

# Day 7 Assignment

---

## Section 1: Theory

### 1. What is the purpose of a module in a backend framework like NestJS?

In NestJS, a module serves as a way to organize related components such as controllers, providers (services), and other modules. It helps in encapsulating functionality and managing dependencies effectively. Modules provide a modular structure which makes the application more maintainable and scalable. They allow for logical grouping of features and promote separation of concerns within the app.

### 2. Explain the difference between a Controller and a Service. Why is it beneficial to separate them?

A Controller in NestJS handles incoming HTTP requests and routes them to appropriate services. A Service contains the business logic and is responsible for performing operations like fetching data or computations. Separating them improves code readability, maintainability, and makes it easier to test. It also allows reusability of logic as services can be injected into multiple controllers or other services.

### 3. What is a DTO and how does it help in maintaining clean architecture?

A DTO (Data Transfer Object) is a plain TypeScript class used to define the shape of data sent over the network. It ensures type safety and data validation at compile time. DTOs help in maintaining clean architecture by clearly defining the data structure and keeping data handling consistent. They separate data structure concerns from business logic, enhancing modularity.

### 4. Describe the flow of an HTTP request from a client to the server in a NestJS application.

When a client sends a request, it first reaches a Controller based on the route defined. The Controller then calls the appropriate method in the Service for processing. The Service contains the logic to process the request, interact with databases or perform operations. The result is sent back to the Controller, which then sends the response back to the client.

### 5. What decorators are used in controllers to map HTTP methods to functions?

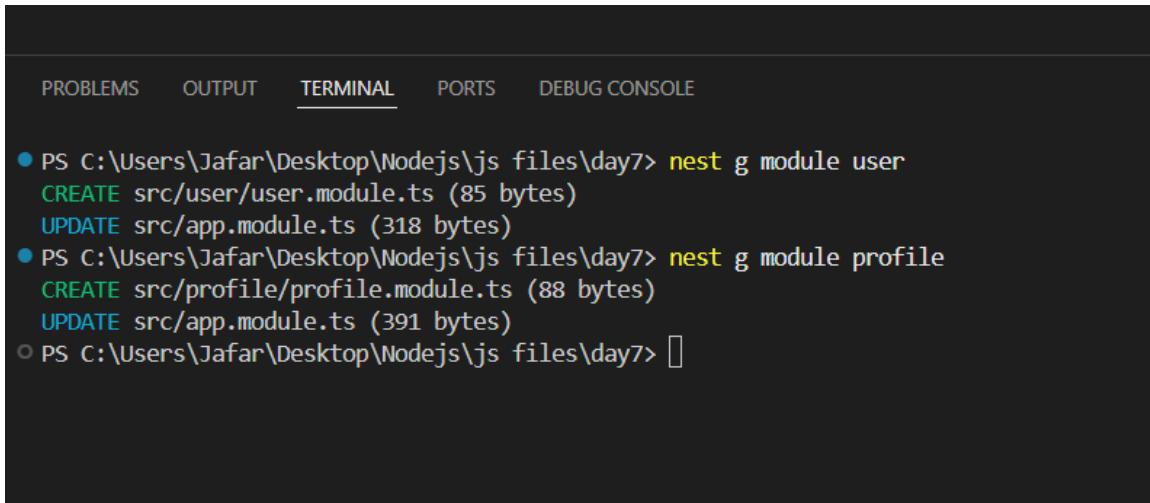
#### Explain briefly each one of them.

@Get() is used to handle HTTP GET requests. @Post() handles POST requests. @Put() is for updating resources using PUT, while @Delete() handles DELETE operations. Each decorator maps a specific HTTP method to a controller function, making it easy to define RESTful endpoints. These decorators enhance readability and maintainability of code by clearly indicating the request method.

## Section 2: Practical (Code-based)

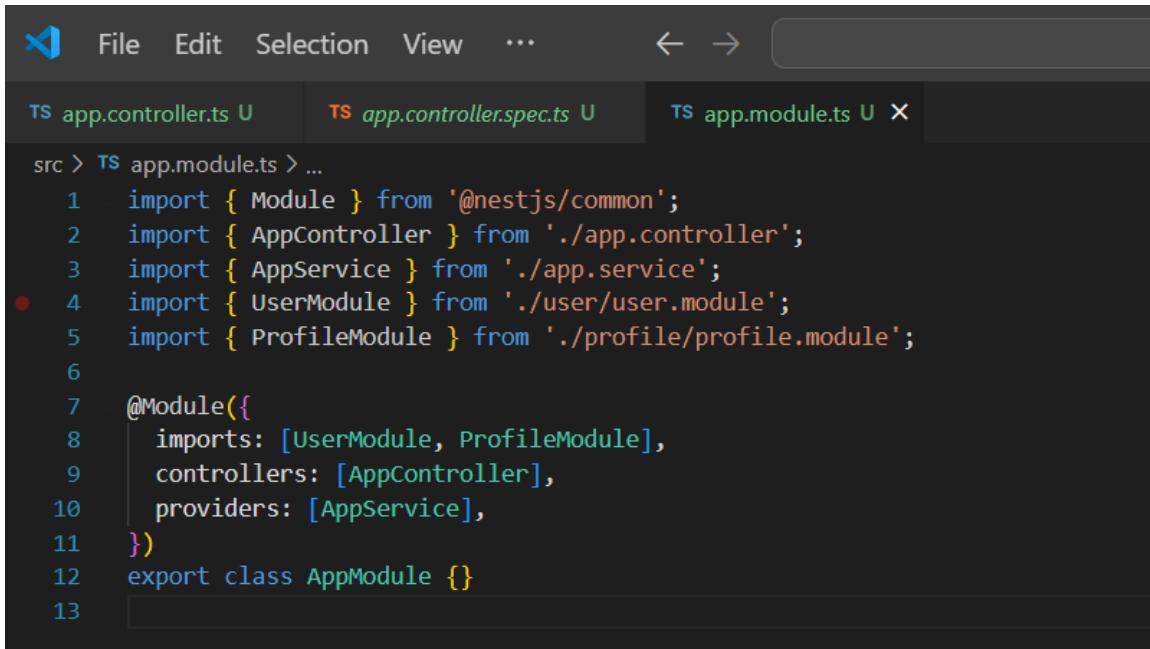
### 6. Create a **UserModule** and a **ProfileModule**, and show how they can be imported into the root **AppModule**.

The NestJS CLI commands `nest g module user` and `nest g module profile` to generate the modules. To use them in AppModule, import them in the imports array of app.module.ts like this:



```
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

● PS C:\Users\Jafar\Desktop\Nodejs\js files\day7> nest g module user
CREATE src/user/user.module.ts (85 bytes)
UPDATE src/app.module.ts (318 bytes)
● PS C:\Users\Jafar\Desktop\Nodejs\js files\day7> nest g module profile
CREATE src/profile/profile.module.ts (88 bytes)
UPDATE src/app.module.ts (391 bytes)
○ PS C:\Users\Jafar\Desktop\Nodejs\js files\day7> [ ]
```

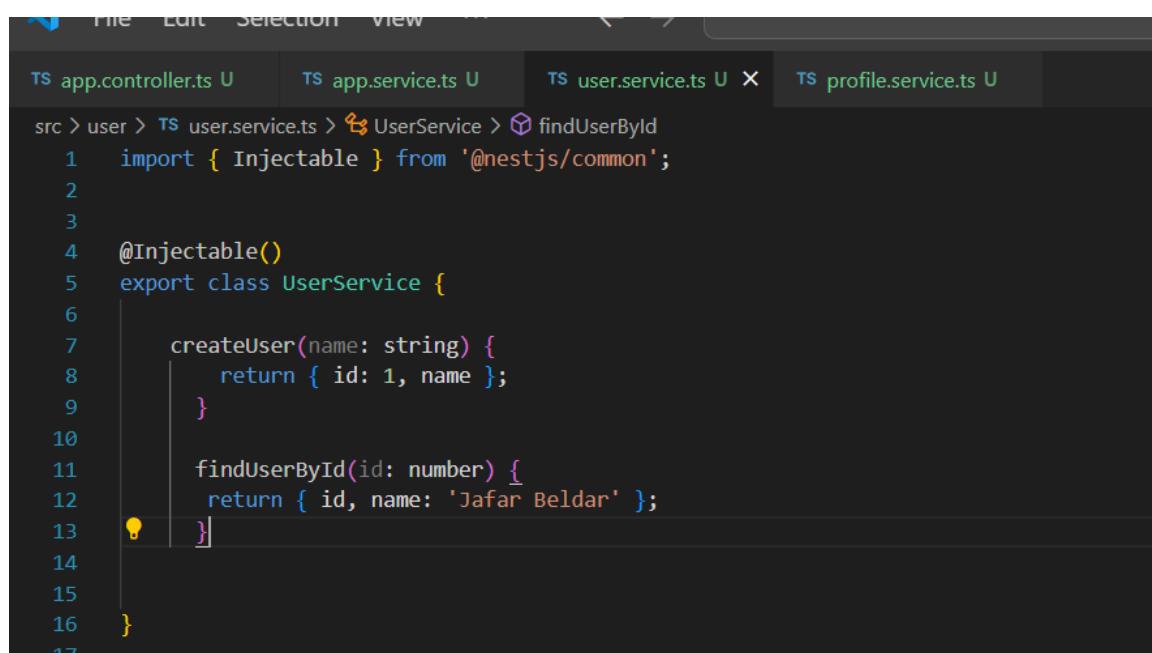


```
File Edit Selection View ...
src > TS app.module.ts > ...
1 import { Module } from '@nestjs/common';
2 import { AppController } from './app.controller';
3 import { AppService } from './app.service';
4 import { UserModule } from './user/user.module';
5 import { ProfileModule } from './profile/profile.module';
6
7 @Module({
8   imports: [UserModule, ProfileModule],
9   controllers: [AppController],
10  providers: [AppService],
11 })
12 export class AppModule {}
```

## 7. Write a UserService with createUser and findUserById, and a ProfileService with createProfile and getProfileForUser. Let ProfileService depend on UserService (i.e., one service uses another).

we can inject UserService into ProfileService using constructor injection. This allows ProfileService to access and use methods from UserService. It's an example of dependency injection which is a core concept in NestJS for managing service interactions. Each service handles its own responsibilities, and collaboration is done through injection.

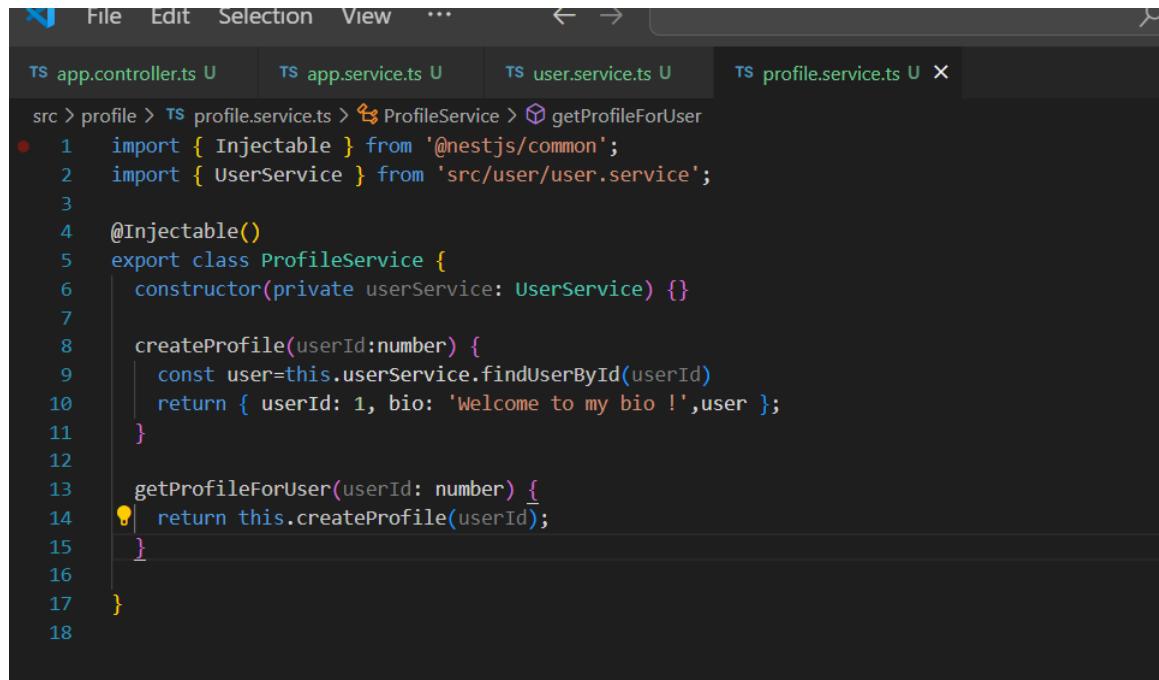
```
PS C:\Users\Jafar\Desktop\Nodejs\js files\day7> nest g service user
● >>
CREATE src/user/user.service.ts (92 bytes)
CREATE src/user/user.service.spec.ts (464 bytes)
UPDATE src/user/user.module.ts (159 bytes)
● PS C:\Users\Jafar\Desktop\Nodejs\js files\day7> nest g service profile
CREATE src/profile/profile.service.ts (95 bytes)
CREATE src/profile/profile.service.spec.ts (485 bytes)
UPDATE src/profile/profile.module.ts (171 bytes)
```



The screenshot shows a code editor with four tabs at the top: app.controller.ts, app.service.ts, user.service.ts (which is currently selected), and profile.service.ts. The user.service.ts file contains the following code:

```

src > user > user.service.ts > UserService > findUserById
1 import { Injectable } from '@nestjs/common';
2
3
4 @Injectable()
5 export class UserService {
6
7   createUser(name: string) {
8     return { id: 1, name };
9   }
10
11   findUserById(id: number) {
12     return { id, name: 'Jafar Beldar' };
13   }
14
15
16 }
```



The screenshot shows a code editor with a dark theme. The tab bar at the top has five tabs: 'app.controller.ts U', 'app.service.ts U', 'user.service.ts U', 'profile.service.ts U' (which is the active tab), and 'profile.service.ts U X'. The code in the editor is as follows:

```
src > profile > profile.service.ts > ProfileService > getProfileForUser
1 import { Injectable } from '@nestjs/common';
2 import { UserService } from 'src/user/user.service';
3
4 @Injectable()
5 export class ProfileService {
6   constructor(private userService: UserService) {}
7
8   createProfile(userId:number) {
9     const user=this.userService.findUserById(userId)
10    return { userId: 1, bio: 'Welcome to my bio !',user };
11  }
12
13   getProfileForUser(userId: number) {
14     return this.createProfile(userId);
15   }
16
17 }
18
```

## 8. Create two controllers: UserController and ProfileController. Each should expose basic CRUD endpoints.

Use the CLI to create controllers with `nest g controller user` and `nest g controller profile`. Each controller defines routes for creating, reading, updating, and deleting users, profiles. The controllers use decorators like `@Get()`, `@Post()`, etc., to map routes to functions. These routes interact with services to execute the underlying logic.

USER CONTROLLER:

```
● ● ●

1 import {
2   Body,
3   Controller,
4   Delete,
5   Get,
6   Param,
7   Post,
8   Put,
9 } from '@nestjs/common';
10 import { UserService } from './user.service';
11
12 @Controller('user')
13 export class UserController {
14   constructor(private userService: UserService) {}
15
16   @Get('get-user/:userId')
17   getUserId(@Param('userId') userId: number) {
18     return this.userService.findUserById(userId);
19   }
20
21   @Post('create-user/')
22   createUSeR(@Body('name') name: string) {
23     return this.userService.createUser(name);
24   }
25
26   @Put('update-user/:userId')
27   updateUser(@Param('id') id: number, @Body('name') name: string) {
28     return { message: `Updated user ${id} with name ${name}` };
29   }
30
31   @Delete('delete-user/:userId')
32   deleteUser(@Param('userId') userId: number) {
33     return { message: `Deleted user with id: ${userId}` };
34   }
35 }
36
```

## Jafar Beldar

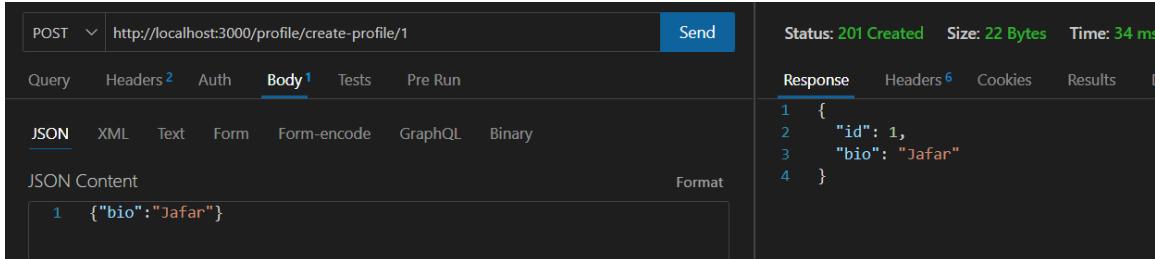
The image displays four separate screenshots of the Postman application interface, each showing a different API request and its response.

- Screenshot 1 (Top Left):** A GET request to `http://localhost:3000/user/get-user/1`. The response status is 200 OK, size is 32 Bytes, and time is 7 ms. The response body is a JSON object with `id: "1"` and `name: "Jafar Beldar"`.
- Screenshot 2 (Top Right):** A POST request to `http://localhost:3000/user/create-user`. The response status is 201 Created, size is 23 Bytes, and time is 4 ms. The response body is a JSON object with `id: 1` and `name: "Jafar"`.
- Screenshot 3 (Bottom Left):** A PUT request to `http://localhost:3000/user/update-user/1`. The response status is 200 OK, size is 52 Bytes, and time is 21 ms. The response body is a JSON object with a message: `"Updated user undefined with name Jafar"`.
- Screenshot 4 (Bottom Right):** A DELETE request to `http://localhost:3000/user/delete-user/1`. The response status is 200 OK, size is 37 Bytes, and time is 29 ms. The response body is a JSON object with a message: `"Deleted user with id: 1"`.

## **PROFILE :**

```
1 import {
2   Body,
3   Controller,
4   Delete,
5   Get,
6   Param,
7   Post,
8   Put,
9 } from '@nestjs/common';
10 import { ProfileService } from './profile.service';
11
12 @Controller('profile')
13 export class ProfileController {
14   constructor(private profileService: ProfileService) {}
15
16   @Post('create-profile/:userId')
17   create(@Param('userId') userId: number, @Body('bio') bio: string) {
18     return this.profileService.createProfile(bio);
19   }
20
21   @Get('get-profile/:userId')
22   get(@Param('userId') userId: number) {
23     return this.profileService.getProfileForUser(userId);
24   }
25
26   @Put('update-profile/:userId')
27   update(@Param('userId') userId: number, @Body('bio') bio: string) {
28     return { message: `Updated profile for user ${userId} with bio: ${bio}` };
29   }
30
31   @Delete('delete-profile/:userId')
32   delete(@Param('userId') userId: number) {
33     return { message: `Deleted profile for user ${userId}` };
34   }
35 }
```

## Jafar Beldar



POST  Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

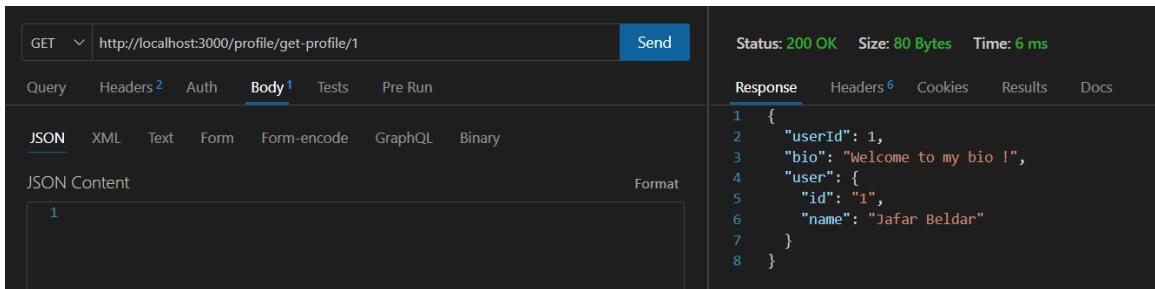
JSON Content Format

```
1 {"bio": "Jafar"}
```

Status: 201 Created Size: 22 Bytes Time: 34 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "id": 1,
3   "bio": "Jafar"
4 }
```



GET  Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

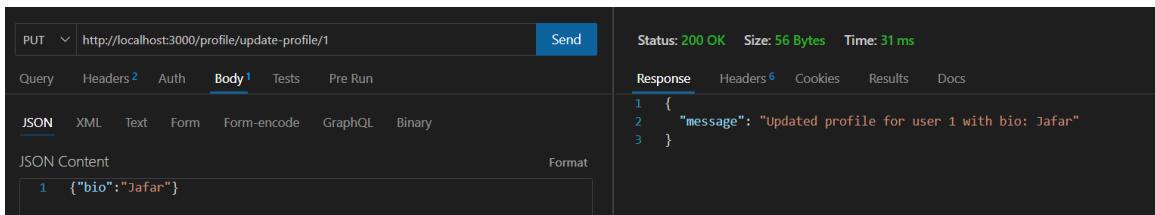
JSON Content Format

```
1
```

Status: 200 OK Size: 80 Bytes Time: 6 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "userId": 1,
3   "bio": "Welcome to my bio !",
4   "user": {
5     "id": "1",
6     "name": "Jafar Beldar"
7   }
8 }
```



PUT  Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

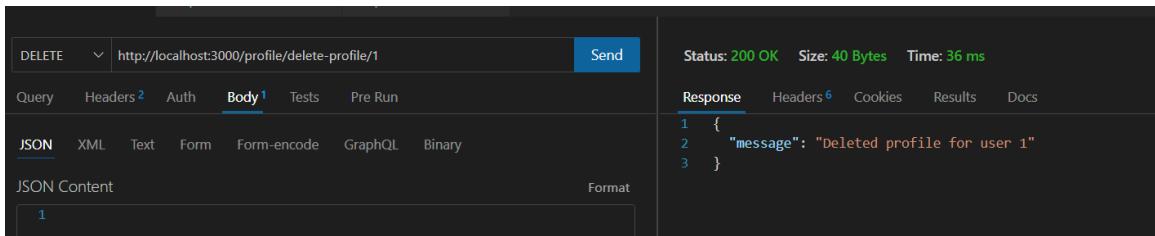
JSON Content Format

```
1 {"bio": "Jafar"}
```

Status: 200 OK Size: 56 Bytes Time: 31 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "message": "Updated profile for user 1 with bio: Jafar"
3 }
```



DELETE  Send

Query Headers 2 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1
```

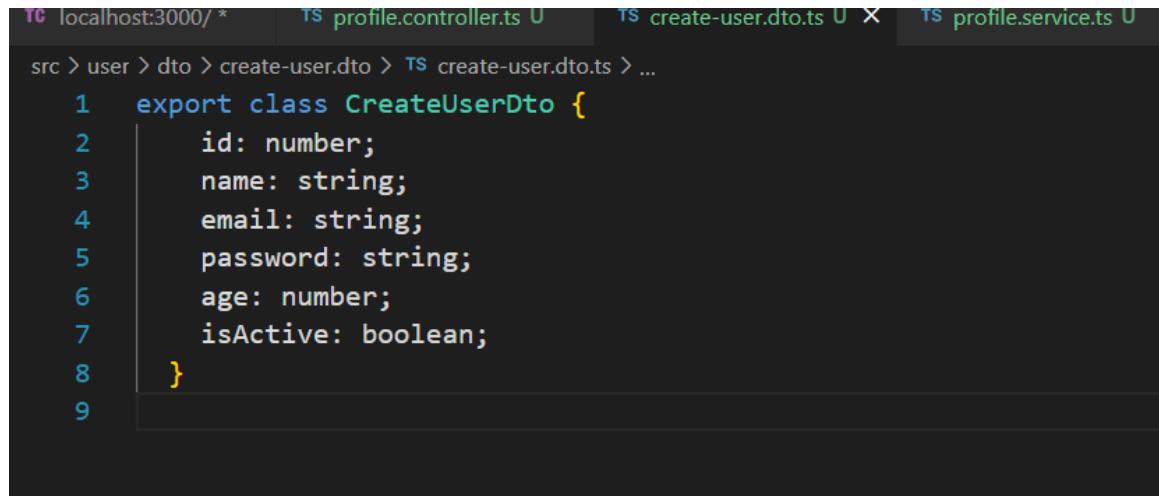
Status: 200 OK Size: 40 Bytes Time: 36 ms

Response Headers 6 Cookies Results Docs

```
1 {
2   "message": "Deleted profile for user 1"
3 }
```

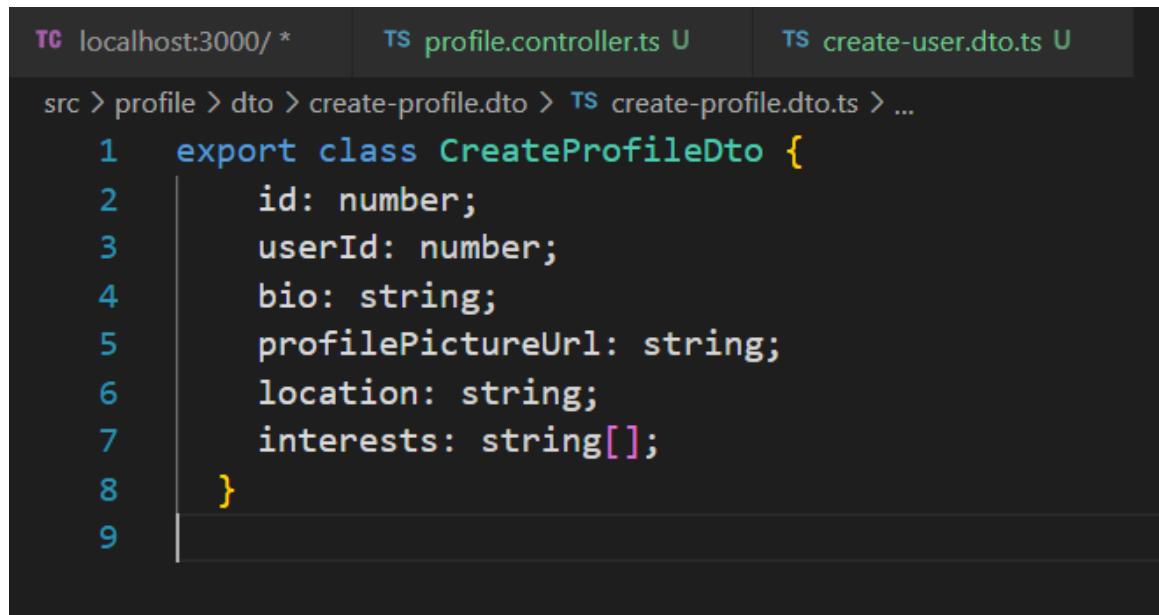
## 9. Define a `CreateUserDto` and a `CreateProfileDto`. Both should be plain TypeScript classes with appropriate fields (no decorators required).

DTOs can be created as simple classes like `class CreateUserDto { name: string; email: string; }`. Similarly, CreateProfileDto might have fields like bio, userId, etc. Using DTOs ensures the data format is well-defined and enforces type safety. They act as a contract for the data that a controller should expect.



The screenshot shows a code editor with four tabs at the top: 'localhost:3000/\*' (TC), 'profile.controller.ts U', 'create-user.dto.ts X', and 'profile.service.ts U'. The 'create-user.dto.ts' tab is active. Below the tabs, the file path 'src > user > dto > create-user.dto > create-user.dto.ts > ...' is shown. The code for the `CreateUserDto` class is displayed:

```
1  export class CreateUserDto {
2      id: number;
3      name: string;
4      email: string;
5      password: string;
6      age: number;
7      isActive: boolean;
8  }
9
```



The screenshot shows a code editor with three tabs at the top: 'localhost:3000/\*' (TC), 'profile.controller.ts U', and 'create-user.dto.ts U'. The 'create-user.dto.ts' tab is active. Below the tabs, the file path 'src > profile > dto > create-profile.dto > create-profile.dto.ts > ...' is shown. The code for the `CreateProfileDto` class is displayed:

```
1  export class CreateProfileDto {
2      id: number;
3      userId: number;
4      bio: string;
5      profilePictureUrl: string;
6      location: string;
7      interests: string[];
8  }
9
```

## 10. Demonstrate with code how to inject one service (e.g., UserService) into another module's service (e.g., ProfileService) using NestJS dependency injection.

First, UserService is exported in UserModule using the `exports` array. Then, import UserModule in ProfileModule's imports. In ProfileService, we inject UserService via the constructor. This allows services across modules to communicate and use each other's functionalities.

Step 1: Export UserService in UserModule

```
1 // user.module.ts
2 import { Module } from '@nestjs/common';
3 import { UserService } from './user.service';
4 import { UserController } from './user.controller';
5
6 @Module({
7   providers: [UserService],
8   controllers: [UserController],
9   exports: [UserService], // for injection into other modules
10 })
11 export class UserModule {}
```

## Step 2: Import UserModule in ProfileModule

```
● ● ●
1 // profile.module.ts
2 import { Module } from '@nestjs/common';
3 import { ProfileService } from './profile.service';
4 import { ProfileController } from './profile.controller';
5 import { UserModule } from '../user/user.module'; // importing UserModule
6
7 @Module({
8   imports: [UserModule], // important for using exported services
9   providers: [ProfileService],
10  controllers: [ProfileController],
11 })
12 export class ProfileModule {}
```

## Step 3: Inject UserService in ProfileService

```
● ● ●
1 // profile.service.ts
2 import { Injectable } from '@nestjs/common';
3 import { UserService } from '../user/user.service';
4
5 @Injectable()
6 export class ProfileService {
7   constructor(private readonly userService: UserService) {}
8
9   getUserData(id: number) {
10     return this.userService.findUserById(id); // using method from UserService
11   }
12 }
13
14
```