NestJS

Contents

1. Introduction to NestJS (2-3 pages)

What is NestJS?:

NestJS is a **progressive Node.js framework** used for building efficient, reliable, and scalable server-side applications. It is built with TypeScript and incorporates concepts from object-oriented programming, functional programming, and functional reactive programming, making it highly versatile and maintainable.

Key Features:

1. **Modular Architecture**: Allows you to split your application into reusable modules, promoting a highly organized codebase.
2. **TypeScript Support**: NestJS is written in TypeScript, providing strong typing and better tooling.
3. **Built-in Dependency Injection**: Helps manage dependencies efficiently and makes the application more testable and maintainable.
4. **Decorator-based Syntax**: Uses decorators like @Controller(), @Get(), @Injectable(), etc., making the code more declarative and easy to read.
5. **Platform Agnostic**: While it uses Express by default, you can easily switch to other frameworks like Fastify.

Use Cases:

- Building RESTful APIs
- GraphQL APIs
- Microservices
- WebSockets-based real-time applications

NestJS is popular for its structured and opinionated approach to building server-side applications, making it ideal for large-scale and enterprise-level projects.

Why Choose NestJS?:

Choosing NestJS comes with several advantages, especially if you're aiming for a scalable, maintainable, and organized backend solution. Here's why you might consider using NestJS:

1. Modular Architecture

- **Organized Codebase**: NestJS promotes a modular architecture, which means you can divide your application into smaller, reusable, and maintainable modules.
- **Scalability**: The modular approach makes it easier to add or remove features, making it ideal for large-scale applications.

2. TypeScript Out-of-the-Box

- **Type Safety**: As NestJS is built with TypeScript, it offers strong typing, which reduces runtime errors and makes the code more predictable.
- **Better Tooling**: TypeScript's autocomplete, IntelliSense, and refactoring capabilities provide an enhanced development experience.

3. Built-in Dependency Injection

- **Clean and Maintainable Code**: NestJS has a powerful dependency injection system that makes it easier to manage services and dependencies. This encourages loose coupling and increases testability.
- **Easier Unit Testing**: The use of dependency injection makes it simple to mock dependencies and write unit tests.

4. Decorator-Based and Declarative Programming

- **Cleaner Code**: The use of decorators like @Controller(), @Injectable(), @Get(), etc., makes the code more readable and declarative.
- **Angular Familiarity**: If you have experience with Angular, you'll find NestJS easy to understand since it shares many design principles and patterns.

5. Flexibility with Underlying Platforms

- **Express and Fastify**: NestJS uses Express as the default HTTP server framework but can be easily switched to Fastify for improved performance.
- **Broad Compatibility**: It integrates well with a variety of libraries and tools, such as WebSocket, GraphQL, microservices, and ORM tools like TypeORM, Sequelize, and Mongoose.

6. Microservices and GraphQL Support

- **Versatility**: NestJS has built-in support for microservices and GraphQL, making it an excellent choice for complex, distributed, or real-time applications.
- **Out-of-the-Box Features**: You can build REST APIs, GraphQL APIs, and microservices without needing additional frameworks.

7. Strong Community and Ecosystem

- **Growing Popularity**: NestJS has a rapidly growing community and ecosystem, meaning there are plenty of libraries, plugins, and support available.
- **Official and Third-Party Modules**: Many features and integrations (e.g., authentication, caching, logging) are available through NestJS's core or third-party modules.

8. Enterprise-Grade Solutions

- **Consistency and Best Practices**: NestJS provides a structured, opinionated way of building applications, ensuring adherence to best practices.
- **Scalable Architecture**: Suitable for building complex and enterprise-level applications due to its maintainable and scalable structure.

9. Easy Learning Curve for Angular Developers

- **Familiarity**: Developers who are familiar with Angular will find the learning curve minimal because of the shared concepts like decorators, dependency injection, and module-based architecture.

10. Comprehensive Documentation

- **Well-Documented**: NestJS has excellent and comprehensive documentation, making it easy to get started and find guidance as you build complex features.
- **Comparison with Other Frameworks**: NestJS vs. Express, Fastify, Koa, etc.

- **Installation and Setup**: Installing Node.js, Nest CLI, creating a new NestJS project

2. NestJS Architecture (4-5 pages)

NestJS follows a modular and structured architecture inspired by Angular. It combines concepts from object-oriented programming (OOP), functional programming (FP), and functional reactive programming (FRP) to create a framework that is scalable, maintainable, and easy to work with. Here's a breakdown of the key components of NestJS architecture:

1. Modules

- **What Are They?**: Modules are the fundamental building blocks of a NestJS application. A module is a class decorated with the @Module() decorator, which organizes related components (controllers, services, providers) together.
- **Purpose**: Modules help in organizing your application into cohesive blocks, making the codebase more maintainable and scalable.

```
@Module({
 imports: [],
 controllers: [UsersController],
 providers: [UsersService],
})
export class UsersModule {}
```

- **Root Module**: Every NestJS application has a root module, usually called AppModule, which is the entry point of the application.
- **Feature Modules**: As the application grows, you create feature modules to encapsulate related functionality.

2. Controllers

- **What Are They?**: Controllers are responsible for handling incoming HTTP requests and returning responses to the client. They define the application's routes and determine how requests should be handled.
- **Decorators**: Decorators like @Controller(), @Get(), @Post(), etc., are used to define routes and HTTP methods.

```
@Controller('users')
export class UsersController {
 @Get()
 findAll(): string {
  return 'This action returns all users';
 }

 @Post()
 create(@Body() createUserDto: CreateUserDto): string {
  return 'User created';
 }
}
```

- Controllers should not contain business logic; instead, they delegate that responsibility to services.

3. Providers (Services)

- **What Are They?**: Providers are classes that encapsulate the business logic and data access layer. They can be injected into controllers or other providers using Dependency Injection (DI).
- **Dependency Injection**: With DI, you can manage dependencies and make your code more testable and maintainable.

```
@Injectable()
export class UsersService {
 private readonly users = [];

 create(user: any) {
  this.users.push(user);
 }

 findAll(): any[] {
  return this.users;
 }
}
```

- **Scope**: Providers can be shared across the entire application or be limited to a specific module.

4. Dependency Injection (DI)

- **How It Works**: NestJS has a powerful DI system that allows you to inject dependencies into classes (e.g., controllers or providers). This is achieved using the @Injectable() decorator, which marks a class as a provider that can be injected.
- **Benefits**: DI makes the architecture more modular, testable, and maintainable by promoting loose coupling between components.

5. Decorators

- **What Are They?**: Decorators are special functions that allow you to add metadata to classes, methods, or properties. NestJS heavily uses decorators to provide a declarative way to define routes, modules, providers, and more.
- **Examples**:
  - @Controller(): Defines a controller.
  - @Get(), @Post(), @Put(), @Delete(): Define HTTP methods for routes.
  - @Injectable(): Marks a class as a provider.
  - @Module(): Defines a module.

6. Middleware

- **What Is It?**: Middleware functions are executed before the route handler and have access to the request and response objects. They can be used for tasks like logging, authentication, and request validation.

```
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log('Request received:', req.method, req.url);
    next();
  }
}
```

- **Applying Middleware**: You can apply middleware to routes using the configure() method in a module.

7. Exception Filters

- **What Are They?**: Exception filters handle errors in a centralized way. You can create custom exception filters to catch and handle exceptions gracefully.

```
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const status = exception.getStatus();

    response.status(status).json({
      statusCode: status,
      message: exception.message,
    });
  }
}
```

- **Global and Route-Specific**: Exception filters can be applied globally or to specific routes.

8. Pipes

- **What Are They?**: Pipes are used for data transformation and validation. You can use built-in pipes (e.g., ValidationPipe) or create custom pipes to transform input data.

```
@Post()
create(@Body(new ValidationPipe()) createUserDto: CreateUserDto) {
  return 'User created';
}
```

- **Use Cases**: Common use cases include validating incoming data and transforming data types.

9. Guards

- **What Are They?**: Guards determine whether a request should be processed based on custom authorization logic. They are commonly used for implementing authentication and authorization.

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
    return request.headers.authorization === 'valid-token';
  }
}
```

- **Usage**: Guards can be applied at the method, controller, or global level.

10. Interceptors

- **What Are They?**: Interceptors are used for logging, transforming responses, and handling cross-cutting concerns such as caching or exception handling.

```
@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    console.log('Before handling request...');
    return next.handle().pipe(
      tap(() => console.log('After handling request...')),
    );
  }
}
```

11. Application Lifecycle Hooks

- **What Are They?**: These are methods that you can implement to handle application-level events, such as when the application starts, shuts down, or a module is initialized.
- **Common Hooks**: OnModuleInit, OnModuleDestroy, OnApplicationBootstrap, etc.

12. Microservices Support

- **Built-in Support**: NestJS has built-in support for microservices using different transport layers (e.g., TCP, Redis, MQTT, Kafka), making it easier to create distributed systems.

13. GraphQL Integration

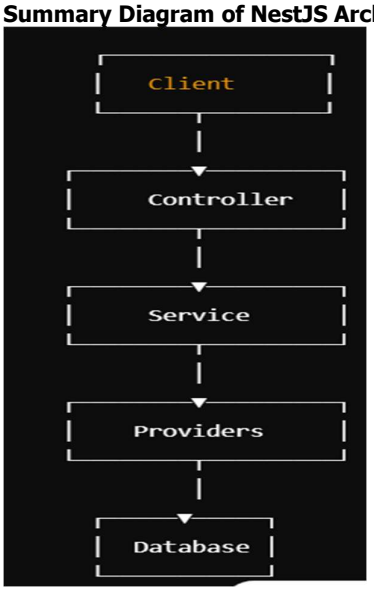
- **GraphQL Module**: NestJS provides seamless integration with GraphQL using the @nestjs/graphql package. You can use decorators to define your GraphQL schema, resolvers, and queries.

14. Testing in NestJS

- **Testing Tools**: NestJS provides out-of-the-box integration with testing tools like Jest, making it easy to write unit and end-to-end tests.

- **Testable Architecture**: The modular design and dependency injection make it simple to isolate and test individual components.

Summary Diagram of NestJS Architecture:



3. Working with Controllers and Routes (5-6 pages)

In NestJS, **controllers** and **routes** are essential components for handling incoming HTTP requests and sending responses back to the client. Controllers define the application's routes, and they determine how requests to a specific endpoint should be handled. Let's dive into how you can work with controllers and routes in NestJS with examples.

1. What Are Controllers in NestJS?

Controllers are responsible for handling incoming requests and returning responses to the client. They define the routes (endpoints) and the corresponding request-handling methods using decorators.

2. Creating a Basic Controller

You can generate a controller using the NestJS CLI:

bash

Copy code

nest generate controller users

Or manually create it:

Example:

```
import { Controller, Get, Post } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Get()
  findAll(): string {
    return 'This action returns all users';
  }

  @Post()
  create(): string {
    return 'This action creates a new user';
  }
}
```

Explanation:

- @Controller('users'): Defines that this controller will handle routes starting with /users.
- @Get(): Maps the HTTP GET request to the findAll() method, which will be accessible via /users.
- @Post(): Maps the HTTP POST request to the create() method, accessible via /users.

3. Route Parameters

NestJS allows you to define route parameters using the @Param() decorator.

Example:

```
import { Controller, Get, Param } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Get(':id')
  findOne(@Param('id') id: string): string {
    return `This action returns user with ID: ${id}`;
  }
}
```

- When you access /users/123, the findOne() method will be invoked with id equal to 123.

4. Query Parameters

You can access query parameters using the @Query() decorator.

Example:

```
import { Controller, Get, Query } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Get()
  findAll(@Query('role') role: string): string {
    return `This action returns all users with role: ${role}`;
  }
}
```

- Accessing /users?role=admin will call findAll() with role equal to admin.

5. Request Body

For handling POST, PUT, or PATCH requests with a body, use the @Body() decorator.

Example:

```
import { Controller, Post, Body } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Post()
  create(@Body() createUserDto: any): string {
    return `User created with name: ${createUserDto.name}`;
  }
}
```

- Sending a POST request with JSON { "name": "John Doe" } will return User created with name: John Doe.

6. Handling Different HTTP Methods

You can handle various HTTP methods like GET, POST, PUT, PATCH, and DELETE using corresponding decorators.

Example:

```
import { Controller, Get, Post, Put, Delete, Param, Body } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Get()
  findAll(): string {
    return 'This action returns all users';
  }

  @Post()
  create(@Body() createUserDto: any): string {
    return `User created with name: ${createUserDto.name}`;
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() updateUserDto: any): string {
    return `User with ID ${id} updated`;
  }

  @Delete(':id')
  remove(@Param('id') id: string): string {
    return `User with ID ${id} deleted`;
  }
}
```

7. Response Handling

You can customize the response using the @Res() decorator to gain direct access to the underlying response object.

Example:

```
import { Controller, Get, Res } from '@nestjs/common';
import { Response } from 'express';

@Controller('users')
export class UsersController {
  @Get()
  findAll(@Res() res: Response) {
    res.status(200).json({ message: 'All users returned successfully' });
  }
}
```

8. NestJS Route Wildcards

You can define wildcard routes by using an asterisk (*), allowing you to catch multiple endpoints.

Example:

```
@Controller('docs')
export class DocsController {
  @Get('*')
  findDocs() {
```

```
  return 'This route handles all /docs/* routes';
 }
}
```

9. Route Grouping and Nesting

You can create nested routes by adding more controllers within a module.

Example:

```
// posts.controller.ts
import { Controller, Get } from '@nestjs/common';

@Controller('posts')
export class PostsController {
 @Get()
 findAll(): string {
  return 'This action returns all posts';
 }
}

// comments.controller.ts
import { Controller, Get } from '@nestjs/common';

@Controller('posts/:postId/comments')
export class CommentsController {
 @Get()
 findAll(): string {
  return 'This action returns all comments for a specific post';
 }
}
```

- Access /posts to get posts, and /posts/:postId/comments to get comments for a specific post.

10. Route Prefixing

You can apply a global prefix to all routes using the app.setGlobalPrefix() method.

Example:

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
 const app = await NestFactory.create(AppModule);
 app.setGlobalPrefix('api/v1'); // All routes will be prefixed with /api/v1
 await app.listen(3000);
}
bootstrap();
```

- With this configuration, /users will now be accessible at /api/v1/users.

11. Using Custom Route Decorators

You can create custom decorators to handle repeated logic.

Example:

```
// admin.decorator.ts
import { SetMetadata } from '@nestjs/common';

export const Admin = () => SetMetadata('role', 'admin');

// users.controller.ts
import { Controller, Get } from '@nestjs/common';
import { Admin } from './admin.decorator';

@Controller('users')
export class UsersController {
 @Get('admin')
 @Admin()
 findAdminUsers(): string {
  return 'This action returns all admin users';
 }
}
```

4. Providers and Dependency Injection (4-5 pages)

Providers and **Dependency Injection (DI)** are core concepts in NestJS that enable building scalable, maintainable, and testable applications. Providers are used to define and manage services, and Dependency Injection is the mechanism that allows NestJS to supply instances of these providers wherever they are needed.

1. What Are Providers?

Providers are classes that handle business logic and can be injected into other parts of the application (e.g., controllers, other providers). They are used to manage dependencies and encapsulate functionality like database operations, utility functions, or third-party integrations.

- A provider is a class decorated with the @Injectable() decorator, indicating that it can be injected into other classes using NestJS's Dependency Injection system.

Example of a Provider:

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

@Injectable()
export class UsersService {
  private users = [{ id: 1, name: 'John Doe' }];

  findAll() {
    return this.users;
  }

  create(user) {
    this.users.push(user);
  }
}
```

2. Registering Providers in Modules

For a provider to be available for injection, it must be registered in a module. Providers are registered in the providers array of the @Module() decorator.

Example:

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';

@Module({
  controllers: [UsersController],
  providers: [UsersService],
})
export class UsersModule {}
```

3. Injecting Providers into Controllers or Other Providers

Once a provider is registered in a module, it can be injected into controllers or other providers using Dependency Injection.

Example: Injecting UsersService into a Controller

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { UsersService } from './users.service';

@Controller('users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Get()
  findAll() {
    return this.usersService.findAll();
  }

  @Post()
  create(@Body() user: any) {
    this.usersService.create(user);
    return 'User created';
  }
}
```

- The UsersService is injected into UsersController via the constructor.
- The private readonly syntax creates a private member usersService and assigns it automatically.

4. Dependency Injection (DI) in NestJS

Dependency Injection is a design pattern where objects are provided with their dependencies instead of creating them directly. NestJS has a built-in DI system that helps manage dependencies efficiently.

How DI Works in NestJS:

1. **Mark Classes as Injectable**: Use the @Injectable() decorator to mark classes that can be injected.
2. **Declare Providers in a Module**: Register providers in the module where they should be available.
3. **Inject Providers Using Constructor Injection**: Use the constructor to inject providers into controllers or other services.

5. Scopes in Providers

By default, providers are **singleton**, meaning a single instance is shared across the entire application. However, NestJS allows defining different scopes for providers:

- **Singleton Scope (Default)**: A single instance is shared across the application.
- **Request Scope**: A new instance is created for each incoming request.
- **Transient Scope**: A new instance is created each time it is injected.

Example of Request Scope:

```
@Injectable({ scope: Scope.REQUEST })
export class RequestScopedService {
  constructor() {
```

```
    console.log('New instance created for each request');
  }
}
```

6. Custom Providers

In addition to the default way of creating providers, you can also define custom providers, allowing you to control how instances are created.

Example: Using a Custom Value

```
const DatabaseConnection = {
  provide: 'DATABASE_CONNECTION',
  useValue: { connect: () => 'Database Connected' },
};

@Module({
  providers: [DatabaseConnection],
})
export class AppModule {}
```

You can inject the custom provider using the @Inject() decorator:

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

@Injectable()
export class AppService {
  constructor(@Inject('DATABASE_CONNECTION') private dbConnection) {}

  getDatabaseStatus() {
    return this.dbConnection.connect();
  }
}
```

7. Using useClass, useExisting, useFactory, and useValue

Providers can be defined in multiple ways:

- **useClass**: The default way of providing a class as a provider.
- **useExisting**: Reuses an existing provider.
- **useFactory**: Uses a factory function to create a provider.
- **useValue**: Provides a fixed value.

Example of useFactory:

```
const CustomProvider = {
  provide: 'CUSTOM_PROVIDER',
  useFactory: () => {
    return new Date().toISOString(); // Returns current date as a string
  },
};

@Module({
  providers: [CustomProvider],
})
export class AppModule {}
```

8. Providers with Dependencies

Providers can depend on other providers, and NestJS's DI system will resolve these dependencies automatically.

Example:

```
@Injectable()
export class OrdersService {
  constructor(private readonly usersService: UsersService) {}

  createOrder() {
    const users = this.usersService.findAll();
    return `Order created for user: ${users[0].name}`;
  }
}

@Module({
  providers: [OrdersService, UsersService],
})
export class OrdersModule {}
```

- OrdersService depends on UsersService, and NestJS injects it automatically.

9. Using Async Providers

In some cases, you may need to initialize a provider asynchronously, for example, when establishing a database connection.

Example:

```
const AsyncProvider = {
  provide: 'ASYNC_CONNECTION',
  useFactory: async () => {
    const connection = await createConnection(); // Some async operation
```

```
  return connection;
 },
};

@Module({
 providers: [AsyncProvider],
})
export class AppModule {}
```

10. Testing Providers

One of the major benefits of DI is that it makes testing easier. You can mock dependencies when testing your providers.

Example: Unit Testing with Mocks

```
import { Test, TestingModule } from '@nestjs/testing';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';

describe('UsersController', () => {
 let usersController: UsersController;
 let usersService: UsersService;

 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   controllers: [UsersController],
   providers: [
    {
     provide: UsersService,
     useValue: {
      findAll: jest.fn().mockReturnValue([{ id: 1, name: 'John Doe' }]),
     },
    },
   ],
  }).compile();

  usersController = module.get<UsersController>(UsersController);
  usersService = module.get<UsersService>(UsersService);
 });

 it('should return all users', async () => {
  expect(await usersController.findAll()).toEqual([{ id: 1, name: 'John Doe' }]);
 });
});
```

- Here, UsersService is mocked to return predefined data, making it easy to test UsersController in isolation.

- **Scopes of Providers**: Singleton, Request, and Transient scopes

1. Singleton Scope (Default)

- **Description**: In the Singleton scope, a single instance of the provider is created and shared across the entire application. It's the default scope in NestJS, meaning you don't have to specify this explicitly.
- **Use Case**: When you want a service to be used globally or maintain state throughout the application's lifecycle. For example, for caching, configuration, or logging services.

Example:

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

@Injectable()
export class SingletonService {
 private count = 0;

 increment() {
  this.count++;
  return this.count;
 }
}
```

- If multiple controllers or services inject SingletonService, they will all share the same instance, and the count value will be maintained across them.

Usage in a Controller:

```
import { Controller, Get } from '@nestjs/common';
import { SingletonService } from './singleton.service';

@Controller('singleton')
export class SingletonController {
 constructor(private readonly singletonService: SingletonService) {}

 @Get()
```

```
  getIncrementedValue() {
    return this.singletonService.increment();
  }
}
```

- Each time the endpoint is called, count will be incremented and maintained since it's a shared instance.

2. Request Scope

- **Description**: In the Request scope, a new instance of the provider is created for each incoming request. This means that if multiple requests hit the endpoint simultaneously, each request will have its instance of the provider.
- **Use Case**: Useful when you need to manage request-specific data, like handling session information, request tracking, or logging data unique to that request.

Example:

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

@Injectable({ scope: Scope.REQUEST })
export class RequestScopedService {
  private requestTime: Date;

  constructor() {
    this.requestTime = new Date();
  }

  getRequestTime() {
    return this.requestTime;
  }
}
```

Usage in a Controller:

```
import { Controller, Get } from '@nestjs/common';
import { RequestScopedService } from './request-scoped.service';

@Controller('request-scope')
export class RequestScopeController {
  constructor(private readonly requestScopedService: RequestScopedService) {}

  @Get()
  getRequestTime() {
    return this.requestScopedService.getRequestTime();
  }
}
```

- Each time you make a request to /request-scope, a new instance of RequestScopedService will be created, and requestTime will be different for each request.

3. Transient Scope

- **Description**: In the Transient scope, a new instance of the provider is created each time it is injected. Unlike Singleton and Request scopes, where instances might be reused, Transient providers are always freshly instantiated whenever needed.
- **Use Case**: Ideal when you need a new instance every time a service is required, such as when dealing with isolated calculations, generating unique identifiers, or working with temporary, non-shared data.

Example:

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

@Injectable({ scope: Scope.TRANSIENT })
export class TransientService {
  private id: number;

  constructor() {
    this.id = Math.floor(Math.random() * 1000);
  }

  getId() {
    return this.id;
  }
}
```

Usage in a Controller:

```
import { Controller, Get } from '@nestjs/common';
import { TransientService } from './transient.service';

@Controller('transient')
export class TransientController {
  constructor(private readonly transientService: TransientService) {}

  @Get()
```

```
  getUniqueId() {
    return this.transientService.getId();
  }
}
```

- If multiple controllers or services inject TransientService, each injection will result in a new instance, and the id value will be different for each instance.

Comparing Scopes

Scope	Description	Instance Creation	Use Cases
Singleton	Default scope, shared instance across app	One instance per application	Caching, logging, configuration
Request	New instance per request	One instance per incoming request	Request-specific data, session handling
Transient	New instance each time it's injected	Multiple instances, one for each injection	Isolated tasks, temporary data

How to Use Scopes in NestJS

- The scope of a provider is specified using the @Injectable() decorator's options:

```
@Injectable({ scope: Scope.REQUEST })
```

- Import Scope from @nestjs/common and choose between Scope.SINGLETON, Scope.REQUEST, or Scope.TRANSIENT.

Real-world Example Combining Scopes

Imagine you have an application that processes user orders. You might have:

- A **Singleton** ConfigService that holds global configuration.
- A **Request-scoped** LoggerService that logs request-specific data for auditing.
- A **Transient** DiscountService that generates a unique discount code each time it's used.

Example Code

```
// ConfigService (Singleton)
@Injectable()
export class ConfigService {
  getConfig() {
    return 'Global configuration';
  }
}

// LoggerService (Request Scope)
@Injectable({ scope: Scope.REQUEST })
export class LoggerService {
  private logs: string[] = [];

  log(message: string) {
    this.logs.push(message);
  }

  getLogs() {
    return this.logs;
  }
}

// DiscountService (Transient)
@Injectable({ scope: Scope.TRANSIENT })
export class DiscountService {
  private discountCode: string;

  constructor() {
    this.discountCode = Math.random().toString(36).substring(7);
  }

  getDiscountCode() {
    return this.discountCode;
  }
}

// OrdersController
@Controller('orders')
export class OrdersController {
  constructor(
    private readonly configService: ConfigService,
    private readonly loggerService: LoggerService,
    private readonly discountService: DiscountService,
  ) {}

  @Get()
```

```
  processOrder() {
    this.loggerService.log('Processing order');
    return {
      config: this.configService.getConfig(),
      logs: this.loggerService.getLogs(),
      discountCode: this.discountService.getDiscountCode(),
    };
  }
}
```

Explanation:

- The ConfigService is shared across all requests and is injected as a singleton.
- A new LoggerService is created for each request, allowing separate logging per request.
- A new DiscountService instance is created each time it is injected, resulting in a unique discount code every time the endpoint is accessed.

5. Modules Deep Dive (5-6 pages)

1. What is a Module in NestJS?

In NestJS, a module is a class annotated with a @Module decorator. It acts as a container for a specific set of functionalities, encapsulating providers, controllers, and other modules. This modular architecture enhances the scalability and maintainability of applications.

2. Creating a Module

To create a module, you typically use the Nest CLI:

```
nest generate module users
```

This command generates a users module with a file named users.module.ts.

Basic Structure

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
  imports: [], // Other modules needed
  controllers: [UsersController], // Controllers for this module
  providers: [UsersService], // Providers (services) for this module
  exports: [UsersService], // Exported providers
})
export class UsersModule {}
```

3. Module Components

- **Controllers:** Handle incoming requests and return responses. They are defined within the controllers array of the module.
- **Providers:** Services that contain the business logic. They are defined within the providers array. Providers can be injected into controllers or other providers.
- **Imports:** Specifies other modules that this module depends on. Use this array to import other modules.
- **Exports:** Allows other modules to access the providers declared in this module.

4. Module Nesting

Modules can import other modules, creating a tree-like structure. This allows for better organization of related functionality.

```
import { Module } from '@nestjs/common';
import { UsersModule } from './users/users.module';
import { ProductsModule } from './products/products.module';

@Module({
  imports: [UsersModule, ProductsModule],
})
export class AppModule {}
```

5. Dynamic Modules

Dynamic modules allow you to create modules that can be configured at runtime. This is useful when you need to pass in options or configurations.

```
import { Module, DynamicModule } from '@nestjs/common';

@Module({})
export class ConfigurableModule {
  static register(options: any): DynamicModule {
    return {
      module: ConfigurableModule,
      providers: [
        {
          provide: 'CONFIG_OPTIONS',
          useValue: options,
        },
      ],
      exports: ['CONFIG_OPTIONS'],
```

```
  };
 }
}
```

6. Global Modules

If you have a module that should be available throughout the application, you can mark it as global. This means its providers will be accessible in all other modules without needing to import it.

```
import { Module, Global } from '@nestjs/common';

@Global()
@Module({
 providers: [SomeGlobalService],
 exports: [SomeGlobalService],
})
export class GlobalModule {}
```

7. Providers

Providers are classes annotated with the @Injectable decorator. They can be services, repositories, or any class that encapsulates logic.

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

@Injectable()
export class UsersService {
 findAll() {
  return [];
 }
}
```

8. Dependency Injection

NestJS uses dependency injection (DI) to manage the lifecycle of classes. When you define providers in a module, they can be injected into controllers or other providers.

```
import { Controller, Get } from '@nestjs/common';

@Controller('users')
export class UsersController {
 constructor(private readonly usersService: UsersService) {}

 @Get()
 findAll() {
  return this.usersService.findAll();
 }
}
```

9. Module Communication

Modules can communicate by exporting providers from one module and importing them into another. This is how you share functionality across different parts of your application.

10. Testing Modules

When testing modules, you can use the Test.createTestingModule method to create a testing module that allows you to mock providers or import other modules as needed.

```
import { Test, TestingModule } from '@nestjs/testing';
import { UsersService } from './users.service';

describe('UsersService', () => {
 let service: UsersService;

 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   providers: [UsersService],
  }).compile();

  service = module.get<UsersService>(UsersService);
 });

 it('should be defined', () => {
  expect(service).toBeDefined();
 });
});
```

6. Middleware in NestJS (3-4 pages)

Middleware in NestJS allows you to execute code before the request is processed by the route handler. This can be useful for various tasks such as logging, authentication, or modifying the request and response objects. Here's a detailed overview of how to create and use middleware in NestJS:

1. What is Middleware?

Middleware is a function that has access to the request (req), response (res), and the next middleware function in the application's request-response cycle. It can perform operations on the request or response, end the request-response cycle, or call the next middleware function.

2. Creating Middleware

You can create middleware in NestJS by implementing the NestMiddleware interface or by creating a simple function.

Using NestMiddleware Interface

To create a middleware class, you need to implement the NestMiddleware interface:

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log(`Request... ${req.method} ${req.url}`);
    next(); // Call the next middleware or route handler
  }
}
```

Using a Function

You can also create middleware as a plain function without creating a class:

```
import { Request, Response, NextFunction } from 'express';

export function logger(req: Request, res: Response, next: NextFunction) {
  console.log(`Request... ${req.method} ${req.url}`);
  next();
}
```

3. Applying Middleware

Middleware can be applied at the module level, globally, or to specific routes.

a. Module-Level Middleware

To apply middleware to a specific module, use the configure method in the module class:

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';
import { LoggerMiddleware } from './logger.middleware';

@Module({
  controllers: [UsersController],
  providers: [UsersService],
})
export class UsersModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes(UsersController); // Apply to specific routes or controllers
  }
}
```

b. Global Middleware

To apply middleware globally, you can use the app.use() method in your main application file (usually main.ts):

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { LoggerMiddleware } from './logger.middleware';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(LoggerMiddleware); // Apply globally
  await app.listen(3000);
}
bootstrap();
```

4. Middleware Order

Middleware is executed in the order they are added. If a middleware does not call next(), the request will be terminated, and no subsequent middleware or route handlers will be executed.

5. Using Middleware with Specific Routes

You can apply middleware to specific HTTP methods or routes:

```
consumer
  .apply(LoggerMiddleware)
  .forRoutes({ path: 'users', method: RequestMethod.GET }); // Only for GET requests on /users
```

6. Accessing Request and Response Objects

Inside middleware, you can access the request and response objects to modify them or add data:

```
use(req: Request, res: Response, next: NextFunction) {
  req.user = { id: 1 }; // Add custom property to request object
  next();
}
```

7. Error Handling in Middleware

If you want to handle errors in middleware, you can create an error-handling middleware function:

```
export function errorHandlingMiddleware(err, req: Request, res: Response, next: NextFunction) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
}
```

8. Chaining Middleware

You can chain multiple middleware functions by applying them one after the other:

```
consumer
  .apply(LoggerMiddleware, AnotherMiddleware)
  .forRoutes(UsersController);
```

9. Creating Custom Middleware

Custom middleware can be created either as a class implementing NestMiddleware or as a simple function.

Creating Middleware as a Class

To create middleware as a class, follow these steps:

1. Import the necessary modules from NestJS and Express.
2. Implement the NestMiddleware interface.
3. Define your logic in the use method.

```
// logger.middleware.ts
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log(`[${new Date().toISOString()}] ${req.method} ${req.originalUrl}`);
    next(); // Pass control to the next middleware or route handler
  }
}
```

Creating Middleware as a Function

You can also create middleware as a simple function, which may be more suitable for small or reusable logic.

```
// logger.middleware.ts
import { Request, Response, NextFunction } from 'express';

export function logger(req: Request, res: Response, next: NextFunction) {
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.originalUrl}`);
  next(); // Pass control to the next middleware or route handler
}
```

Applying Custom Middleware

You can apply your custom middleware at the module level, globally, or to specific routes.

Applying Middleware at the Module Level

To apply the middleware to a specific module, you need to implement the NestModule interface and use the MiddlewareConsumer:

```
// app.module.ts
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { UsersModule } from './users/users.module';
import { LoggerMiddleware } from './logger.middleware';

@Module({
  imports: [UsersModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('*'); // Apply to all routes or specify the controller like 'UsersController'
  }
}
```

You can apply the middleware to a specific route or controller by using:

```
consumer
  .apply(LoggerMiddleware)
  .forRoutes('users'); // Applies to routes starting with /users
```

You can also apply it to specific HTTP methods:

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

consumer
  .apply(LoggerMiddleware)
  .forRoutes({ path: 'users', method: RequestMethod.GET }); // Only for GET /users route
```

Applying Middleware Globally

To apply the middleware globally, you can do so in your main.ts file:

```
// main.ts
import { NestFactory } from '@nestjs/core';
```

```
import { AppModule } from './app.module';
import { LoggerMiddleware } from './logger.middleware';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Apply the logger middleware globally
  app.use(new LoggerMiddleware().use); // For class-based middleware
  // or app.use(logger); // For function-based middleware

  await app.listen(3000);
}
bootstrap();
```

Using Dependency Injection in Middleware

Because NestJS middleware can be treated like any other provider, you can inject dependencies into middleware using @Injectable().

Example with Dependency Injection:

```
// auth.middleware.ts
import { Injectable, NestMiddleware, Inject } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';
import { AuthService } from './auth.service';

@Injectable()
export class AuthMiddleware implements NestMiddleware {
  constructor(private readonly authService: AuthService) {}

  async use(req: Request, res: Response, next: NextFunction) {
    const token = req.headers['authorization'];

    if (!token) {
      return res.status(403).json({ message: 'Forbidden' });
    }

    const isValid = await this.authService.validateToken(token);
    if (!isValid) {
      return res.status(401).json({ message: 'Unauthorized' });
    }

    next();
  }
}
```

In this example, the AuthService is injected into the AuthMiddleware to handle the token validation logic.

Order of Middleware Execution

Middleware is executed in the order in which they are applied. If multiple middleware functions are chained, the first one will run first, followed by the next, until the end of the chain or when next() is called.

10. Third part middlewares

1. Installing Third-Party Middleware

First, you need to install the middleware package via npm. Let's take a look at a few popular examples:

- **Body-parser**: For parsing incoming request bodies.
- **Cors**: For enabling Cross-Origin Resource Sharing (CORS).
- **Helmet**: For securing HTTP headers.

For instance, if you want to use helmet and body-parser, you would install them like this:

```
npm install body-parser helmet
```

2. Applying Third-Party Middleware Globally

You can apply middleware globally in your main.ts file using the app.use() method. This method is particularly useful for applying middleware across your entire application.

Example with body-parser and helmet:

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as bodyParser from 'body-parser';
import * as helmet from 'helmet';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Apply body-parser middleware
  app.use(bodyParser.json()); // To parse JSON bodies
  app.use(bodyParser.urlencoded({ extended: true })); // To parse URL-encoded bodies

  // Apply helmet middleware
```

```
 app.use(helmet()); // Secure HTTP headers

 await app.listen(3000);
}
bootstrap();
```

3. Applying Third-Party Middleware at the Module Level

If you want to apply middleware to a specific module or set of routes, use the configure() method of the NestModule interface within your module.

Example: Applying helmet to a specific module

1. First, import MiddlewareConsumer and NestModule:
2. Use the configure() method to apply the middleware:

```
// app.module.ts
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import * as helmet from 'helmet';
import { UsersModule } from './users/users.module';

@Module({
 imports: [UsersModule],
})
export class AppModule implements NestModule {
 configure(consumer: MiddlewareConsumer) {
  consumer
   .apply(helmet())
   .forRoutes('*'); // Applies helmet to all routes in this module
 }
}
```

4. Applying Middleware with Specific Routes and HTTP Methods

You can also target specific routes or HTTP methods when applying third-party middleware.

Example: Apply body-parser to a specific route

```
import { Module, NestModule, MiddlewareConsumer, RequestMethod } from '@nestjs/common';
import * as bodyParser from 'body-parser';

@Module({})
export class AppModule implements NestModule {
 configure(consumer: MiddlewareConsumer) {
  consumer
   .apply(bodyParser.json())
   .forRoutes({ path: 'users', method: RequestMethod.POST }); // Applies only to POST /users
 }
}
```

5. Integrating Other Common Third-Party Middlewares

Here are some examples of other popular third-party middlewares you might use:

a. Morgan (HTTP request logger)

1. Install morgan:

```
npm install morgan
```

2. Apply it globally or at the module level:

```
// main.ts
import * as morgan from 'morgan';

app.use(morgan('combined')); // Use 'tiny', 'combined', or any morgan format
```

b. Compression (Gzip compression for response bodies)

1. Install compression:

```
npm install compression
```

2. Apply it:

```
// main.ts
import * as compression from 'compression';

app.use(compression());
```

c. Session Management (express-session)

1. Install express-session:

```
npm install express-session
```

2. Apply it:

```
// main.ts
import * as session from 'express-session';

app.use(session({
 secret: 'my-secret',
 resave: false,
 saveUninitialized: false,
 cookie: { secure: false }, // Use secure: true in production
```

```
}));
```

7. Exception Filters (3-4 pages)

1. What are Exception Filters?

Exception filters are used to catch unhandled exceptions thrown in your application and provide a way to respond with custom logic. When an exception occurs, the filter captures it and allows you to modify the response sent back to the client. NestJS comes with a default exception filter, but you can create custom ones for more control.

2. Built-in Exception Handling

By default, NestJS has a global exception filter that handles errors and sends a response with a status code and message. For example, if you throw an HttpException, the built-in filter catches it and returns an appropriate response.

```
import { HttpException, HttpStatus } from '@nestjs/common';

throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
This would result in a response with a status code of 403 and a JSON body:
json
Copy code
{
  "statusCode": 403,
  "message": "Forbidden"
}
```

3. Creating a Custom Exception Filter

You can create a custom exception filter by implementing the ExceptionFilter interface. Here's a step-by-step guide:

a. Create the Custom Filter

1. Create a new filter file (e.g., http-exception.filter.ts):

```
// http-exception.filter.ts
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
  HttpException,
  HttpStatus,
} from '@nestjs/common';

@Catch()
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: any, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const request = ctx.getRequest();
    const status =
      exception instanceof HttpException
        ? exception.getStatus()
        : HttpStatus.INTERNAL_SERVER_ERROR;

    const responseBody = {
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      message: exception.message || 'Internal server error',
    };

    response.status(status).json(responseBody);
  }
}
```

In this example, the @Catch() decorator allows you to specify which exceptions to catch. Leaving it empty (@Catch()) means it will catch all exceptions.

b. Apply the Custom Filter

You can apply the filter at different levels:

- **At the Controller Level**:

```
import { Controller, Get, UseFilters } from '@nestjs/common';
import { HttpExceptionFilter } from './http-exception.filter';

@Controller('users')
@UseFilters(HttpExceptionFilter)
export class UsersController {
  @Get()
  findAll() {
    throw new HttpException('User not found', HttpStatus.NOT_FOUND);
  }
}
```

- **At the Method Level**:

```
import { Controller, Get, UseFilters } from '@nestjs/common';
```

```
@Controller('products')
export class ProductsController {
 @Get()
 @UseFilters(HttpExceptionFilter)
 findAll() {
  throw new HttpException('Product not found', HttpStatus.NOT_FOUND);
 }
}
```

- **Globally**:

To apply the exception filter globally, you can do it in your main.ts:

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { HttpExceptionFilter } from './http-exception.filter';

async function bootstrap() {
 const app = await NestFactory.create(AppModule);

 app.useGlobalFilters(new HttpExceptionFilter()); // Applying filter globally

 await app.listen(3000);
}
bootstrap();
```

4. Handling Specific Exceptions

You can create exception filters that handle specific types of exceptions by passing the exception class to the @Catch() decorator:

```
import { ExceptionFilter, Catch, ArgumentsHost, NotFoundException } from '@nestjs/common';

@Catch(NotFoundException)
export class NotFoundExceptionFilter implements ExceptionFilter {
 catch(exception: NotFoundException, host: ArgumentsHost) {
  const ctx = host.switchToHttp();
  const response = ctx.getResponse();
  const request = ctx.getRequest();

  response.status(404).json({
   statusCode: 404,
   error: 'Resource Not Found',
   path: request.url,
   timestamp: new Date().toISOString(),
  });
 }
}
```

This filter will only catch NotFoundException instances and respond with a custom message.

5. Using Built-in HTTP Exceptions

NestJS provides built-in HTTP exceptions you can throw in your application:

- BadRequestException
- UnauthorizedException
- ForbiddenException
- NotFoundException
- ConflictException
- InternalServerErrorException

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

throw new BadRequestException('Invalid data provided');
```

6. Exception Filter for GraphQL (if you're using GraphQL)

If you’re working with GraphQL, the process is similar, but you’ll handle exceptions differently since the context is not HTTP:

```
import {
 ArgumentsHost,
 Catch,
 ExceptionFilter,
 GqlArgumentsHost,
} from '@nestjs/common';
import { ApolloError } from 'apollo-server-express';

@Catch()
export class GraphQLExceptionFilter implements ExceptionFilter {
 catch(exception: any, host: ArgumentsHost) {
  const gqlHost = GqlArgumentsHost.create(host);
```

```
    return new ApolloError(exception.message, exception.extensions?.code, exception.extensions);
  }
}
```

7. Handling Logging in Exception Filters

Exception filters can also be used to log errors. You can inject NestJS's LoggerService or create your own logging mechanism.

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

export class HttpExceptionFilter implements ExceptionFilter {
  private readonly logger = new Logger(HttpExceptionFilter.name);

  catch(exception: any, host: ArgumentsHost) {
    this.logger.error(`Exception thrown: ${exception.message}`);
    // ... other exception handling logic
  }
}
```

8. Pipes and Validation (4-5 pages)

1. What are Pipes in NestJS?

A pipe in NestJS can be used for two main purposes:

1. **Transformation**: Transforming input data into the desired format (e.g., converting strings to integers).
2. **Validation**: Validating the input data to ensure it meets specific requirements (e.g., checking if a string is a valid email).

2. Built-in Pipes

NestJS comes with a few built-in pipes for common transformations and validations:

- **ValidationPipe**: Used for validating incoming request data.
- **ParseIntPipe**: Converts a string to an integer.
- **ParseBoolPipe**: Converts a string to a boolean.
- **ParseUUIDPipe**: Validates and converts a UUID string.
- **DefaultValuePipe**: Provides a default value if the input is undefined or null.

3. Setting Up Validation Using ValidationPipe

The ValidationPipe is the most commonly used pipe for validating incoming request data. It works with class-validator and class-transformer libraries to perform validation based on decorators applied to your DTO (Data Transfer Object) classes.

Step 1: Install Dependencies

```
npm install class-validator class-transformer
```

Step 2: Create a DTO (Data Transfer Object)

Define a DTO with validation rules using decorators from class-validator:

```
// create-user.dto.ts
import { IsString, IsInt, IsEmail, Length, Min } from 'class-validator';

export class CreateUserDto {
  @IsString()
  @Length(3, 20)
  name: string;

  @IsInt()
  @Min(18)
  age: number;

  @IsEmail()
  email: string;
}
```

In this example:

- @IsString() ensures name is a string.
- @Length(3, 20) ensures name has a length between 3 and 20.
- @IsInt() ensures age is an integer.
- @Min(18) ensures age is at least 18.
- @IsEmail() ensures email is a valid email address.

Step 3: Use ValidationPipe in Your Controller

You can apply ValidationPipe at the controller method level, controller level, or globally:

a. Applying at the Route Handler Level

```
// users.controller.ts
import { Controller, Post, Body, UsePipes } from '@nestjs/common';
import { CreateUserDto } from './create-user.dto';

@Controller('users')
export class UsersController {
  @Post()
  @UsePipes(new ValidationPipe())
  createUser(@Body() createUserDto: CreateUserDto) {
    return `User created: ${createUserDto.name}`;
  }
```

```
}
```

b. Applying at the Controller Level

```
@Controller('users')
@UsePipes(new ValidationPipe())
export class UsersController {
  @Post()
  createUser(@Body() createUserDto: CreateUserDto) {
    return `User created: ${createUserDto.name}`;
  }
}
```

c. Applying Globally

To apply ValidationPipe globally, add it in your main.ts file:

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

4. Transformation Using ValidationPipe

By default, data passed to your route handlers is not automatically transformed to the appropriate types. However, ValidationPipe can handle transformation when you enable transform: true:

```
app.useGlobalPipes(new ValidationPipe({ transform: true }));
```

With this option enabled, if you have an endpoint expecting an integer (@IsInt()), and a string "25" is sent in the request body, it will be automatically converted to an integer.

5. Customizing ValidationPipe

You can customize the behavior of ValidationPipe by passing options:

```
app.useGlobalPipes(
  new ValidationPipe({
    transform: true,
    whitelist: true, // Strip properties not included in the DTO
    forbidNonWhitelisted: true, // Throw an error if any non-whitelisted property is provided
    transformOptions: { enableImplicitConversion: true }, // Automatically convert primitive types
  }),
);
```

- whitelist: Removes properties that are not included in the DTO.
- forbidNonWhitelisted: Throws an error when non-whitelisted properties are present.
- transformOptions: Converts properties based on the expected type.

6. Creating Custom Pipes

You can create custom pipes by implementing the PipeTransform interface.

Example: Custom ParseIntPipe

```
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException } from '@nestjs/common';

@Injectable()
export class CustomParseIntPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException(`Validation failed. "${value}" is not an integer.`);
    }
    return val;
  }
}
```

You can then use this custom pipe in your controller:

```
@Controller('items')
export class ItemsController {
  @Get(':id')
  getItem(@Param('id', new CustomParseIntPipe()) id: number) {
    return `Item with ID: ${id}`;
  }
}
```

7. Combining Multiple Pipes

You can apply multiple pipes to a single parameter or controller method:

```
@Get(':id')
getItem(@Param('id', new ParseIntPipe(), new CustomValidationPipe()) id: number) {
  return `Item with ID: ${id}`;
```

```
}
```

8. Handling Query Parameters and Route Parameters

You can use pipes for transforming and validating query parameters (@Query()) or route parameters (@Param()):

```
@Get()
getItems(@Query('limit', ParseIntPipe) limit: number) {
  return `Getting ${limit} items`;
}
```

9. Guards for Authorization (3-4 pages)

Guards in NestJS are an essential feature for implementing authorization and protecting your routes from unauthorized access. They provide a way to control access to certain parts of your application by determining whether a request should be processed by the route handler based on specific criteria, such as user roles, authentication status, or any other custom logic.

1. What Are Guards in NestJS?

Guards are classes that implement the CanActivate interface. They are used to make authorization decisions based on the request context. A guard's primary purpose is to determine whether a given request will be allowed to proceed to the route handler.

- If a guard returns true or resolves successfully, the request is allowed to proceed.
- If it returns false or throws an exception, the request is denied.

2. Creating a Custom Guard

Here's how you can create a simple custom guard to check if a user is authenticated.

Step 1: Implementing the Guard

1. Create a new guard file, e.g., auth.guard.ts:

```
// auth.guard.ts
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return this.validateRequest(request);
  }

  validateRequest(request: any): boolean {
    // Custom logic to check if the user is authenticated
    return request.isAuthenticated && request.isAuthenticated();
  }
}
```

In this example, the AuthGuard checks whether the request is authenticated by calling request.isAuthenticated(). This method can be part of popular libraries like passport.

Step 2: Applying the Guard

You can apply the guard at different levels:

- **Route Handler Level**:

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { AuthGuard } from './auth.guard';

@Controller('profile')
export class ProfileController {
  @Get()
  @UseGuards(AuthGuard)
  getProfile() {
    return 'This is a protected profile';
  }
}
```

- **Controller Level**:

```
@Controller('profile')
@UseGuards(AuthGuard)
export class ProfileController {
  @Get()
  getProfile() {
    return 'This is a protected profile';
  }
}
```

- **Global Level**:

To apply the guard globally, modify the main.ts file:

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { AuthGuard } from './auth.guard';
```

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalGuards(new AuthGuard());
  await app.listen(3000);
}
bootstrap();
```

3. Role-Based Authorization

You can extend guards to perform role-based authorization, ensuring that only users with specific roles can access certain routes.

Step 1: Create a Roles Guard

1. Create a roles.guard.ts file:

```
// roles.guard.ts
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Reflector } from '@nestjs/core';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<string[]>('roles', context.getHandler());
    if (!roles) {
      return true; // No roles required, allow access
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;

    return user && user.roles && roles.some(role => user.roles.includes(role));
  }
}
```

In this example, the RolesGuard checks if the user has at least one of the required roles by leveraging NestJS's Reflector to retrieve metadata set on the route.

Step 2: Define a Roles Decorator

Create a decorator to set the required roles:

```
// roles.decorator.ts
import { SetMetadata } from '@nestjs/common';

export const Roles = (...roles: string[]) => SetMetadata('roles', roles);
```

Step 3: Apply the RolesGuard and Roles Decorator

Use the Roles decorator and RolesGuard in your controller:

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { Roles } from './roles.decorator';
import { RolesGuard } from './roles.guard';

@Controller('admin')
@UseGuards(RolesGuard)
export class AdminController {
  @Get()
  @Roles('admin')
  getAdminData() {
    return 'This is admin data';
  }
}
```

In this example, only users with the admin role can access the getAdminData() route.

4. Combining Multiple Guards

You can combine multiple guards by using the @UseGuards() decorator with multiple arguments:

```
@Controller('protected')
@UseGuards(AuthGuard, RolesGuard)
export class ProtectedController {
  @Get()
  @Roles('admin', 'manager')
  getProtectedData() {
    return 'This data is protected by both authentication and role-based authorization';
  }
}
```

5. Handling Authentication with Passport.js

NestJS integrates well with passport for handling authentication. Here's how you can create an AuthGuard using Passport strategies:

Step 1: Install Passport and Passport-JWT

```
npm install @nestjs/passport passport passport-jwt
npm install --save-dev @types/passport-jwt
```

Step 2: Implement a JWT Auth Guard

1. Create a jwt-auth.guard.ts file:

```
// jwt-auth.guard.ts
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

2. Create a JWT strategy (jwt.strategy.ts):

```
// jwt.strategy.ts
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: 'your_jwt_secret_key', // Replace with your secret key
    });
  }

  async validate(payload: any) {
    return { userId: payload.sub, username: payload.username, roles: payload.roles };
  }
}
```

3. Register the strategy in your module (auth.module.ts):

```
import { Module } from '@nestjs/common';
import { PassportModule } from '@nestjs/passport';
import { JwtStrategy } from './jwt.strategy';

@Module({
  imports: [PassportModule],
  providers: [JwtStrategy],
  exports: [PassportModule],
})
export class AuthModule {}
```

4. Apply the JwtAuthGuard in your controller:

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { JwtAuthGuard } from './jwt-auth.guard';

@Controller('profile')
export class ProfileController {
  @Get()
  @UseGuards(JwtAuthGuard)
  getProfile() {
    return 'This is a protected profile';
  }
}
```

6. Custom Error Handling in Guards

You can throw exceptions from guards to provide custom error messages using NestJS's built-in UnauthorizedException or ForbiddenException:

```
import { Injectable, CanActivate, ExecutionContext, ForbiddenException } from '@nestjs/common';

@Injectable()
export class CustomGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
    if (!request.user || !request.user.isActive) {
      throw new ForbiddenException('User is inactive and cannot access this route');
    }
    return true;
  }
}
```

10. Interceptors and Enhancing Response (3-4 pages)

Interceptors in NestJS are a powerful feature that allows you to modify the incoming request and the outgoing response at different stages of the request lifecycle. They can be used for a variety of purposes, including logging, transforming responses, adding additional metadata, or implementing caching and validation logic.

1. What Are Interceptors?

Interceptors are classes that implement the NestInterceptor interface. They can be used to:

- Transform the response data.
- Add extra logic before or after the method execution.
- Bind extra logic to the request-response cycle.
- Handle exceptions and modify the response.

2. Creating an Interceptor

Here's how to create a basic interceptor that modifies the response before it is sent to the client.

Step 1: Implementing the Interceptor

1. Create a new interceptor file, e.g., transform.interceptor.ts:

```
// transform.interceptor.ts
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  CallHandler,
} from '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class TransformInterceptor<T> implements NestInterceptor<T, T> {
  intercept(context: ExecutionContext, next: CallHandler): Observable<T> {
    return next.handle().pipe(
      map(data => ({
        statusCode: 200,
        data,
        message: 'Request was successful',
      })),
    );
  }
}
```

In this example, the TransformInterceptor modifies the outgoing response to wrap the original data in an object that includes a statusCode and a message.

Step 2: Applying the Interceptor

You can apply the interceptor at different levels:

- **Method Level**:

```
import { Controller, Get, UseInterceptors } from '@nestjs/common';
import { TransformInterceptor } from './transform.interceptor';

@Controller('users')
export class UsersController {
  @Get()
  @UseInterceptors(TransformInterceptor)
  getAllUsers() {
    return [{ id: 1, name: 'John Doe' }, { id: 2, name: 'Jane Doe' }];
  }
}
```

- **Controller Level**:

```
@Controller('users')
@UseInterceptors(TransformInterceptor)
export class UsersController {
  @Get()
  getAllUsers() {
    return [{ id: 1, name: 'John Doe' }, { id: 2, name: 'Jane Doe' }];
  }
}
```

- **Global Level**:

To apply an interceptor globally, modify your main.ts file:

```
// main.ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { TransformInterceptor } from './transform.interceptor';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalInterceptors(new TransformInterceptor());
  await app.listen(3000);
}
bootstrap();
```

3. Using Interceptors for Caching

Interceptors can also be used to implement caching logic. You can cache the response based on certain criteria and return the cached response for subsequent requests.

Example: Caching Interceptor

```
// cache.interceptor.ts
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  CallHandler,
} from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class CacheInterceptor implements NestInterceptor {
  private cache = new Map();

  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const request = context.switchToHttp().getRequest();
    const response = context.switchToHttp().getResponse();
    const key = request.url;

    // Check if the response is in cache
    if (this.cache.has(key)) {
      return this.cache.get(key);
    }

    return next.handle().pipe(
      tap(data => {
        // Cache the response
        this.cache.set(key, data);
      }),
    );
  }
}
```

In this example, the CacheInterceptor checks if the requested URL is already cached. If it is, it returns the cached response; otherwise, it proceeds with the request and caches the response for future use.

4. Using Interceptors for Logging

You can create an interceptor that logs incoming requests and outgoing responses.

Example: Logging Interceptor

```
// logging.interceptor.ts
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  CallHandler,
} from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const request = context.switchToHttp().getRequest();
    console.log(`Incoming request to: ${request.url}`);

    return next.handle().pipe(
      tap(data => {
        console.log(`Outgoing response: ${JSON.stringify(data)}`);
      }),
    );
  }
}
```

5. Handling Exceptions with Interceptors

Interceptors can also be used to handle exceptions thrown during the request processing.

Example: Exception Handling Interceptor

```
// exception.interceptor.ts
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
```

```
  CallHandler,
} from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
export class ExceptionInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next.handle().pipe(
      catchError(err => {
        // Custom error handling logic
        return throwError({
          statusCode: 500,
          message: 'An error occurred',
          error: err.message,
        });
      }),
    );
  }
}
```

6. Combining Multiple Interceptors

You can combine multiple interceptors on the same route or controller, and they will execute in the order they are applied.

```
@Controller('users')
@UseInterceptors(LoggingInterceptor, TransformInterceptor)
export class UsersController {
  @Get()
  getAllUsers() {
    return [{ id: 1, name: 'John Doe' }, { id: 2, name: 'Jane Doe' }];
  }
}
```

7. Using Interceptors with Observables

Interceptors work seamlessly with RxJS Observables, allowing you to perform additional operations on the data stream before sending the response.

11. Working with Databases (6-7 pages)

1. Setting Up a Database in NestJS

We'll go through the steps to set up a database connection using TypeORM as an example, as it is one of the most commonly used ORMs in NestJS projects.

Step 1: Install Dependencies

First, install TypeORM and the database driver for your database of choice. For example, if you are using PostgreSQL, you would run:

```
npm install @nestjs/typeorm typeorm pg
```

For other databases, replace pg with the appropriate driver:

- MySQL: mysql2
- SQLite: sqlite3
- MongoDB: mongodb (use Mongoose instead)

Step 2: Create a Database Module

Create a module for your database connection. You can name it database.module.ts.

```
// database.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity'; // Your entities here

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres', // Change to your database type
      host: 'localhost',
      port: 5432,
      username: 'your_username',
      password: 'your_password',
      database: 'your_database',
      entities: [User], // Add your entity classes here
      synchronize: true, // Set to false in production
    }),
  ],
})
export class DatabaseModule {}
```

Step 3: Import the Database Module

Import the DatabaseModule into your main application module (app.module.ts).

```
// app.module.ts
import { Module } from '@nestjs/common';
```

```
import { DatabaseModule } from './database.module';
import { UsersModule } from './users/users.module'; // Example user module

@Module({
 imports: [
  DatabaseModule,
  UsersModule,
 ],
})
export class AppModule {}
```

2. Creating Entities

Entities represent tables in your database. Here's how to create a User entity.

```
// entities/user.entity.ts
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class User {
 @PrimaryGeneratedColumn()
 id: number;

 @Column()
 name: string;

 @Column()
 email: string;

 @Column()
 password: string;
}
```

3. Creating a Service for Data Operations

Create a service that will handle CRUD operations for the User entity.

```
// users/users.service.ts
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from '../entities/user.entity';

@Injectable()
export class UsersService {
 constructor(
  @InjectRepository(User)
  private usersRepository: Repository<User>,
 ) {}

 create(user: Partial<User>): Promise<User> {
  const newUser = this.usersRepository.create(user);
  return this.usersRepository.save(newUser);
 }

 findAll(): Promise<User[]> {
  return this.usersRepository.find();
 }

 findOne(id: number): Promise<User> {
  return this.usersRepository.findOne(id);
 }

 async update(id: number, user: Partial<User>): Promise<User> {
  await this.usersRepository.update(id, user);
  return this.findOne(id);
 }

 async remove(id: number): Promise<void> {
  await this.usersRepository.delete(id);
 }
}
```

4. Creating a Controller

Create a controller to handle HTTP requests for the User entity.

```
// users/users.controller.ts
import { Controller, Get, Post, Body, Param, Put, Delete } from '@nestjs/common';
import { UsersService } from './users.service';
```

```
import { User } from '../entities/user.entity';

@Controller('users')
export class UsersController {
 constructor(private readonly usersService: UsersService) {}

 @Post()
 create(@Body() user: Partial<User>) {
  return this.usersService.create(user);
 }

 @Get()
 findAll() {
  return this.usersService.findAll();
 }

 @Get(':id')
 findOne(@Param('id') id: number) {
  return this.usersService.findOne(id);
 }

 @Put(':id')
 update(@Param('id') id: number, @Body() user: Partial<User>) {
  return this.usersService.update(id, user);
 }

 @Delete(':id')
 remove(@Param('id') id: number) {
  return this.usersService.remove(id);
 }
}
```

5. Setting Up the Users Module

Create a module for users to organize the service and controller.

```
// users/users.module.ts
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { User } from '../entities/user.entity';

@Module({
 imports: [TypeOrmModule.forFeature([User])],
 providers: [UsersService],
 controllers: [UsersController],
})
export class UsersModule {}
```

6. Running the Application

Now that everything is set up, run your application using:

```
npm run start
```

You can test the CRUD operations using tools like Postman or curl.

7. Advanced Database Operations

Querying with TypeORM

TypeORM allows you to perform complex queries using the repository methods. You can also use QueryBuilder for more advanced queries.

```
// Example of using QueryBuilder
async findUserByEmail(email: string): Promise<User> {
 return this.usersRepository.createQueryBuilder('user')
  .where('user.email = :email', { email })
  .getOne();
}
```

Using Migrations

Migrations help you keep your database schema in sync with your entities. You can generate migrations and run them using TypeORM CLI.

To create a migration:

```
npx typeorm migration:generate -n MigrationName
```

To run migrations:

```
npx typeorm migration:run
```

Using Transactions

You can use transactions in TypeORM to ensure that a set of operations either all succeed or all fail.

```
async createUserWithTransaction(userData: Partial<User>): Promise<User> {
```

```
  return await this.usersRepository.manager.transaction(async (entityManager) => {
    const user = entityManager.create(User, userData);
    return await entityManager.save(user);
  });
}
```

12. NestJS and GraphQL (5-6 pages)

1. Setting Up GraphQL in NestJS

Step 1: Install Required Packages

To get started with GraphQL in NestJS, install the necessary packages:

```
npm install @nestjs/graphql graphql-tools graphql apollo-server-express
```

Step 2: Create a GraphQL Module

Create a module to configure GraphQL. You can name it graphql.module.ts.

```
// graphql.module.ts
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo';
import { UsersModule } from './users/users.module'; // Example user module

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: true, // Automatically generate the schema file
    }),
    UsersModule,
  ],
})
export class GraphqlModule {}
```

Step 3: Import the GraphQL Module

Import the GraphqlModule into your main application module (app.module.ts).

```
// app.module.ts
import { Module } from '@nestjs/common';
import { GraphqlModule } from './graphql.module';

@Module({
  imports: [
    GraphqlModule,
    // other modules...
  ],
})
export class AppModule {}
```

2. Creating GraphQL Entities and Resolvers

Step 1: Define a GraphQL Schema

Using decorators provided by @nestjs/graphql, define your GraphQL entities. For instance, let's create a User type.

```
// users/user.model.ts
import { ObjectType, Field, Int } from '@nestjs/graphql';

@ObjectType()
export class User {
  @Field(() => Int)
  id: number;

  @Field()
  name: string;

  @Field()
  email: string;

  @Field()
  password: string;
}
```

Step 2: Create a Resolver

Create a resolver to handle GraphQL queries and mutations related to users.

```
// users/users.resolver.ts
import { Resolver, Query, Mutation, Args } from '@nestjs/graphql';
import { UsersService } from './users.service';
import { User } from './user.model';
import { CreateUserInput } from './create-user.input';

@Resolver(() => User)
export class UsersResolver {
```

```
  constructor(private readonly usersService: UsersService) {}

  @Query(() => [User])
  async users(): Promise<User[]> {
    return this.usersService.findAll();
  }

  @Mutation(() => User)
  async createUser(@Args('input') input: CreateUserInput): Promise<User> {
    return this.usersService.create(input);
  }
}
```

Step 3: Create Input Types

Input types are used for passing arguments into mutations. Create an input type for creating a user.

```
// users/create-user.input.ts
import { InputType, Field } from '@nestjs/graphql';

@InputType()
export class CreateUserInput {
  @Field()
  name: string;

  @Field()
  email: string;

  @Field()
  password: string;
}
```

3. Updating the Service

Make sure your service has the necessary methods for fetching and creating users.

```
// users/users.service.ts
import { Injectable } from '@nestjs/common';
import { User } from './user.model';
import { CreateUserInput } from './create-user.input';

@Injectable()
export class UsersService {
  private users: User[] = []; // Simulating a database

  create(input: CreateUserInput): User {
    const user = { id: this.users.length + 1, ...input };
    this.users.push(user);
    return user;
  }

  findAll(): User[] {
    return this.users;
  }
}
```

4. Updating the Module

Make sure to update the UsersModule to include the resolver and service.

```
// users/users.module.ts
import { Module } from '@nestjs/common';
import { UsersResolver } from './users.resolver';
import { UsersService } from './users.service';
import { TypeOrmModule } from '@nestjs/typeorm'; // Only if you are using TypeORM

@Module({
  providers: [UsersResolver, UsersService],
  exports: [UsersService], // Export service for use in other modules
})
export class UsersModule {}
```

5. Running the Application

Now that everything is set up, run your application:

6. Testing GraphQL API

You can test your GraphQL API using tools like GraphQL Playground or Apollo Studio.

Here are some example queries and mutations you can try:

Query to Fetch All Users

```
query {
  users {
    id
```

```
    name
    email
  }
}
```

Mutation to Create a User

```
mutation {
  createUser(input: { name: "John Doe", email: "john@example.com", password: "password" }) {
    id
    name
    email
  }
}
```

7. Using Middleware, Guards, and Interceptors

You can use middleware, guards, and interceptors with GraphQL just as you would with REST APIs. For example, you can create an authentication guard and apply it to your resolver methods.

Example: Applying a Guard

```
import { UseGuards } from '@nestjs/common';
import { AuthGuard } from './auth.guard';

@Resolver(() => User)
export class UsersResolver {
  constructor(private readonly usersService: UsersService) {}

  @Query(() => [User])
  @UseGuards(AuthGuard)
  async users(): Promise<User[]> {
    return this.usersService.findAll();
  }

  // Other methods...
}
```

8. Advanced Features

Pagination and Filtering

You can implement pagination and filtering in your GraphQL queries. Create input types for pagination and filtering and handle them in your resolvers.

Subscriptions

NestJS also supports GraphQL subscriptions for real-time functionalities. You can implement subscriptions by using the @Subscription decorator and managing WebSocket connections.

13. Building Microservices with NestJS (6-7 pages)

Building microservices with NestJS using RabbitMQ as the message broker allows you to create scalable and distributed systems that can communicate asynchronously. RabbitMQ is widely used for its reliability and flexibility in handling messaging between different services. Here's a step-by-step guide to set up a NestJS microservices architecture with RabbitMQ.

1. Setting Up a NestJS Microservice with RabbitMQ

Step 1: Install Required Packages

To use RabbitMQ with NestJS, you'll need to install the required dependencies. Run the following command:

```
npm install @nestjs/microservices amqplib
```

Step 2: Setting Up the RabbitMQ Service

Create a new module for your microservice. You can call it app.module.ts (or another appropriate name).

```
// app.module.ts
import { Module } from '@nestjs/common';
import { AppService } from './app.service';
import { AppController } from './app.controller';
import { ClientsModule, Transport } from '@nestjs/microservices';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: 'RABBITMQ_SERVICE', // Name of the microservice
        transport: Transport.RABBITMQ,
        options: {
          urls: ['amqp://localhost:5672'], // RabbitMQ server URL
          queue: 'my_queue', // Name of the queue
          queueOptions: {
            durable: false,
          },
        },
      },
    ]),
  ],
  controllers: [AppController],
```

```
  providers: [AppService],
})
export class AppModule {}
```

Step 3: Creating a Service to Handle Messages

Create a service that will handle messages sent to your RabbitMQ microservice.

```
// app.service.ts
import { Injectable } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';

@Injectable()
export class AppService {
  @MessagePattern('my_topic') // The topic to listen for messages
  getMessage(data: any): string {
    console.log('Received message:', data);
    return `Hello ${data.name}!`;
  }
}
```

Step 4: Creating a Controller to Send Messages

Create a controller to send messages to the RabbitMQ microservice.

```
// app.controller.ts
import { Controller, Post, Body } from '@nestjs/common';
import { ClientProxy, MessagePattern, RpcException } from '@nestjs/microservices';
import { Inject } from '@nestjs/common';

@Controller('message')
export class AppController {
  constructor(@Inject('RABBITMQ_SERVICE') private client: ClientProxy) {}

  @Post()
  sendMessage(@Body() message: { name: string }) {
    return this.client.send({ cmd: 'my_topic' }, message);
  }
}
```

2. Running the RabbitMQ Server

Make sure you have RabbitMQ installed and running. You can use Docker to quickly set up RabbitMQ:

```
docker run -d --hostname my-rabbit --name some-rabbit -p 5672:5672 -p 15672:15672 rabbitmq:management
```

You can access the RabbitMQ management interface at http://localhost:15672/ (default username and password are both guest).

3. Running the Application

Now, run your NestJS application:

```
npm run start
```

4. Sending a Test Message

You can use a tool like Postman to test your setup. Send a POST request to http://localhost:3000/message with a JSON body:

```
{
  "name": "John"
}
```

You should see the log output in your service indicating that the message was received:

```
Received message: { name: 'John' }
```

5. Setting Up Multiple Microservices

You can set up multiple microservices, each with its own message patterns and queues. Simply create additional modules, services, and controllers, configuring them similarly to the steps above.

Example of a Second Microservice

Create another microservice, let's say a logging service.

1. **Create a new service and controller for logging.**

```
// logging.service.ts
import { Injectable } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';

@Injectable()
export class LoggingService {
  @MessagePattern('log_event')
  logEvent(data: any): void {
    console.log('Logging event:', data);
  }
}
```

2. **Add a controller for sending log messages.**

```
// logging.controller.ts
import { Controller, Post, Body } from '@nestjs/common';
import { ClientProxy, Inject } from '@nestjs/microservices';

@Controller('log')
```

```
export class LoggingController {
  constructor(@Inject('RABBITMQ_SERVICE') private client: ClientProxy) {}

  @Post()
  log(@Body() message: { event: string }) {
    this.client.emit('log_event', message);
  }
}
```

6. Handling Errors and Retries

You can implement error handling and message retries in your microservice to handle cases where messages fail to process.

Example: Handling Errors

Use the RpcException class to handle errors in your services.

```
// app.service.ts
import { RpcException } from '@nestjs/microservices';

@MessagePattern('my_topic')
getMessage(data: any): string {
  if (!data.name) {
    throw new RpcException('Name is required');
  }
  return `Hello ${data.name}!`;
}
```

7. Implementing Middleware and Guards

You can also implement middleware and guards in your microservices to handle cross-cutting concerns, such as authentication and logging.

8. Deploying Microservices

When deploying your microservices, consider using Docker and Kubernetes to manage your services effectively. This will allow you to scale your services and manage container orchestration.

- **Transport Layers**: Redis, NATS, RabbitMQ, gRPC

Feature	Redis	NATS	RabbitMQ	gRPC
Type	In-memory data structure store	Lightweight messaging system	Message broker	RPC framework
Protocol	Redis protocol	NATS protocol	AMQP	HTTP/2
Message Pattern	Pub/Sub, Streams, Queues	Pub/Sub, Request/Reply	Queues, Pub/Sub, RPC	Unary, Streaming, Bidirectional
Performance	Very high throughput (in-memory)	High performance, low latency	Moderate latency, reliable	High performance (due to HTTP/2)
Persistence	Yes (with persistence options)	No (volatile by default)	Yes (durable queues)	No (stateless)
Delivery Guarantee	At-most-once	At-most-once	At-least-once, exactly-once	Exactly-once (if implemented)
Complexity	Simple to set up	Very simple and lightweight	More complex (configuration needed)	Simple to use with auto-generated code
Scalability	Vertical and horizontal scaling	Horizontal scaling	Horizontal scaling	Horizontal scaling
Use Cases	Caching, real-time analytics, session store	Microservices, IoT, cloud-native apps	Task queues, event-driven apps	Microservices, inter-service communication
Ecosystem	Rich ecosystem (Redis modules)	Lightweight, minimal dependencies	Rich ecosystem (plugins)	Strong integration with various languages
Client Libraries	Extensive support across languages	Lightweight client libraries	Extensive support across languages	Extensive support across languages
Transaction Support	Limited (MULTI/EXEC)	No	Yes	No (but can be implemented in application)
Message Ordering	No inherent guarantee	No inherent guarantee	Yes (depending on queue setup)	Yes (for streaming)

- **Communication Patterns**: Request-response, event-based

When designing microservices architectures, it's crucial to understand the various communication patterns that can be employed. These patterns determine how microservices interact with each other, affecting performance, reliability, and scalability. Here's an overview of the primary communication patterns used in microservices:

1. Synchronous Communication Patterns

Synchronous communication involves direct communication between microservices, where the caller waits for a response before proceeding.

1.1. HTTP/REST API

- **Description:** Microservices expose RESTful APIs over HTTP. Clients make requests to these APIs and wait for responses.
- **Use Cases:** Common for web services, mobile applications, and internal service-to-service communication.
- **Pros:** Simple to implement, widely understood, and supported by many tools.

- **Cons:** Can lead to tight coupling between services; potential latency issues if services are slow.

1.2. gRPC

- **Description:** A high-performance RPC framework that uses HTTP/2 for transport and Protocol Buffers for serialization.
- **Use Cases:** Ideal for high-throughput microservices and internal service communications where performance is critical.
- **Pros:** Supports streaming, efficient binary serialization, and built-in authentication.
- **Cons:** More complex setup, and requires knowledge of Protocol Buffers.

2. Asynchronous Communication Patterns

Asynchronous communication allows microservices to send messages without waiting for a response, promoting decoupling and improving scalability.

2.1. Message Queues

- **Description:** Microservices communicate through message brokers (like RabbitMQ, Kafka, or NATS). A producer sends messages to a queue, and a consumer processes them asynchronously.
- **Use Cases:** Background jobs, event-driven architectures, and scenarios requiring decoupling.
- **Pros:** Increases resilience and scalability; supports load balancing; can handle bursts of traffic.
- **Cons:** Message delivery can be complex; requires management of message queues.

2.2. Event Streams

- **Description:** Services publish events to an event stream, and other services subscribe to those events. This pattern is often implemented using systems like Apache Kafka or AWS Kinesis.
- **Use Cases:** Event-driven systems, real-time analytics, and tracking changes in state.
- **Pros:** Provides a complete history of events, decouples services, and supports replaying events.
- **Cons:** Complexity in managing event schema and data consistency.

2.3. Webhooks

- **Description:** A service can register a URL with another service, which then calls this URL with a payload when an event occurs.
- **Use Cases:** Integrating with third-party services, notifications, and callbacks.
- **Pros:** Lightweight and easy to implement; decouples the event producer from the consumer.
- **Cons:** Limited reliability (e.g., if the consumer is down) and possible security issues.

3. Hybrid Communication Patterns

Combining synchronous and asynchronous communication can leverage the strengths of both approaches.

3.1. Request/Reply

- **Description:** A service sends a request to another service and expects a response. This can be done using message brokers, where the response is sent back through a different channel.
- **Use Cases:** When you need to maintain a request-response pattern while benefiting from the advantages of message queues.
- **Pros:** Combines decoupling with synchronous behavior.
- **Cons:** Can introduce complexity in message handling and increase latency.

4. Communication Patterns Overview Table

Pattern	Type	Description	Use Cases	Pros	Cons
HTTP/REST API	Synchronous	Direct API calls over HTTP	Web services, internal services	Simple to use and implement	Tight coupling, potential latency
gRPC	Synchronous	High-performance RPC using HTTP/2	High-throughput services	Efficient serialization, streaming	More complex setup
Message Queues	Asynchronous	Communication through message brokers	Background jobs, decoupling	Resilient and scalable	Delivery complexity
Event Streams	Asynchronous	Publishing and subscribing to event streams	Event-driven systems	Historical event tracking	Schema management complexity
Webhooks	Asynchronous	Callbacks via HTTP when events occur	Third-party integrations	Lightweight	Reliability issues
Request/Reply	Hybrid	Request/response using message brokers	Combining both approaches	Decoupled and synchronous	Increased latency and complexity

14. Advanced Topics (6-7 pages)

- **CQRS Pattern**: Using the Command Query Responsibility Segregation
- **Event-Driven Architecture**: Integrating with event emitters
- **Testing with NestJS**:

1. Setting Up Testing with Jest

NestJS uses Jest as its default testing framework. If you created your NestJS project using the Nest CLI, Jest should already be set up. If not, you can install it by running:

```
npm install --save-dev jest @nestjs/testing ts-jest @types/jest
```

You can create a jest.config.js file in the root of your project to customize your Jest configuration:

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  testPathIgnorePatterns: ['/node_modules/', '/dist/'],
};
```

2. Unit Testing with Jest

Unit testing focuses on testing individual components or services in isolation.

Step 1: Create a Simple Service
Let's say you have a simple service called UsersService that creates a user:

```
// src/users/users.service.ts
import { Injectable } from '@nestjs/common';

@Injectable()
export class UsersService {
  private users = [];

  createUser(name: string, email: string) {
    const user = { name, email };
    this.users.push(user);
    return user;
  }

  getUsers() {
    return this.users;
  }
}
```

Step 2: Write Unit Tests for the Service
Create a test file for UsersService:

```
touch src/users/users.service.spec.ts
```

Edit users.service.spec.ts:

```
// src/users/users.service.spec.ts
import { Test, TestingModule } from '@nestjs/testing';
import { UsersService } from './users.service';

describe('UsersService', () => {
  let service: UsersService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [UsersService],
    }).compile();

    service = module.get<UsersService>(UsersService);
  });

  it('should be defined', () => {
    expect(service).toBeDefined();
  });

  it('should create a user', () => {
    const user = service.createUser('Jane Doe', 'jane@example.com');
    expect(user).toEqual({ name: 'Jane Doe', email: 'jane@example.com' });
    expect(service.getUsers()).toContainEqual(user);
  });

  it('should return all users', () => {
    service.createUser('Jane Doe', 'jane@example.com');
    service.createUser('John Doe', 'john@example.com');
    expect(service.getUsers()).toHaveLength(2);
  });
});
```

Step 3: Run the Unit Tests
Run your tests using the following command:

```
npm run test
```

3. Integration Testing
Integration tests focus on the interaction between components or services. In NestJS, you can use the @nestjs/testing module to set up your testing environment.
Step 1: Write Integration Tests for a Controller
Let's say you have a UsersController that uses UsersService. Create a test file for it:

```
touch src/users/users.controller.spec.ts
```

Edit users.controller.spec.ts:

```
// src/users/users.controller.spec.ts
import { Test, TestingModule } from '@nestjs/testing';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

describe('UsersController', () => {
  let controller: UsersController;
```

```
  let service: UsersService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      controllers: [UsersController],
      providers: [
        {
          provide: UsersService,
          useValue: {
            createUser: jest.fn(),
            getUsers: jest.fn(),
          },
        },
      ],
    }).compile();

    controller = module.get<UsersController>(UsersController);
    service = module.get<UsersService>(UsersService);
  });

  it('should create a user', async () => {
    const createUserDto = { name: 'Jane Doe', email: 'jane@example.com' };
    const user = { ...createUserDto };

    jest.spyOn(service, 'createUser').mockImplementation(async () => user);

    expect(await controller.createUser(createUserDto)).toEqual(user);
    expect(service.createUser).toHaveBeenCalledWith(createUserDto.name, createUserDto.email);
  });

  it('should return all users', async () => {
    const users = [{ name: 'Jane Doe', email: 'jane@example.com' }];
    jest.spyOn(service, 'getUsers').mockImplementation(async () => users);

    expect(await controller.getUsers()).toEqual(users);
    expect(service.getUsers).toHaveBeenCalled();
  });
});
```

Step 2: Run the Integration Tests

Run the tests as before:

```
npm run test
```

4. End-to-End Testing

End-to-end (E2E) tests check the complete flow of your application, including the HTTP layer. NestJS provides a built-in way to set up E2E tests.

Step 1: Create an E2E Test File

In your test directory, create an E2E test file:

```
mkdir test
touch test/app.e2e-spec.ts
```

Edit app.e2e-spec.ts:

```
// test/app.e2e-spec.ts
import { Test, TestingModule } from '@nestjs/testing';
import { INestApplication } from '@nestjs/common';
import * as request from 'supertest';
import { AppModule } from '../src/app.module';

describe('Users (e2e)', () => {
  let app: INestApplication;

  beforeAll(async () => {
    const moduleFixture: TestingModule = await Test.createTestingModule({
      imports: [AppModule],
    }).compile();

    app = moduleFixture.createNestApplication();
    await app.init();
  });

  it('/users (POST)', () => {
    return request(app.getHttpServer())
      .post('/users')
      .send({ name: 'Jane Doe', email: 'jane@example.com' })
```

```
    .expect(201);
  });

  it('/users (GET)', () => {
    return request(app.getHttpServer())
      .get('/users')
      .expect(200)
      .expect((res) => {
        expect(res.body).toBeInstanceOf(Array);
      });
  });

  afterAll(async () => {
    await app.close();
  });
});
```

Step 2: Run the E2E Tests

Run the E2E tests with the following command:

```
npm run test:e2e
```

5. Mocking Dependencies

When writing tests, you may want to mock certain dependencies to isolate the unit being tested. You can use jest.mock() or manual mocks.

6. Best Practices for Testing in NestJS

- **Keep Tests Isolated:** Ensure tests don't depend on the state of other tests.
- **Use Mock Services:** For unit tests, mock external services to avoid hitting real APIs or databases.
- **Test Edge Cases:** Always consider edge cases and failure scenarios in your tests.
- **Run Tests Regularly:** Integrate tests into your CI/CD pipeline to ensure code quality.

15. Security and Authentication (5-6 pages)

- **Authentication Strategies**:

Authentication is a fundamental aspect of any web application, ensuring that only authorized users can access certain resources. NestJS provides a flexible way to implement various authentication strategies using modules like Passport.js. Below is a guide on how to set up authentication strategies in a NestJS application.

1. Setting Up NestJS with Authentication

If you haven't already created a NestJS project, you can do so with the following commands:

```
npm i -g @nestjs/cli
nest new auth-example
```

2. Install Required Packages

You need to install @nestjs/passport, passport, and any specific strategy you want to implement (e.g., passport-local for username/password authentication or passport-jwt for JWT authentication):

```
npm install @nestjs/passport passport passport-local passport-jwt jsonwebtoken bcrypt
```

3. Create an Authentication Module

You can create a dedicated module for authentication:

```
nest generate module auth
```

4. Implementing Local Authentication

Step 1: Create Local Strategy

Create a local strategy to handle username and password authentication:

```
nest generate service auth/auth.service.ts
nest generate guard auth/local-auth.guard.ts
```

Edit auth.service.ts:

```
// src/auth/auth.service.ts
import { Injectable } from '@nestjs/common';
import { User } from '../users/user.entity'; // Assuming you have a User entity
import * as bcrypt from 'bcrypt';

@Injectable()
export class AuthService {
  private users: User[] = []; // Replace with a real user store

  async validateUser(username: string, password: string): Promise<any> {
    const user = this.users.find(user => user.username === username);
    if (user && (await bcrypt.compare(password, user.password))) {
      return user;
    }
    return null;
  }
}
```

Step 2: Create Local Strategy Class

Create the local strategy:

```
// src/auth/local.strategy.ts
import { Strategy } from 'passport-local';
```

```
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import { AuthService } from './auth.service';

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super(); // Call the parent class constructor
  }

  async validate(username: string, password: string): Promise<any> {
    return this.authService.validateUser(username, password);
  }
}
```

Step 3: Implement Local Auth Guard

Edit local-auth.guard.ts:

```
// src/auth/local-auth.guard.ts
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

Step 4: Create Auth Controller

Create an authentication controller:

```
nest generate controller auth/auth.controller.ts
```

Edit auth.controller.ts:

```
// src/auth/auth.controller.ts
import { Controller, Post, UseGuards, Request } from '@nestjs/common';
import { AuthService } from './auth.service';
import { LocalAuthGuard } from './local-auth.guard';

@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @UseGuards(LocalAuthGuard)
  @Post('login')
  async login(@Request() req) {
    return req.user; // Handle your login response here (e.g., JWT)
  }
}
```

Step 5: Register the Module

Update the auth.module.ts to include the new strategy and services:

```
// src/auth/auth.module.ts
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { LocalStrategy } from './local.strategy';
import { PassportModule } from '@nestjs/passport';

@Module({
  imports: [PassportModule],
  providers: [AuthService, LocalStrategy],
  controllers: [AuthController],
})
export class AuthModule {}
```

5. Implementing JWT Authentication

To add JWT authentication, you can follow these steps:

Step 1: Install Additional Packages

If you haven't already, install the necessary packages for JWT:

```
npm install @nestjs/jwt
```

Step 2: Update Auth Service

Edit auth.service.ts to include JWT logic:

```
// src/auth/auth.service.ts
import { Injectable } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { User } from '../users/user.entity'; // Assuming you have a User entity
import * as bcrypt from 'bcrypt';

@Injectable()
```

```
export class AuthService {
  private users: User[] = []; // Replace with a real user store

  constructor(private readonly jwtService: JwtService) {}

  async validateUser(username: string, password: string): Promise<any> {
    const user = this.users.find(user => user.username === username);
    if (user && (await bcrypt.compare(password, user.password))) {
      return user;
    }
    return null;
  }

  async login(user: any) {
    const payload = { username: user.username, sub: user.userId };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}
```

Step 3: Create JWT Strategy

Create a JWT strategy class:

```
// src/auth/jwt.strategy.ts
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { AuthService } from './auth.service';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: 'yourSecretKey', // Use environment variables for sensitive data
    });
  }

  async validate(payload: any) {
    return { userId: payload.sub, username: payload.username };
  }
}
```

Step 4: Register JWT Module

Update the auth.module.ts to include the JWT module and strategy:

```
// src/auth/auth.module.ts
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { LocalStrategy } from './local.strategy';
import { JwtStrategy } from './jwt.strategy';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';

@Module({
  imports: [
    PassportModule,
    JwtModule.register({
      secret: 'yourSecretKey', // Use environment variables for sensitive data
      signOptions: { expiresIn: '60s' }, // Token expiration time
    }),
  ],
  providers: [AuthService, LocalStrategy, JwtStrategy],
  controllers: [AuthController],
})
export class AuthModule {}
```

Step 5: Update Auth Controller for JWT Login

Edit auth.controller.ts to return the JWT token:

```
// src/auth/auth.controller.ts
import { Controller, Post, UseGuards, Request } from '@nestjs/common';
import { AuthService } from './auth.service';
import { LocalAuthGuard } from './local-auth.guard';
```

```
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @UseGuards(LocalAuthGuard)
  @Post('login')
  async login(@Request() req) {
    return this.authService.login(req.user); // Returns JWT
  }
}
```

6. Securing Routes with JWT Guard

You can create a JWT guard to protect your routes:

```
// src/auth/jwt-auth.guard.ts
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

Using the Guard

You can use the guard in your controllers to protect certain routes:

```
// src/users/users.controller.ts
import { Controller, Get, UseGuards } from '@nestjs/common';
import { JwtAuthGuard } from '../auth/jwt-auth.guard';

@Controller('users')
export class UsersController {
  @UseGuards(JwtAuthGuard)
  @Get()
  getUsers() {
    // Logic to get users
  }
}
```

- **Using Guards for Authentication**: Implementing and protecting routes

Example: JWT-based authentication

```
@UseGuards(AuthGuard('jwt'))
@Get('profile')
getProfile(@Request() req) {
  return req.user;
}
```

Oauth2 nestjs

Implementing OAuth2 in a NestJS application involves a few key steps. You typically use libraries like passport, passport-oauth2, or @nestjs/passport along with a strategy for handling OAuth2 flows. Here's a basic outline of how you can set up OAuth2 in your NestJS application:

Step 1: Install Required Packages

You need to install the following packages:

npm install @nestjs/passport passport passport-oauth2

Step 2: Create an OAuth2 Strategy

You can create a custom OAuth2 strategy by extending the PassportStrategy class. Here's a simple example:

```
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { Strategy, VerifyCallback } from 'passport-oauth2';

@Injectable()
export class OAuth2Strategy extends PassportStrategy(Strategy, 'oauth2') {
  constructor() {
    super({
      authorizationURL: 'https://provider.com/oauth2/auth',
      tokenURL: 'https://provider.com/oauth2/token',
      clientID: 'your-client-id',
      clientSecret: 'your-client-secret',
      callbackURL: 'http://localhost:3000/auth/callback',
    });
  }

  async validate(accessToken: string, refreshToken: string, profile: any, done: VerifyCallback): Promise<void> {
    const user = { accessToken, profile }; // customize user object as needed
    done(null, user);
  }
}
```

Step 3: Set Up Authentication Module

Create an authentication module to handle the configuration:

```
import { Module } from '@nestjs/common';
import { PassportModule } from '@nestjs/passport';
import { OAuth2Strategy } from './oauth2.strategy';
import { AuthService } from './auth.service';

@Module({
  imports: [PassportModule.register({ defaultStrategy: 'oauth2' })],
  providers: [OAuth2Strategy, AuthService],
  exports: [AuthService],
})
export class AuthModule {}
```

Step 4: Create Auth Controller

Next, you need to create a controller that will handle the authentication routes:

```
import { Controller, Get, UseGuards, Req, Res } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';
import { Request, Response } from 'express';

@Controller('auth')
export class AuthController {
  @Get('login')
  @UseGuards(AuthGuard('oauth2'))
  login() {
    // Initiates the OAuth2 login flow
  }

  @Get('callback')
  @UseGuards(AuthGuard('oauth2'))
  callback(@Req() req: Request, @Res() res: Response) {
    // Successful authentication
    const user = req.user;
    res.json(user); // Or redirect to your frontend
  }
}
```

Step 5: Set Up Your Main Module

Make sure to import the AuthModule in your main application module:

```
import { Module } from '@nestjs/common';
import { AuthModule } from './auth/auth.module';
import { AuthController } from './auth/auth.controller';

@Module({
  imports: [AuthModule],
  controllers: [AuthController],
})
export class AppModule {}
```

Step 6: Configure the OAuth Provider

Make sure you configure your OAuth provider (like Google, GitHub, etc.) to use the same clientID, clientSecret, and callback URL.

Additional Considerations

- **Security**: Ensure you handle tokens securely and consider the implications of storing user information.
- **Session Management**: You may need to implement session management, possibly using a library like express-session.
- **Error Handling**: Implement proper error handling for your authentication flows.

16. NestJS Deployment and Best Practices (5-6 pages)

- **Deployment Strategies**: Dockerizing, deploying to AWS, Heroku

Dockerizing and deploying a NestJS application to AWS can be broken down into several key steps. Below is a comprehensive guide to help you through the process.

Step 1: Dockerize Your NestJS Application

1. **Create a Dockerfile**: In the root of your NestJS project, create a Dockerfile:

```
# Use the official Node.js image as a base image
FROM node:18-alpine

# Set the working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install --production

# Copy the rest of the application code
```

```
COPY . .

# Build the NestJS application
RUN npm run build

# Expose the application port (default is 3000)
EXPOSE 3000

# Command to run the application
CMD ["node", "dist/main"]
```

2. **Create a .dockerignore File**: This file will exclude unnecessary files from being included in the Docker image.

```
node_modules
dist
npm-debug.log
.env
```

3. **Build the Docker Image**: Run the following command in your terminal:

```
docker build -t my-nest-app .
```

Step 2: Run the Docker Container Locally

To test if everything works correctly, you can run your Docker container locally:

```
docker run -p 3000:3000 my-nest-app
```

You should be able to access your application at http://localhost:3000.

Step 3: Push the Docker Image to a Container Registry

You can use AWS Elastic Container Registry (ECR) to store your Docker images.

1. **Login to ECR**:

```
aws ecr get-login-password --region your-region | docker login --username AWS --password-stdin your-account-id.dkr.ecr.your-region.amazonaws.com
```

2. **Create an ECR Repository**:

```
aws ecr create-repository --repository-name my-nest-app
```

3. **Tag the Docker Image**:

```
docker tag my-nest-app:latest your-account-id.dkr.ecr.your-region.amazonaws.com/my-nest-app:latest
```

4. **Push the Image to ECR**:

```
docker push your-account-id.dkr.ecr.your-region.amazonaws.com/my-nest-app:latest
```

Step 4: Deploying to AWS Elastic Beanstalk

1. **Install the AWS Elastic Beanstalk CLI**: If you haven't already, install the EB CLI:

```
pip install awsebcli
```

2. **Initialize Your Elastic Beanstalk Application**:

```
eb init -p docker my-nest-app
```

Follow the prompts to configure your application.

3. **Create an Environment and Deploy**:

```
eb create my-nest-app-env
```

This will create an environment and deploy your Docker container.

4. **Open Your Application**:

Once the deployment is complete, you can open your application with:

```
eb open
```

Step 5: Configure Environment Variables (if needed)

You can configure environment variables through the Elastic Beanstalk console or by using the EB CLI:

```
eb setenv VAR_NAME=value
```

Step 6: Monitor and Manage Your Application

You can monitor the logs and manage your application using:

```
eb logs
```

Additional Considerations

- **Scaling and Load Balancing**: AWS Elastic Beanstalk automatically manages scaling and load balancing. You can configure these settings in the console.
- **Database Integration**: If your application requires a database, consider using AWS RDS and configure your application to connect to it.
- **CI/CD**: You might want to set up Continuous Integration/Continuous Deployment pipelines using AWS CodePipeline and CodeBuild for automating your deployments.
- **Security**: Ensure to configure your security groups and IAM roles to limit access to your application and resources.

Performance Optimization:

Performance optimization in a NestJS application involves various techniques and best practices that can enhance the speed, efficiency, and scalability of your application. Here's a comprehensive list of strategies you can employ:

1. Optimize Database Queries

- **Use Efficient Queries**: Avoid N+1 query problems by using eager loading or batch fetching.
- **Indexing**: Ensure that database fields that are frequently queried are properly indexed.
- **Connection Pooling**: Use connection pooling to manage database connections efficiently.
- **Caching**: Implement caching strategies for frequently accessed data using tools like Redis or in-memory caching.

2. Enable Compression

- **Gzip Compression**: Enable Gzip compression for responses to reduce the size of the data sent over the network.

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as compression from 'compression';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(compression());
  await app.listen(3000);
}
bootstrap();
```

3. Use Asynchronous Programming

- **Async/Await**: Use async/await for handling asynchronous operations, which can improve code readability and performance.
- **Promise.all**: Use Promise.all() to run multiple promises in parallel when possible.

4. Implement Caching

- **HTTP Caching**: Use caching strategies for HTTP responses using libraries like @nestjs/cache-manager.

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

@Module({
  imports: [CacheModule.register()],
})
export class AppModule {}
```

- **Redis Caching**: Consider using Redis for caching data that doesn't change frequently.

5. Optimize Middleware and Interceptors

- **Avoid Heavy Middleware**: Ensure middleware is lightweight. If possible, avoid processing requests that don't require it.
- **Use Interceptors**: Leverage interceptors to modify or log requests/responses without affecting core logic.

6. Use Proper Error Handling

- **Global Error Filters**: Use global error filters to catch and handle errors efficiently.

```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from '@nestjs/common';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    // Handle error
  }
}
```

7. Optimize NestJS Configuration

- **Enable Fastify**: Consider using Fastify instead of Express as the underlying HTTP adapter for better performance.

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { FastifyAdapter } from '@nestjs/platform-fastify';

async function bootstrap() {
  const app = await NestFactory.create(AppModule, new FastifyAdapter());
  await app.listen(3000);
}
bootstrap();
```

- **Use the Latest Node.js Version**: Ensure you're using a supported and optimized version of Node.js.

8. Optimize Static Assets

- **Serve Static Files Efficiently**: Use a CDN (Content Delivery Network) for serving static assets.
- **Optimize Images**: Compress images before deployment to reduce load times.

9. Analyze Performance

- **Profiling**: Use profiling tools (like Node.js built-in profiling, clinic.js, or node --inspect) to identify bottlenecks.
- **Logging and Monitoring**: Implement logging (e.g., using winston or pino) and monitoring (e.g., using Prometheus and Grafana) to track application performance.

10. Scaling

- **Horizontal Scaling**: Scale your application horizontally by running multiple instances and using a load balancer.
- **Kubernetes**: If using Kubernetes, utilize Horizontal Pod Autoscaler to manage scaling based on load.

11. Optimize API Responses

- **Pagination**: Implement pagination for large datasets to reduce response sizes.
- **Data Transformation**: Use DTOs (Data Transfer Objects) and validation pipes to ensure only necessary data is sent to clients.

12. Use Content Delivery Networks (CDNs)

- **CDN for Static Assets**: Serve static files through a CDN to improve load times for users across different geographical locations.

Best Practices:

When developing applications with NestJS, following best practices can help you build maintainable, scalable, and efficient applications. Here's a comprehensive list of best practices to consider:

1. Project Structure

- **Modular Architecture**: Organize your application into modules. Each module should encapsulate related functionality. Use feature-based modules for better maintainability.

```
src/
├── app.module.ts
├── auth/
│   ├── auth.module.ts
│   ├── auth.controller.ts
│   └── auth.service.ts
├── users/
│   ├── users.module.ts
│   ├── users.controller.ts
│   └── users.service.ts
└── common/
    ├── decorators/
    ├── filters/
    └── pipes/
```

2. Use TypeScript Features

- **Leverage Type Safety**: Use TypeScript's features like interfaces and types to enforce contracts and ensure type safety throughout your application.
- **DTOs (Data Transfer Objects)**: Use DTOs to validate and transform incoming data. This improves security and maintains data integrity.

```
import { IsString, IsEmail } from 'class-validator';

export class CreateUserDto {
  @IsString()
  name: string;

  @IsEmail()
  email: string;
}
```

3. Dependency Injection

- **Utilize Dependency Injection**: Take full advantage of NestJS's powerful dependency injection system to manage service dependencies, making your code more testable and modular.

4. Middleware, Guards, Interceptors, and Pipes

- **Use Middleware**: Implement middleware for cross-cutting concerns like logging, authentication, and body parsing.
- **Guards**: Use guards to handle authorization and role-based access control.
- **Interceptors**: Use interceptors for modifying requests/responses, logging, and transforming results.
- **Pipes**: Use pipes for input validation and transformation.

5. Exception Handling

- **Global Exception Filters**: Implement global exception filters to handle errors uniformly across your application.

```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from '@nestjs/common';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    // Handle error response
  }
}
```

6. Configuration Management

- **Use Configuration Module**: Utilize the @nestjs/config package to manage application configurations. This helps in keeping sensitive data secure and configurable.

```
import { ConfigModule } from '@nestjs/config';

@Module({
  imports: [ConfigModule.forRoot()],
})
export class AppModule {}
```

7. Testing

- **Write Unit and Integration Tests**: Ensure you write tests for your services, controllers, and modules. Use Jest, which comes pre-configured with NestJS.

```
import { Test, TestingModule } from '@nestjs/testing';
import { UsersService } from './users.service';

describe('UsersService', () => {
  let service: UsersService;
```

```
  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [UsersService],
    }).compile();

    service = module.get<UsersService>(UsersService);
  });

  it('should be defined', () => {
    expect(service).toBeDefined();
  });
});
```

8. Logging

- **Use a Logger**: Implement a logging mechanism to capture application events and errors. NestJS provides a built-in logger, or you can use libraries like winston or pino.

9. Security Practices

- **Sanitize Inputs**: Always validate and sanitize inputs to protect against injection attacks.
- **Use HTTPS**: Ensure your application is served over HTTPS in production.
- **Rate Limiting**: Implement rate limiting to protect your APIs from abuse.

10. Documentation

- **API Documentation**: Use tools like Swagger to document your APIs. NestJS provides the @nestjs/swagger module to help generate OpenAPI documentation.

```
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';

const options = new DocumentBuilder()
  .setTitle('API Documentation')
  .setDescription('API description')
  .setVersion('1.0')
  .build();
const document = SwaggerModule.createDocument(app, options);
SwaggerModule.setup('api', app, document);
```

11. Performance Optimization

- **Use Caching**: Implement caching for frequently accessed data and responses.
- **Optimize Database Queries**: Use efficient queries and consider indexing to improve performance.

12. Environment Management

- **Use Environment Variables**: Store sensitive data like API keys and database connection strings in environment variables.

13. Version Control

- **Version Your APIs**: Use versioning for your APIs to ensure backward compatibility when making changes.

14. Health Checks and Monitoring

- **Implement Health Checks**: Add health checks to monitor the status of your application.
- **Monitoring**: Use monitoring tools (like Prometheus and Grafana) to track application performance and logs.

Interview Questions

1. **What's the difference between NestJS and Angular?**
   Angular is a framework for building client-side applications and It provides a way to organize your frontend code using components, modules, services, etc.
   while NestJS is a framework for building server-side applications. NestJS is built on top of TypeScript and Express, and it aims to provide a more robust and scalable architecture for enterprise-level applications. However It's heavily inspired by Angular and shares similar concepts like modules, decorators, and dependency injection.
2. **Is it possible to use other languages like C++, Ruby or Python with NestJS? If yes, then how?**
   Yes, it is possible to use other languages with NestJS. NestJS is language agnostic, meaning that it can work with any language that can compile to JavaScript.
   As for Python, Ruby, or other languages, they can't be used directly with NestJS because NestJS relies on the Node.js runtime, which executes JavaScript. As for Python, Ruby, or other languages, they can't be used directly with NestJS because NestJS relies on the Node.js runtime, which executes JavaScript.
   However, you can certainly build separate services in Python, Ruby, or any other language, and have them communicate with your NestJS application via HTTP, gRPC, or any other communication protocol. This is a common pattern in microservices architecture.
3. **Can you explain how to use decorators in a NestJS controller?**
   Before explaining how to use decorators, let me explain more about what are decorators:
   decorators are special functions that are prefixed with an @ symbol and can be attached to classes, methods, or properties. They are used to add metadata, methods, properties, or observe the behavior of the classes, methods, or properties they are attached to.
   NestJS provides several built-in decorators, and you can also create custom decorators. Here are some examples of built-in decorators in NestJS:
   i. Class decorators like @Controller(), @Module(), @Injectable(), etc. These are used to annotate classes.
   ii. Method decorators like @Get(), @Post(), @Put(), etc. These are used to annotate methods within controller classes to handle specific routes.
   iii. Parameter decorators like @Req(), @Res(), @Body(), etc. These are used to annotate parameters within route handling methods.
   iv. Property decorators like @Inject(). These are used to annotate properties within classes.
   v. Custom decorators. You can create your own decorators to handle common tasks across your application.
   For example, below is how method decorators are used to handle GET, POST, PUT, DELETE requests respectively.

```
import {
  Controller,
  Get,
  Param,
  Body,
  Post,
  Patch,
  Delete,
} from "@nestjs/common";

@Controller("cats")
export class CatsController {
  @Get()
  findAll(): string {
    return "This action returns all cats";
  }

  @Get(":id")
  findOne(@Param("id") id: number): string {
    return `This action returns a cat with the provided id`;
  }

  @Post()
  create(@Body() body: any): string {
    return `This action returns the body of the cat`;
  }

  @Patch("id")
  update(@Param("id") id: number, @Body() body: any): string {
    return `This action updates the body of the cat`;
  }

  @Delete("id")
  remove(@Param("id") id: number): string {
    return `This action removes a cat`;
  }
}
```

4. **Explain the concept of Dependency Injection in NestJS. How does it help in building modular and testable applications?**
   Dependency Injection (DI) is a design pattern in which a class receives its dependencies from external sources rather than creating them itself. This pattern is fundamental to the way NestJS is designed. dependency injection involves

letting the framework manage the creation and injection of dependencies into the components (controllers, services, and more) as needed.
This is achieved through decorators, providers, and the NestJS IoC (Inversion of Control) container Here is an example:

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

@Injectable()
export class AppService {
 getHello(): string {
  return "Hello World!";
 }
}
 import { Controller, Get } from '@nestjs/common';
 import { AppService } from './app.service';

 @Controller()
 export class AppController {
  constructor(private appService: AppService) {}

    @Get()
     getHello(): string {
      return this.appService.getHello();
    }
 }
```

In this example, AppService is a provider that is injected into AppController through the constructor. When AppController is created, NestJS automatically creates an instance of AppService and passes it to the constructor.
DI enables us build modular and testable applications in several ways:
Modularity: By injecting dependencies, you can easily swap out one implementation for another. This is useful when you want to change the behavior of parts of your application without changing the classes that use them.
Testability: DI makes it easy to unit test your classes, as you can inject mock versions of dependencies for testing purposes.
Separation of Concerns: Each class focuses on its own behavior, delegating the behavior of its dependencies to the classes that implement those dependencies. This leads to cleaner, more readable code

5. **What's the difference between @injectable() and @inject() decorators?**
   @Injectable(): This decorator marks a class as a provider that can be managed by the NestJS dependency injection system. It means that NestJS will create an instance of this class and can inject it where it's needed.
   It's typically used on services, which can then be injected into controllers or other services.

```
@Injectable()
export class CatsService {
  // ...
}
```

   @Inject():This decorator is used inside a class to inject a dependency. It's used in the constructor of a class to specify a dependency that should be injected.
   If you're injecting a class provider, you don't need to use @Inject() because TypeScript's reflection system can infer the type. But if you're injecting a non-class provider (like a value or a factory), or if you're working in JavaScript, you need to use @Inject() to tell NestJS what to inject.

```
 export class CatsController {
   constructor(@Inject('CatsService') private catsService: CatsService) {}
 }
```

6. **How does the Nest logger differ from the standard console.log() and when would you prefer one over the other?**
   The NestJS Logger provides additional features compared to console.log(). It includes context information, supports log levels such as (log, fatal, error,warn, debug, and verbose), and can be customized.
   Use console.log() for quick debugging or simple logging needs.
   Use NestJS Logger when you need more control over your logs, such as in larger or production applications. It helps in filtering logs based on levels and provides context, making it easier to trace the source of the log.
7. **What is the difference between interceptors and middleware?**
   In NestJS, both interceptors and middleware can be used to add extra logic before or after HTTP requests. However, they have some key differences:
   Interceptors: have a more comprehensive scope than middleware. They can be used with both HTTP requests and other types of transport like WebSockets and microservices.
   Interceptors can also manipulate the response sent back to the client, for example by transforming the response object, adding extra headers, or changing the status code. They can also be used to implement performance tracking, logging, caching, etc.
   Middleware: in NestJS is similar to Express middleware. It's specific to the HTTP request-response cycle and can't be used with other types of transport. Middleware functions have access to the request and response objects, and they can end the request-response cycle or call the next middleware function in the stack.
   They are useful for tasks like logging, error handling, or validating request data.
   In general, if you're working with HTTP requests and you need to add logic that doesn't modify the response sent to the client, middleware can be a good choice. If you need to add logic that applies to other types of transport such as WebSockets and microservices, or if you need to modify the response, use an interceptor.
8. **What testing frameworks work best with NestJS?**

NestJS is a Node.js framework, so any testing framework that works with Node.js will work with NestJS. Some popular options include Jest, Mocha, and Jasmine. NestJS is built with testing in mind and it comes with its own testing module called @nestjs/testing. This module provides utilities for testing, such as a testing module and HTTP testing utilities.

9. **Explain the purpose of DTOs (Data Transfer Objects) in NestJS.**
   DTOs are used to define the structure of data exchanged between different layers of an application. They define the shape of data for a specific operation, such as creating, updating, or returning data.
   DTOs serve several purposes which include: Validation: With the class-validator package, you can add validation rules to the fields in your DTOs. NestJS can automatically validate incoming requests against these rules and return an error if the request is invalid.
   Documentation: DTOs provide a clear model of what the data should look like. This can be helpful for other developers, and for tools like Swagger to automatically generate API documentation.
   Type Safety: DTOs provide type safety in TypeScript, which can help catch errors at compile time.

```
import { IsString, IsInt } from "class-validator";

export class CreateCatDto {
 @IsString()
 name: string;
 @IsInt()
 age: number;
 @IsString()
 breed: string;
}
```

   In the above example, CreateCatDto is a DTO that represents the data needed to create a cat. It expects a name and breed of type string, and an age of type number. The @IsString() and @IsInt() decorators are used to apply validation rules.

10. **How can you handle asynchronous operations in NestJS, and what is the role of the Promise object?**
    NestJS supports asynchronous operations through the use of the async and await keywords. When a function returns a Promise, it can be awaited, allowing non-blocking execution. The Promise object represents a value that may be available now, or in the future, or never.

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

@Injectable()
export class AppService {
 async getHello(): Promise<string> {
  const result = await someAsyncOperation();
  return `Hello ${result}`;
 }
}
```

    In this example, getHello() is an asynchronous method that returns a Promise. The await keyword is used to pause the execution of the function until someAsyncOperation() completes and the Promise is resolved.
    Promises are useful for handling single asynchronous operations. If you need to handle a stream of asynchronous values, you might want to use Observables instead, which are provided by the RxJS library and are also integrated into NestJS.

11. **Explain the purpose of the @InjectRepository() decorator in NestJS.**
    The @InjectRepository() decorator in NestJS is used to inject a repository instance into a service or controller. It is commonly used with TypeORM for database interaction, allowing the injection of repositories for specific entities.
    A repository in TypeORM is a way to manage entities: it provides methods to insert, update, remove, and load entities. By injecting a repository, you can use these methods in your service or controller.
    Here is an example:

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';

@Injectable()
export class UserService {
 constructor(
  @InjectRepository(User)
  private usersRepository: Repository<User>,
 ) {}

 findAll(): Promise<User[]> {
  return this.usersRepository.find();
 }
}
```

    In the above example @InjectRepository(User) is used to inject a repository for the User entity. This repository is then used in the findAll method to load all users.

12. **Describe the mechanism for a token refresh in NestJS. How can you implement an automatic token refresh strategy to maintain user sessions?**
    In NestJS, a token refresh strategy involves issuing a refresh token when a user logs in.

A refresh token is a special kind of token that can be used to obtain a new access token when the current one expires.
When the user logs in, along with the access token, a refresh token is also generated and sent to the client.
When the access token expires, the client sends the refresh token to the server, the server verifies the refresh token and issues a new access token.
This allows the user to stay authenticated without having to log in again, while still limiting the potential damage of a stolen access token.
Refresh tokens usually have a longer expiration time than access tokens, and they can be revoked by the server if needed, for example, in case of a logout.
Here is how you can implement a token refresh strategy:
Issue a Refresh Token: When a user logs in, along with the access token, issue a refresh token. This can be done similarly to how you issue an access token, but typically with a longer expiration time.
Store the Refresh Token: Store the refresh token in your database associated with the user. This allows you to invalidate the refresh token when necessary, such as when the user logs out.
Create a Refresh Endpoint: Create an endpoint in your application that accepts a refresh token and returns a new access token. In this endpoint, you should verify the refresh token, check that it hasn't been invalidated, and then issue a new access token.
Use the Refresh Token: On the client side, when you receive a 401 Unauthorized response, it means the access token has expired. In this case, send a request to the refresh endpoint with the refresh token to get a new access token. Replace the old access token with the new one in your client's storage.

13. **What is the difference between Provider and Services in Nestjs, can we have a provider without an injectable decorator, Give examples?**

A provider is a more general concept, while a service is a specific type of provider. Providers are a fundamental concept in NestJS system of dependency injection. They can be used to inject not only services, but also values, factories, and more.
A service is a class annotated with @Injectable() decorator, typically used to handle complex business logic or to provide access to shared data. Here's an example:

```
import { Injectable } from "@nestjs/common";
@Injectable()
export class CatsService {
  // ...
}
```

However, a provider doesn't necessarily need to be a service. It can be any value that should be available for injection. For example, you can provide a simple string:

```
  {
    provide: 'HelloMessage',
    useValue: 'Hello, World!',
  }
```

In this case, HelloMessage is a token that can be used to inject the string Hello, World!.
While services are typically decorated with @Injectable(), other types of providers don't need this decorator.
The @Injectable() decorator is needed when a class has its own dependencies that need to be injected. If the provider doesn't have any dependencies, like in the string example above, you don't need @Injectable().

14. **How can you generate API documentation using Swagger in NestJS? Discuss the importance of documenting your API and how it benefits developers?**

Generating API documentation in NestJS can be done using the @nestjs/swagger package. This package provides decorators and a SwaggerModule to easily create Swagger documentation.
Here is a basic setup:

- Start by installing the required dependency.
- npm install --save @nestjs/swagger

```
import { NestFactory } from "@nestjs/core";
import { SwaggerModule, DocumentBuilder } from "@nestjs/swagger";
import { AppModule } from "./app.module";

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  const config = new DocumentBuilder()
    .setTitle("Cats example")
    .setDescription("The cats API description")
    .setVersion("1.0")
    .addTag("cats")
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup("api", app, document);

  await app.listen(3000);
}
bootstrap();
```

In the above example, SwaggerModule.createDocument(app, config) generates the Swagger JSON. SwaggerModule.setup('api', app, document) serves the Swagger UI at the specified path ('api' in this case).
Documenting your API is important for several reasons:
Ease of Use: It helps other developers understand how to use your API. They can see the available endpoints, the expected request format, and the response format.

Testing: Tools like Swagger UI allow developers to test the API directly from the browser.
Maintenance: It helps maintain the API. When changes are made, the documentation serves as a reference to ensure the API's behavior remains consistent.
Onboarding: It speeds up the process of onboarding new developers. They can quickly understand the API's functionality without needing to dig into the codebase.

15. **Explain the purpose of the @nestjs/swagger ApiProperty(), ApiOperation() decorators?**
The @nestjs/swagger package provides several decorators to help with creating Swagger documentation for your API. Two of these decorators are @ApiProperty() and @ApiOperation().
@ApiProperty(): This decorator is used within DTO (Data Transfer Object) classes to provide metadata for properties. This metadata is used to generate the Swagger documentation for the model that the API expects in the request body or returns in the response.

```
import { ApiProperty } from "@nestjs/swagger";

export class CreateUserDto {
  @ApiProperty({
    description: "The id of the user.",
    minimum: 1,
    example: 42,
  })
  id: number;

  @ApiProperty({ description: "The name of the user.", example: "Thomas" })
  username: string;
}
```

The @ApiProperty is used to indicate that id and name are properties of the CreateUserDto
@ApiOperation(): This decorator is used within controller methods to provide metadata for operations (API endpoints). This metadata is used to generate the Swagger documentation for the operation. It allows developers to provide additional information such as operation summary, description, and custom tags, enhancing the generated Swagger documentation.
Below is an example:

```
import { ApiOperation } from "@nestjs/swagger";

@Controller("users")
export class UsersController {
  @Post()
  @ApiOperation({ summary: "Create user" })
  create(@Body() createUserDto: CreateUserDto) {
    // ...
  }
}
```

@ApiOperation() is used to provide a summary for the create operation. This summary will be displayed in the Swagger UI.
There are more decorators that are used to display error messages to the user such as @ApiNotFoundResponse,@ApiBadRequestResponse, @ApiInternalServerErrorResponse and many more. While decorators that display success messages include @ApiOkResponse, @ApiCreatedResponse etc.

16. **Explain the purpose of the Dockerfile in a NestJS application, and how it facilitates containerization?**
A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. In the context of a NestJS application, a Dockerfile is used to create a Docker image of the application.
A Docker image is a lightweight, standalone, executable package of software that includes everything needed to run an application. It includes code, runtime, system tools, system libraries, and settings.
This image can be run consistently on any machine that has Docker installed, regardless of the underlying operating system.
Here is an example of a Dockerfile for Nest application:

```
// Start from a base image
FROM node:14-alpine // or node:latest to use the latest version of node

// Set the working directory
WORKDIR /usr/src/app

//Install dependencies
COPY package*.json ./
RUN npm install

// Copy source code
COPY . .

// Expose the application on port 3000
EXPOSE 3000

// Start the application
CMD ["npm", "run", "start"]
```

This Dockerfile does the following:
Starts from a base image with Node.js installed (node:14-alpine).
Sets the working directory in the container to /usr/src/app.
Copies package.json and package-lock.json (if available) to the working directory.
Installs the dependencies using npm install.
Copies the rest of the source code to the working directory.
Exposes port 3000, which the application uses.
Defines the command to start the application (npm run start).
Note: The benefit of this is that it encloses the application and its environment into a single runnable entity (a container). This ensures that the application runs the same way, regardless of where it's deployed, providing consistency and reliability across different deployment environments.

17. **How can you use Docker Compose with NestJS, and what is its role in a multi-container setup?**
Docker Compose Docker Compose is a tool for defining and running multi-container applications. Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single, comprehensible YAML configuration file. Then, with a single command, you create and start all the services from your configuration file.
Here's a basic example of a docker-compose.yml file for a NestJS application with a PostgreSQL database:

```
version: '3'
services:
  app:
    build: .
    ports:
      - 3000:3000
    depends_on:
      - db
  db:
    image: postgres:13-alpine
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: dbname
```

In this example, there are two services: app and db. The app service is built using the Dockerfile in the current directory, and it exposes port 3000. The db service uses the postgres:13-alpine image and sets some environment variables to configure the database.
The depends_on option is used to express dependency between services, which has two effects:
db will be started before app. Docker Compose will wait until db is "ready" before starting app. To start the application with Docker Compose, you would use the command docker-compose up.
The benefit of using Docker Compose is that it simplifies the management of multi-container applications. You can start, stop, and rebuild services with a single command, and it ensures that your application's services are started in the correct order.

18. **What is the purpose of the @nestjs/passport package, and how does it facilitate authentication in NestJS?**
@nestjs/passport is a popular node.js authentication library. The main purpose is to facilitate authentication in a NestJS application. It does this by providing a way to implement different authentication strategies (like local, JWT, OAuth, etc.) in a consistent and modular way. It provides a set of tools that make it easier to implement authentication in a NestJS application using Passport.js.
Here is a basic example of how you might use @nestjs/passport

```
import { Injectable } from "@nestjs/common";
import { AuthGuard } from "@nestjs/passport";

@Injectable()
export class JwtAuthGuard extends AuthGuard("jwt") {}
```

19. **How can you handle file uploads in NestJS, and what is the role of the Multer library?**
NestJS offers convenient ways to handle file uploads, and one common approach is using the multer middleware. Additionally, the framework provides the @UploadedFile decorator, which simplifies the process of accessing and processing uploaded files in NestJS controllers. These tools collectively offer a flexible and efficient solution for managing file uploads in NestJS applications.
By using the @UseInterceptors() decorator with FileInterceptor or FilesInterceptor, you can handle single or multiple file uploads in your application.

20. **What is Circular dependency (dependency cycle) in Nestjs, and how can they be fixed?**
A circular dependency occurs when two classes depend on each other. For example, class A needs class B, and class B also needs class A. Circular dependencies can arise in Nest between modules and between providers.
Nest enables resolving circular dependencies between providers in two ways.
1.forward referencing: allows Nest to reference classes which aren't yet defined using the forwardRef() utility function. For example, if CatsService and CommonService depend on each other, both sides of the relationship can use @Inject() and the forwardRef() utility to resolve the circular dependency. Otherwise Nest won't instantiate them because all of the essential metadata won't be available. Here's an example:

```
import { forwardRef } from '@nestjs/common'

@Injectable()
export class CatsService {
```

```
  constructor(
    @Inject(forwardRef(() => CommonService))
    private commonService: CommonService,
  ) {}
 }
Now let's do the same with CommonService
import { forwardRef } from '@nestjs/common'

@Injectable()
 export class CommonService {
  constructor(
    @Inject(forwardRef(() => CatsService))
    private catsService: CatsService,
  ) {}
 }
```

2.ModuleRef class: An alternative to using forwardRef(). for an example you can refactor the above examples and use the ModuleRef class to retrieve a provider on one side of the (otherwise) circular relationship.

21. **How does NestJS handle CORS (Cross-Origin Resource Sharing)?**

CORS (Cross-Origin Resource Sharing) is a mechanism that allows many resources (e.g., fonts, JavaScript, etc.) on a web page to be requested from another domain outside the domain from which the resource originated.
In the context of web development, an API server runs on a different domain or port from the client-side web application. For security reasons, browsers prohibit web pages from making requests to a different domain than the one the web page came from, unless the server supports CORS.
By enabling CORS on the server, you're allowing the server to respond to cross-origin requests. This means that your server's resources can be accessed from a different domain, protocol, or port than the one your server is hosted on.
NestJS uses the capabilities of the underlying platform (Express or Fastify) to handle Cross-Origin Resource Sharing (CORS). For Express, you can enable CORS globally for all routes in your main.ts file like this:

```
import { NestFactory } from "@nestjs/core";
import { AppModule } from "./app.module";

async function bootstrap() {
 const app = await NestFactory.create(AppModule);
 app.enableCors({
   origin: "http://localhost:3000",
   methods: "GET,HEAD,PUT,PATCH,POST,DELETE",
   allowedHeaders: "Content-Type",
 });
 await app.listen(3000);
}
bootstrap();
```

In this example, CORS is enabled only for requests from 'http://localhost:3000' and for the specified methods and headers.

22. **Explain the purpose of the ExecutionContext in NestJS Middleware?**

ExecutionContext can be used to access the Request and Response objects, or any other details about the current request-response cycle. This can be useful for tasks like logging, validation, transformation, and other operations that need to be performed before the request reaches the route handler.

23. **How can you implement soft deletes in NestJS using TypeORM, and why might soft deletes be preferred over hard deletes?**

Soft deletes in TypeORM are implemented using the @DeleteDateColumn decorator.When you delete an entity that has a @DeleteDateColumn, TypeORM doesn't actually remove it from the database. Instead, it sets the @DeleteDateColumn to the current timestamp. This is known as a "soft delete".
Here's an example of how you might use @DeleteDateColumn in an entity:

```
import { Entity, PrimaryGeneratedColumn, Column, DeleteDateColumn } from 'typeorm';

 @Entity()
 export class User {
   @PrimaryGeneratedColumn()
   id: number;

   @Column()
   name: string;

   @DeleteDateColumn()
   deletedAt?: Date;
 }
```

In this example, when you call userRepository.softDelete(user.id), TypeORM will set deletedAt to the current timestamp, but the User will remain in the database.
Soft deletes can be preferred over hard deletes for a few reasons:
1.Data recovery: If a record is accidentally deleted, it can be easily restored.

i. Audit trail: Soft deletes allow you to keep a history of all records, even ones that are deleted.

ii. Relationship integrity: If other tables reference the deleted record, those relationships won't be broken by a soft delete.

24. Explain the concept of environment variables in NestJS, and how can they be utilized for configuration management?

Environment variables are a way to store configuration settings that can change between different environments (like development, staging, production, etc.). They are often used to store sensitive information like database credentials, API keys, or any other configuration that might change depending on the environment.

NestJS provides a ConfigModule that uses the dotenv package to load environment variables from a .env file into process.env.

Here's an example of how you might use ConfigModule to load environment variables:

```
import { Module } from "@nestjs/common";
import { ConfigModule } from "@nestjs/config";

@Module({
 imports: [ConfigModule.forRoot()],
})
export class AppModule {}
```

In the above example, ConfigModule.forRoot() loads the .env file and the variables can be accessed anywhere in your application using process.env.

25. What is the purpose of ExecutionContext in NestJS?

ExecutionContext represents the context of the currently processed HTTP request. It contains information about the request, response, route handler, and other details. ExecutionContext is often used in custom decorators, guards, and interceptors to access and manipulate request-related information.

26. What is the purpose of the @Res() decorator in NestJS controllers?

Nest provides @Res() and @Response() decorators. @Res() is simply an alias for @Response(). @Res() or @Response()allows you to directly interact with the response object and use its methods. When using them, you should also import the typings for the underlying library (e.g., @types/express) to take full advantage.

Here is an example of using `@Res()` decorator:

```
import { Controller, Get, Res } from "@nestjs/common";
import { Response } from "express";

@Controller("cats")
export class CatsController {
 @Get()
 findAll(@Res() res: Response) {
  res.status(200).send("This action returns all cats");
 }
}
```

Note When you inject either @Res() or @Response() in a method handler, you put Nest into Library-specific mode for that handler, and you become responsible for managing the response. When doing so, you must issue some kind of response by making a call on the response object (e.g., res.json(...) or res.send(...)), or the HTTP server will hang.

27. What is the entry file of NestJs application?

The entry file of a NestJS application is typically main.ts. This is where the application's root module is bootstrapped, which kickstarts the application.

Here's an example of what the main.ts file might look like:

```
import { NestFactory } from "@nestjs/core";
import { AppModule } from "./app.module";

async function bootstrap() {
 const app = await NestFactory.create(AppModule);
 await app.listen(3000);
}
bootstrap();
```

NestFactory.create(AppModule) initializes the application with the root module (``AppModule), and app.listen(3000)` starts the application, listening for incoming requests on port 3000.

28. What is the difference between dependency injection and inversion of control (IoC)?

Dependency Injection (DI) and Inversion of Control (IoC) are both design patterns used to reduce the coupling between classes, making the code more modular, easier to test and maintain. However, they are not the same thing, but rather, DI is a form of IoC.

Inversion of Control (IoC) is a general principle where the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the external framework or runtime controls it.

Dependency Injection (DI) is a form of IoC where the creation and binding of dependent objects is controlled by a container or a framework. Instead of a class creating or finding its dependencies, they are passed in (injected) at runtime by another piece of code, typically a container or a framework. This makes the code more flexible, testable and modular because it decouples the usage of an object from its creation. This is the form of IoC that is used in many modern frameworks such as Angular, Spring, and NestJS.

In summary IoC is a design principle which can be implemented in several ways, one of which is DI

29. How can you implement Caching in NestJS?

Caching is a great and simple technique that helps improve your app's performance. It acts as a temporary data store providing high performance data access.

NestJS supports caching through various mechanisms, including the use of caching libraries like cache-manager and built-in decorators such as @CacheKey and @CacheTTL. By incorporating caching strategies in your application, you can enhance performance and reduce response times for frequently requested data.
In order to enable caching, import the CacheModule and call its register() method

```
import { Module } from "@nestjs/common";
import { CacheModule } from "@nestjs/cache-manager";
import { AppController } from "./app.controller";

@Module({
 imports: [CacheModule.register()],
 controllers: [AppController],
})
```

export class AppModule {}
To interact with the cache manager instance, inject it to your class using the CACHE_MANAGER token, as follows:
constructor(@Inject(CACHE_MANAGER) private cacheManager: Cache) {}
The get method is used to retrieve items from the cache. If the item does not exist in the cache, null will be returned.
const value = await this.cacheManager.get("key");
To add an item to the cache, use the set method:
await this.cacheManager.set("key", "value");
The default expiration time of the cache is 5 seconds, however you can change this.
await this.cacheManager.set("key", "value", 1000);
To disable expiration of the cache, set the ttl configuration property to 0:
await this.cacheManager.set("key", "value", 0);
To remove an item from the cache, use the del method:
await this.cacheManager.del("key");
To clear the entire cache, use the reset method:
await this.cacheManager.reset();

30. **Explain the purpose of the Dependency Inversion Principle (DIP) in NestJS?**

The Dependency Inversion Principle (DIP) is one of the five principles of SOLID, an acronym. The principle states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

In the context of NestJS, or any other framework that supports dependency injection, the purpose of DIP is to reduce the coupling between modules, making the system more flexible, easier to test, and easier to maintain.
By depending on abstractions, not on concrete implementations, you can easily swap out modules without changing the high-level code. For example, you might have a service that depends on a repository. If you code to an interface, you can easily change the repository (e.g., from an in-memory repository to a database repository) without changing the service code.
NestJS supports DIP through its modular system and the use of decorators like @Injectable(), @Inject(), and custom providers. These features allow you to define providers and inject them where needed, making it easy to manage dependencies and adhere to the Dependency Inversion Principle.

31. **How can you schedule tasks in NestJS?**

Task scheduling allows you to schedule arbitrary code (methods/functions) to execute at a fixed date/time, at recurring intervals, or once after a specified interval.
Nest provides the @nestjs/schedule package, which integrates with the popular Node.js cron package.

i. To implement scheduling, you can install the required dependencies.

```
npm install --save @nestjs/schedule
```

ii. Then, import the ScheduleModule into your module:

```
import { Module } from "@nestjs/common";
import { ScheduleModule } from "@nestjs/schedule";
import { TasksService } from "./tasks.service";

@Module({
 imports: [ScheduleModule.forRoot()],
 providers: [TasksService],
})
export class TasksModule {}
```

iii. Now, you can use the decorators in your service to schedule tasks. Here's an example:

```
import { Injectable } from "@nestjs/common";
import { Cron, CronExpression } from "@nestjs/schedule";

@Injectable()
export class TasksService {
 @Cron(CronExpression.EVERY_5_SECONDS)
 handleCron() {
  console.log("Called every 5 seconds");
 }
}
```

In the above example, handleCron() will be called every 5 seconds. The @Cron() decorator takes a CronExpression which determines the schedule.
Remember, the ScheduleModule uses the node-schedule package under the hood, so you can use any cron expression that node-schedule supports.

32. **How can you handle database transactions in NestJS, and why are transactions important in certain scenarios?**
    Database transactions in NestJS can be handled using the TypeORM package. Transactions are important when you want to ensure data integrity. If a series of database operations need to succeed or fail together, transactions can ensure that if any operation fails, all changes are rolled back and the database remains in a consistent state.
    This is particularly important in scenarios such as financial operations, where it's crucial that either all parts of a transaction are completed or none of them are.
33. **How can you implement versioning in NestJS APIs?**
    Versioning allows you to have different versions of your controllers or individual routes running within the same application.
    There are 4 types of versioning that are supported:
    i. URI Versioning: The version will be passed within the URI of the request (default).
    ii. Header Versioning: A custom request header will specify the version.
    iii. Media Type Versioning: The Accept header of the request will specify the version.
    iv. Custom Versioning: Any aspect of the request may be used to specify the version(s). A custom function is provided to extract said version(s).
    To enable Header Versioning for your application, do the following:

```
const app = await NestFactory.create(AppModule);
app.enableVersioning({
  type: VersioningType.HEADER,
  header: "Custom-Header",
});
await app.listen(3000);
```

34. **Explain the purpose of the @nestjs/graphql Resolver and @nestjs/graphql Scalar decorators, and how they relate to GraphQL in NestJS?**
    GraphQL is a powerful query language for APIs and a runtime for fulfilling those queries with your existing data. It's an elegant approach that solves many problems typically found with REST APIs.
    The @nestjs/graphql package provides decorators that allow you to define GraphQL schemas directly from your TypeScript classes.
    i. @Resolver(): This decorator is used to mark a class as a GraphQL resolver. Resolvers are the building blocks of GraphQL servers. They are responsible for fetching the data for individual fields in a schema. When you send a query to a GraphQL server, the server uses resolver functions to produce a response.
    ii. @Scalar(): This decorator is used to define a custom scalar. Scalars are primitive values: Int, Float, String, Boolean, or ID. When you need to use a custom primitive type (like a Date or a type from your database), you can define a custom scalar.
35. **Explain the concept of Serialization and Deserialization in NestJS?**
    Serialization and deserialization are fundamental concepts in computer science, not just in NestJS. They are used when data needs to be converted into a format that can be stored or transmitted and then reconstructed later.
    Serialization: This is the process of converting a data structure or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment).
    Deserialization: This is the reverse process of serialization, where the serialized format is converted back into an actual object in memory.
    In the context of NestJS, these concepts are often used when dealing with HTTP requests and responses. For example, when you send data from a client to a NestJS server, the data is serialized into a JSON format, transmitted over the network, and then deserialized back into a JavaScript object on the server.
    NestJS provides a Pipes mechanism that can be used for data transformation (serialization/deserialization) and validation. For example, the ValidationPipe provided by NestJS can be used to automatically validate and transform incoming request payloads into instances of DTO classes.
36. **Explain the role of NestJS middleware in the context of Microservices and provide a scenario where middleware is beneficial in a Microservices setup?**
    Middleware is a function that is executed before the route handler. Middleware functions have access to the request and response objects, and the next() middleware function in the application's request-response cycle. They can execute any code, make changes to the request and the response objects, end the request-response cycle, and call the next middleware function in the stack.
    In the context of microservices, middleware can play several important roles:
    i. Request Logging: Middleware can be used to log details of incoming requests. This can be useful for debugging, as well as for tracking usage patterns and user behavior.
    ii. Authentication and Authorization: Middleware can verify a user's identity and permissions before a request reaches a service. This can prevent unauthorized access and ensure that each service doesn't have to implement its own authentication checks.
    iii. Error Handling: Middleware can catch and handle errors. This can help to ensure that the microservice responds with a well-defined error structure even when something goes wrong.
    iv. Rate Limiting: Middleware can track the number of requests from a client and limit usage to prevent abuse.
37. **Discuss the different types of coupling, such as tight coupling and loose coupling, and provide examples of how NestJS modules contribute to achieving loose coupling in a modularized application.**
    Coupling is the degree of interdependence between software modules, a measure of how closely connected two routines or modules are, the strength of the relationships between modules.
    Tight Coupling: In this case, a module (or class) is highly dependent on another module. Changes in one module may require changes in the dependent module. This makes the system harder to maintain and evolve over time.
    Loose Coupling: Here, a module is not highly dependent on other modules. Changes in one module have minimal or no effect on other modules. This makes the system more maintainable and adaptable to change.

NestJS promotes loose coupling through its modular development structure. Each module contains a portion of the application's functionality and can operate independently of other modules. This means that changes in one module do not affect others, leading to a loosely coupled system.
Here is a more easier example:

```
// users.service.ts
 import { Injectable } from '@nestjs/common';
 import { User } from './user.entity';

 @Injectable()
 export class UsersService {
  private users: User[] = [];

  create(user: User) {
   this.users.push(user);
  }

  findAll(): User[] {
   return this.users;
  }
 }

 // meal.service.ts
 import { Injectable } from '@nestjs/common';
 import { UsersService } from '../users/users.service';

 @Injectable()
 export class MealService {
  constructor(private usersService: UsersService) {}

  createMeal(userId: string, MealData: CreateMealDTO) {
   const user = this.usersService.findById(userId);
   // Create meal for the user
  }
 }
```

In above example, MealService depends on UsersService, but it doesn't manipulate user data directly. This is an example of loose coupling.

38. **How does NestJS support Server-Sent Events (SSE), and what are the primary advantages of using SSE for real-time communication in web applications?**

Server-Sent Events (SSE) is a server push technology enabling a client to receive automatic updates from a server via HTTP connection. SSE is a one-way communication channel from server to client. If you need bi-directional communication, you might want to use WebSockets instead.
SSE they have been used in Facebook/Twitter updates, stock price updates, news feeds etc.
To enable Server-Sent events on a route (route registered within a controller class), annotate the method handler with the @Sse() decorator.

```
 @Sse('sse')
  sse(): Observable<MessageEvent> {
   return interval(1000).pipe(map((_) => ({ data: { hello: 'world' } })));
 }
```

In the example above, we defined a route named sse that will allow us to propagate real-time updates. These events can be listened to using the EventSource API.
Server-Sent Events (SSE) have several advantages for real-time communication in web applications:
1.Built on HTTP: SSE is built on HTTP, which makes it compatible with most firewalls and networks without requiring any special configuration.
2.Automatic Reconnection: If a connection is lost, the browser will automatically try to reconnect to the server.
3.Event IDs: The server can send an ID with each event, so if a client gets disconnected, it can reconnect and get all the events it missed.
4.Efficient Updates: SSE is ideal for applications that require real-time updates from the server (like live news updates, real-time analytics, etc.). The server can push updates to the client as soon as new data is available.