index = this.users.findIndex((user) => user.id === id);
  if (index === -1) {
    return `User with ID ${id} not found.`;
  }
  this.users.splice(index, 1);
  return `User with ID ${id} has been deleted.`;
}
```

Implement a Simple Rest API using Nest JS

- **Step # 06:** Now open the **user.controller.ts** file to implement the endpoint methods using the user service
  - First inject the constructor by injecting the User Service

```
@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}
```

  - Now implement the **GET** by Id endpoint by using the injected service

```
// GET - Retrieve a user by ID
@Get(':id')
getUser(@Param('id') id: number): User | { message: string } {
  return this.userService.getUser(Number(id));
}
```

Implement a Simple Rest API using Nest JS

- **Step # 06:** Continue...
  - Now implement the **POST** endpoint to save the some values in the pre-defined list

```
// POST - Create a new user
@Post()
createUser(@Body() newUser: { name: string; email: string }): User {
  return this.userService.createUser(newUser);
}
```

  - It's time implement the **PUT** endpoint to update the record in the list

```
// PUT - Replace a user
@Put(':id')
updateUser(@Param('id') id: number, @Body() updateUserDto: { name: string; email: string }
  return this.userService.updateUser(Number(id), updateUserDto);
}
```

Implement a Simple Rest API using Nest JS

➔ **Step # 06:** Continue...

- Now implement the **PATCH** endpoint to update the few or one value in the record of pre-defined list

```
// PATCH - Partially update a user
@Patch(':id')
partiallyUpdateUser(@Param('id') id: number, @Body() updateUserDto: Partial<User>): User |
  return this.userService.partiallyUpdateUser(Number(id), updateUserDto);
}
```

- Now implement the **DELETE** endpoint to delete the record from the list

```
@Delete(':id')
deleteUser(@Param('id') id: string) {
  return this.userService.deleteUser(Number(id));
}
```

Implement a Simple Rest API using Nest JS

➔ **Step # 07**: Now create new file **user.module.ts** by using a below command

- **Command**: nest g module user
- Modify **user.module.ts** to include the service.
- Your user.module.ts should look like this

```
import { Module } from '@nestjs/common';
import { UserController } from './user.controller';
import { UserService } from './user.service';

@Module({
  controllers: [UserController],
  providers: [UserService], // Register the service
})
export class UserModule {}
```

Implement a Simple Rest API using Nest JS

➔ **Step # 08:** Ensure **UserModule** is imported in the **app.module.ts**

- Your app.module.ts should look like this

```
import { Module } from '@nestjs/common';
import { UserModule } from './user/user.module';
import { UserController } from './user/user.controller';
import { UserService } from './user/user.service';
import { AppService } from './app.service';

@Module({
  imports: [UserModule],
  controllers: [UserController],
  providers: [AppService, UserService],
})
export class AppModule {}
```

Test API Endpoints

- Start the Server
  - Npm run start
- GET Request (Retrieve User)
  - **(GET)** http://localhost:3000/user/1
- POST Request (Create a New User)
  - **(POST)** http://localhost:3000/user
  - Body: { "name": "Bob", "email": "bob@example.com" }
- PUT Request (Replace User)
  - **(PUT)** http://localhost:3000/user/1
  - Body: { "name": "John", "email": "john@example.com" }
- PATCH Request (Partially Update User)
  - **(PATCH)** http://localhost:3000/user/1
  - Body: { "email": "john.doe@example.com" }
- DELETE Request
  - **(DELETE)** http://localhost:3000/user/1

Summary of API

- ➔ Defined an Interface (**user.interface.ts**)✅
- ➔ Implemented Service (**user.service.ts**)✅
- ➔ Used Service in Controller (**user.controller.ts**)✅
- ➔ Registered Service in Module (**user.module.ts**)✅
- ➔ Ensure app.module.ts have update imports, controllers & provider ✅
- ➔ Tested All HTTP Methods ✅