NestJS Architecture

Course Code: CSC 4182   Course Title: Advanced Programming In Web Technologies

Dept. of Computer Science
Faculty of Science and Technology

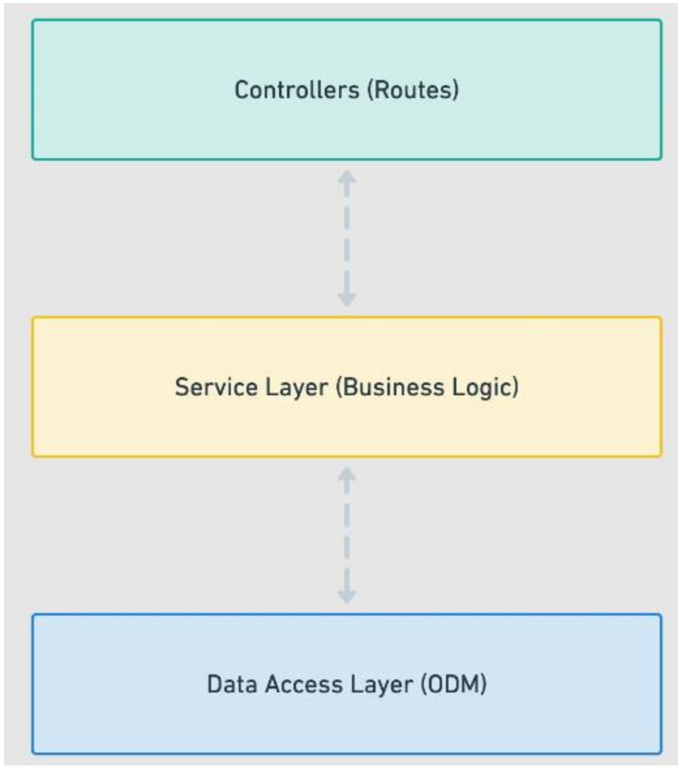
Lecture No:	**1**	**Week No:**	**02**	**Semester:**	
Lecturer:	*Sazzad Hossain; sazzad@aiub.edu*				

Lecture Outline

- ✓ NestJS
- ✓ NestJS Project Directory structure
- ✓ NestJS Controllers
- ✓ Dependency Injection
- ✓ NestJS Services/Provider
- ✓ NestJS Module
- ✓ NestJS Decorators
- ✓ NestJS DTO

NestJS

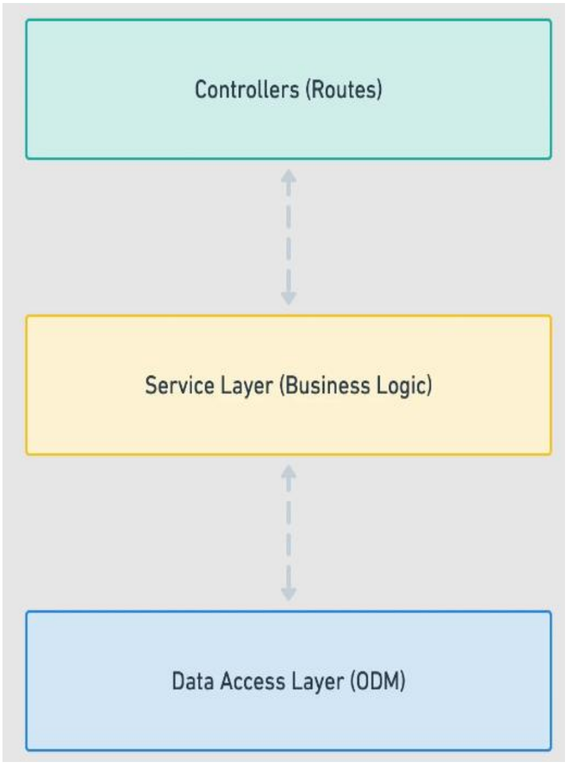
Basic NestJS Architecture has **3-Tier Architecture**



NestJS

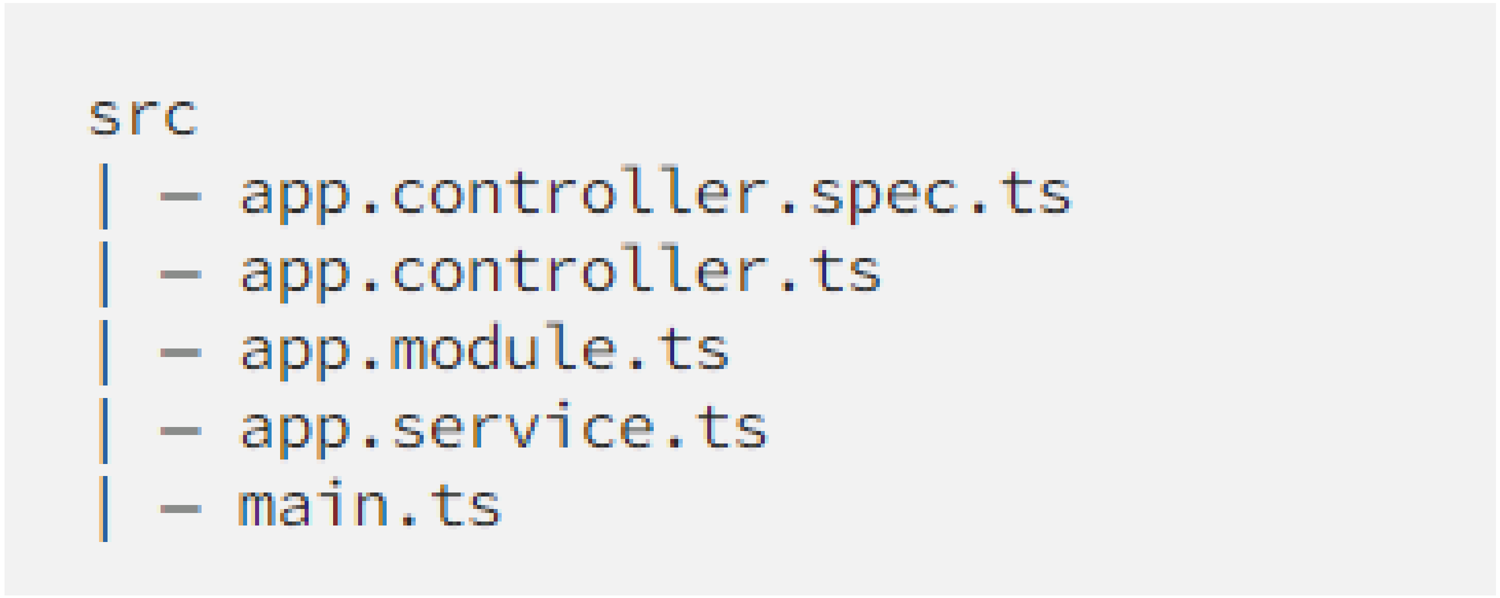
Basic NestJS Architecture has **3-Tier Architecture**

1. **Controllers:** A controller's sole purpose is to receive requests for the application and deal with routes.
2. **Service Layer:** This part of the block should only include business logic. For example, all the CRUD operations and methods to determine how data can be created, stored and updated.
3. **Data Access Layer:** This layer takes care and provides logic to access data stored in persistent storage of some kind. For example an ODM like PostgreSQL, Mongoose.



NestJS Project Directory structure.

These are the core files



```
src
| — app.controller.spec.ts
| — app.controller.ts
| — app.module.ts
| — app.service.ts
| — main.ts
```

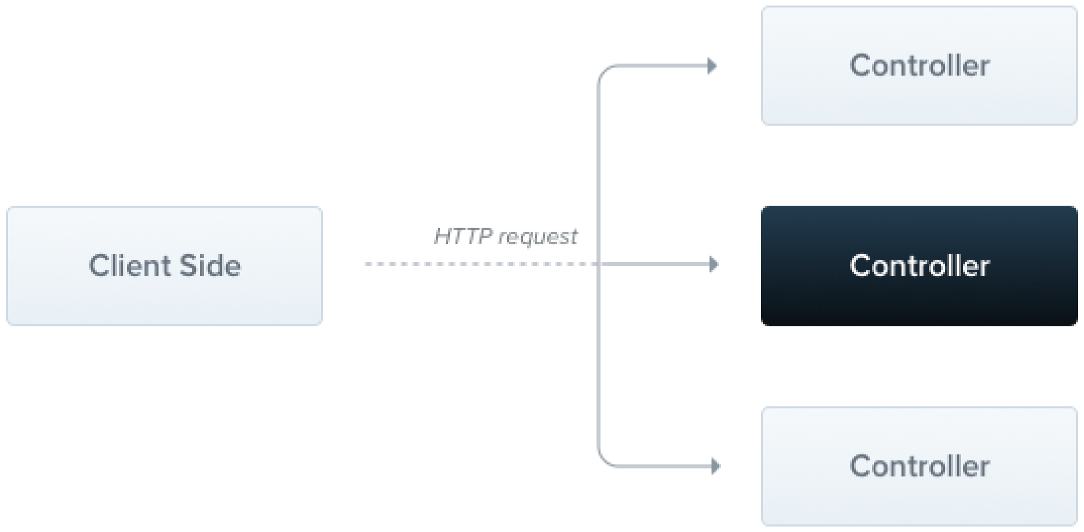
NestJS Project Directory structure.

app.controller.ts: Controller file that will contain all the application routes.
app.controller.spec.ts: This file would help writing out unit tests for the controllers.
app.module.ts: The module file essentially bundles all the controllers and providers of your application together.
app.service.ts: The service will include methods that will perform a certain operation. For example: Registering a new user.
main.ts: The entry file of the application will take in your module bundle and create an app instance using the NestFactory provided by Nest.

NestJS Controllers

- Controllers **handle the routing and request** handling logic for specific **API endpoints**.
- They define **the routes, HTTP methods**, and corresponding actions to be performed when a request hits a specific endpoint.
- Controllers in Nest.js are regular classes annotated with the **@Controller()** decorator.
- Each method is annotated with HTTP method decorators such as @Get(), @Post(), @Put(), @Delete().
- Controllers **return** responses by returning data or using response objects.

NestJS Controllers



NestJS Controllers

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { AppService } from './app.service';
// Define the controller and set the base URL path to '/users'
@Controller('users')
export class UserController {
 //Injecting dependencies through the constructor
  constructor(private readonly userService: UserService) {}
  // Define a route handler for the HTTP GET method to handle requests to
'/users' with GET method
  @Get()
  getUsers() {
    return this.userService.getAllUsers();
  }
  // The `@Body()` decorator extracts the request body and binds it to the
`data` parameter
  @Post()
  createUser(@Body() data: string) {
    return this.userService.createUser(data);
  }
}
```

Dependency Injection

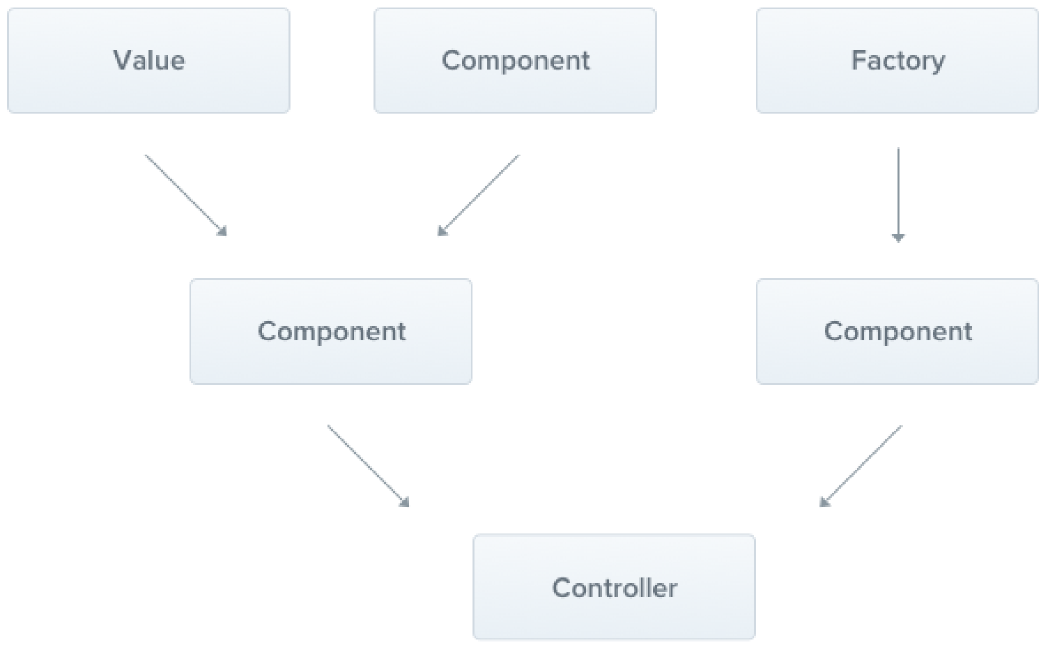
Dependency Injection (DI) is a design pattern and a **technique** used in software development to achieve **loose coupling between components and manage dependencies** between them.

Dependencies: Dependencies are other **classes**, **modules**, or **services** that a particular class or component relies on to perform its functionality. For example, a **UserService** may depend on a **UserRepository** to interact with the database.

Injection: Instead of directly creating an instance of the dependency within the dependent class, the dependency is "**injected**" from the **outside**. This means that **the dependent class doesn't need to know the details** of how the dependency is created or managed

NestJS Services/Provider

- Providers are a fundamental concept in Nest. Many of the basic Nest classes may be treated as a provider – **services, repositories, factories, helpers**, and so on.



NestJS Services/Provider

- Services encapsulate the **business logic** and contain the **implementation details of various functionalities** in your application.
- Services are regular classes annotated with the **@Injectable()** decorator, which marks them as injectable dependencies.
- The **providers** array within a **module** is used to register and provide instances of services.

NestJS Services/Provider

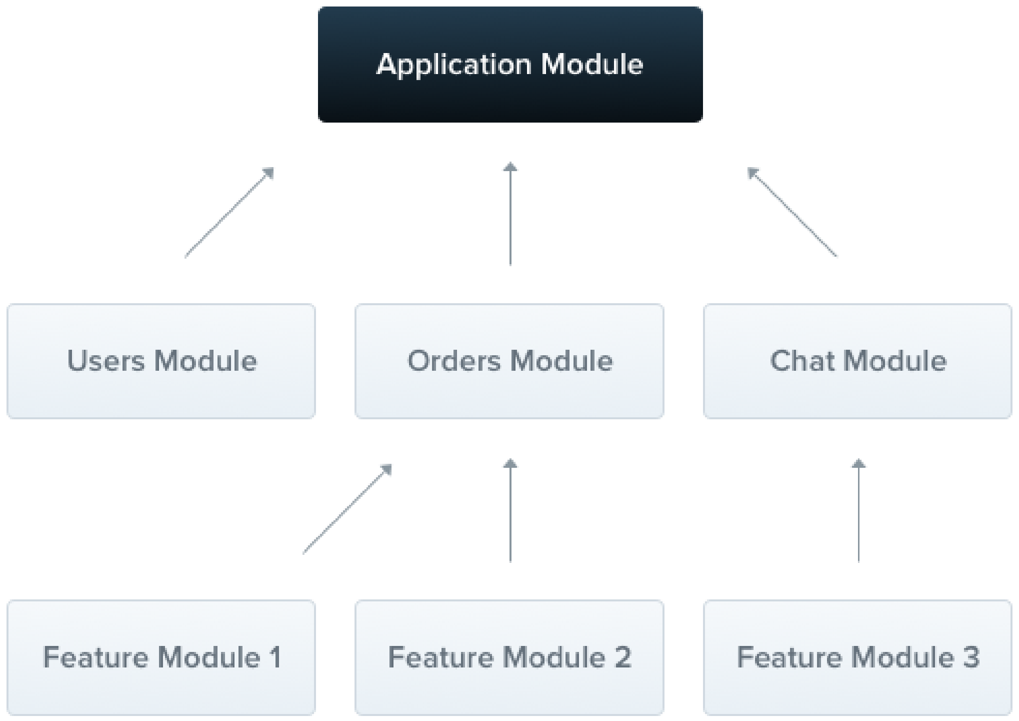
```
import { Injectable } from '@nestjs/common';

// Mark the class as injectable using the `@Injectable()` decorator
@Injectable()
export class UserService {
  // Method to fetch users
  getUsers(): string[] {
    return ['John', 'Jane', 'Doe'];
  }
  // Method to create a user
  // Accepts a `data` parameter of type string representing user
data
  createUser(data: string): string {
    // Logic to create a user using the provided data
    return `User created: ${data}`;
  }
}
```

NestJS Module

- Module is a **class** annotated with a **@Module()** decorator. The @Module() decorator provides **metadata** that Nest makes use of to **organize the application structure.**
- Each application has at least **one** module, a **root** module.
- Root module is the **starting** point Nest uses to build the application graph
- **internal data structure** Nest uses to resolve **module and provider** relationships and dependencies.

NestJS Module



NestJS Module

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

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

NestJS Decorators

1. Module Decorators:
   - **@Module()**: to define **module** metadata.
2. Controller Decorators:
   - **@Controller(path?: string)**: for the **controller**.
3. Provider Decorators:
   - **@Injectable()**: for the injectable provider.
4. Route Handler Decorators:
   - **@Get(path?: string | string[])**: to define the route path(s) for GET requests.
   - **@Post(path?: string | string[])**: to define the route path(s) for POST requests.
   - **@Put(path?: string | string[])**: to define the route path(s) for PUT requests.
   - **@Delete(path?: string | string[])**: to define the route path(s) for DELETE requests.
   - **@Patch(path?: string | string[])**: to define the route path(s) for PATCH requests.

NestJS Decorators

5.Parameter Decorators:

- **@Param(param: string)**: to extract a route parameter within a route handler.
- **@Query(param: string)**: to extract a query parameter within a route handler.
- **@Body(param?: string)**: to extract the request body within a route handler.
- **@Request:** injects the entire request object within a route handler.
- **@Response()**: injects the entire response object within a route handler.

6.Middleware Decorators:

- **@UseGuards()**: Attaches guards to a route handler or controller.
- **@UseInterceptors()**: Attaches interceptors to a route handler or controller.
- **@UsePipes()**: Attaches pipes to a route handler or controller.

7.Exception Decorators:

- **@Catch()**: Defines an exception filter for handling specific exceptions.
- **@HttpException()**: Defines a custom HTTP exception.

NestJS DTO

DTO stands for Data Transfer Object. It is a design pattern commonly used in software development to transfer data between different layers or components of an application. DTOs provide a structured way to encapsulate and transport data without exposing the underlying implementation details.

```
// createUser.dto.ts
export class CreateUserDto {
  name: string;
  email: string;
  password: string;
}
```

NestJS DTO

```
// user.controller.ts

import { Controller, Post, Body } from '@nestjs/common';
import { CreateUserDto } from './createUser.dto';
import { UserService } from './user.service';

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

  @Post()
  createUser(@Body() createUserDto: CreateUserDto) {
    return this.userService.createUser(createUserDto);
  }
}
```

NestJS DTO

```
// user.service.ts

import { Injectable } from '@nestjs/common';
import { CreateUserDto } from './createUser.dto';

@Injectable()
export class UserService {
  createUser(createUserDto: CreateUserDto): string {
    // Logic to create a new user using the data from the DTO
    const { name, email } = createUserDto;
    // ...create user logic...

    return ‘User created: ${name} (${email})’;
  }
}
```

References

1. W3Schools Online Web Tutorials, URL: http://www.w3schools.com
2. Node.js, URL: https://nodejs.org/en/
3. Next.js, URL: https://nextjs.org/
4. TypeScript URL: https://www.typescriptlang.org/
5. MDN Web Docs URL: https://developer.mozilla.org/

Thank You!