Nest.js Course Manual

Chapter 1 Getting Started with Nest.js

What is Nest.js

Managing a large-scale application can be tedious, especially when built without a well-planned structure and strict code organization strategy. Nest.js aids in this by enforcing modularity, thereby following the Single Responsibility Principle, a core tenet of solid software engineering.

Nest.js is a Node.js framework designed for crafting efficient, reliable, and scalable server-side applications. Built on top of Express and TypeScript, it adopts Node.js and Angular design patterns to provide a cohesive development experience. The framework's architecture embraces decorators and dependency injection, patterns borrowed from Angular, to facilitate better code organization and reusability.

What Problem Nestjs Solves

Maintaining consistency across a large codebase can be difficult; Nest.js addresses this by incorporating software engineering design patterns like SOLID and Dependency Injection. These design patterns contribute to a clean, maintainable, and scalable code architecture, adhering to Nest.js's principle of creating easily testable and loosely coupled code.

While unopinionated frameworks like Express offer multiple ways to structure your code, the freedom to choose can lead to decision paralysis, wasting time that could be better-spent building features. Nest.js sidesteps this issue by providing a modular approach right out of the box, allowing for flexible code organization while still enforcing a structured layout, aligning with its principle of modularity.

Sticking with an initial code organizational decision can be difficult, particularly as team members come and go. Nest.js's structured, modular approach helps maintain architectural integrity over time, making it easier for new developers to understand the codebase quickly.

Essentially, the issue Nest aims to solve is creating a robust architecture for backend applications. By providing a well-defined structure and integrating established design patterns, it answers a pressing question many developers have about organizing architecture for enterprise or large server-side applications.

Why Nest.js was created

The creator of Nest.js drew inspiration from Angular's design architecture to build a front-end application. Transferring this idea to the server side allowed for the broad utilization of a familiar architecture that incorporates Nest. js-specific elements like modules, dependency injection, providers, and custom decorators. This strategy embodies the Nest.js principle of modularity and extensibility, making it easier for developers who are already familiar with Angular to adopt Nest.js.

Why should you use Nest.js

If you aim to build server-side APIs utilizing TypeScript, Nest.js is the framework of choice. Nest.js employs strong typing and decorators, elements intrinsic to TypeScript, thus aligning closely with TypeScript principles. It also supports Dependency Injection out-of-the-box, encouraging a modular, scalable architecture.

Nest.js is particularly accessible for developers familiar with Angular, making the learning curve more manageable. Both frameworks share common programming paradigms and syntax, including the use of decorators and modules, enhancing code reusability and maintainability.

Nest.js enables you to construct various backend services, from RESTful APIs and GraphQL endpoints to MVC applications and WebSockets. The framework's flexibility accommodates multiple communication patterns, allowing for an extensive range of backend solutions.

Designed for building both large-scale monolithic and microservices applications, Nest.js offers scalability and modular architecture. It employs a "Module" system to organize code, which allows for straightforward unit testing and easier collaboration.

Nest.js provides native integrations with a host of external tools like Mongoose for MongoDB, TypeORM for various databases including Postgres, and many more. This extensibility allows developers to incorporate a multitude of functionalities without boilerplate code, making the framework more versatile.

Benefits of using Nest.js

- Utilize Angular-style syntax for the backend, a feature unique to Nest.js, that promotes consistency and reusability across the tech stack. This syntax allows developers familiar with Angular to seamlessly transition into backend development.

- Leverage the detailed documentation with examples that Nest.js provides. This not only accelerates the learning curve but also aligns with Nest.js's principle of being developer-friendly and easy to pick up.
- Benefit from the framework's focus on good architecture and fast development. Nest.js follows the modular architecture pattern, allowing for better separation of concerns and easier testing, which in turn speeds up the development process.

What type of Application can you build

- You can build backend REST and GraphQL APIs using Nest.js, benefiting from its modular architecture that promotes code reusability and maintainability. The framework's built-in support for these query languages facilitates quick API development.
- You can build microservices with Nest.js, taking advantage of its inherent support for multiple message transport layers. This aligns with the microservices architecture principle, which advocates for loosely coupled, independently deployable components.
- You can construct a backend for a streaming application using Nest.js, as the framework supports asynchronous data handling through Observables. Nest.js makes it easier to work with real-time data streams, thanks to its integration with libraries like RxJS.
- You can build the backend of a real-time application in Nest.js, utilizing its WebSocket support for real-time two-way communication. This is essential for applications requiring instant data update and interaction, adhering to the principles of real-time system design.

Who is using Nest.js in production

- Roche is a multinational healthcare company operating under two divisions, pharmaceuticals and diagnostics. This American biotechnology company uses the Nest.js framework on its main website to reliably cater to its patients and to further expand its services.

- Adidas is the largest sportswear manufacturer in Europe and the second-largest in the world. Adidas is known for designing and manufacturing shoes, clothing, and accessories. Their global reach and popularity are why they decided to use the highly scalable Nest.js framework to build large-scale, efficient applications for their brand.
- Decathlon is a sporting goods retailer with over 1,500 stores in 57 countries. Decathlon chose Nest.js as the backend for their web application to help them scale and maintain their legacy codebase.

Creating a Nestjs Project

Before creating the Nestjs project. Make sure you installed [Node. js](#) on your machine.

Steps to create Nest.js project

1. Install the `nest-cli` globally

2. `npm install -g @nestjs/cli`

Create a new project using Nest cli

You have successfully installed the nest cli. Now it is time to create a new project using the Nest cli command `nest new [name of the project]`

`nest new n-fundamentals`

Start the Project

The project with the `n-fundamentals` name will be created. You can choose any project name here. When the project is created successfully you can start the project by using `npm run start:dev`

The project will be running at `http://localhost:3000`

Project Directory structure

Nest.js designates the src folder as the location where you'll place your application's source code. This arrangement adheres to Nest.js's modular architecture, promoting better organization and separation of concerns within your application.

Within the src folder, the main.ts file serves as the entry point of your Nest.js application. This file is responsible for bootstrapping the application, utilizing Nest.js's core function NestFactory to create an instance of your Nest application.

```
import { NestFactory } from "@nestjs/core";
import { AppModule } from "./app.module";
async function bootstrap() {
const app = await NestFactory.create(AppModule);
await app.listen(3000);
}
bootstrap();
```

The main.ts file serves as the entry point for the application and employs the core function NestFactory to instantiate a Nest application. In Nest.js, NestFactory is pivotal for bootstrapping the application, setting up the dependency injection system, and initializing modules. This follows the Nest.js principle of modular development and centralized configuration.

- app.controller.ts: This is a basic controller file containing a single route. Controllers in Nest.js are responsible for request handling and response sending, acting as a gateway between client and server.
- app.controller.spec.ts: This file contains unit tests for the controller, adhering to the Nest.js focus on test-driven development (TDD).
- app.module.ts: This is the root module of the application, which imports other modules and providers. Nest.js modules act as organizational units and follow the Single Responsibility Principle.
- app.service.ts: A basic service file with a single method. In Nest.js, services encapsulate business logic and can be injected into controllers, promoting Dependency Injection and the Separation of Concerns.

- nest-cli.json: Utilized for Nest.js-specific configurations, this file allows customization of compiler options, assets, and other settings.
- .prettierrc: This file is used for configuring Prettier, aiding in code formatting and style consistency within the Nest.js project.
- tsconfig.json: This configuration file is for TypeScript and determines how the TypeScript compiler will behave. This aligns with Nest.js's use of TypeScript for strong typing and better code quality.

Chapter 2 Creating REST APIs

What is a module in Nest.js
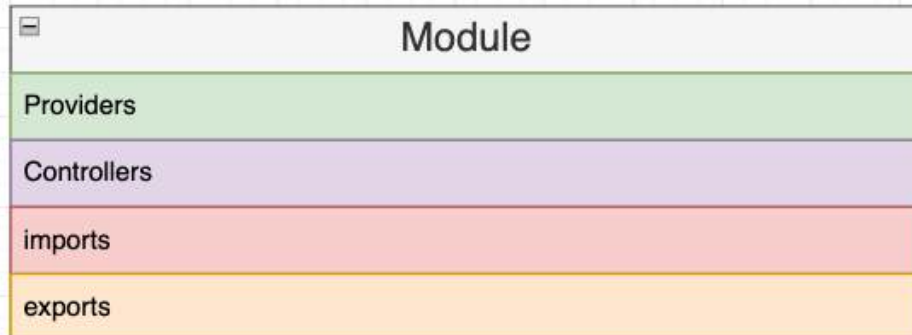
A module serves as the foundational building block of a Nest.js application, adhering to the Nest.js principle of modularity for better code organization. Each Module encapsulates Providers, Controllers, Imports, and Exports, acting as a cohesive unit of related functionality.

- Providers in Nest.js are classes that act as services, factories, or repositories. They encapsulate business logic and can be injected into controllers or other services.
- Controllers serve the function of handling incoming HTTP requests and sending responses back to the client, aligning with the Nest.js use of the controller pattern for request handling.
- Imports is an array that specifies the external modules needed for the current module, enabling code reusability and separation of concerns.
- Exports are utilized to make services available to other modules, aligning with the Nest.js emphasis on encapsulation and modular design.

Your application will contain a Root Module, which is specific to the Nest.js framework. The Root Module serves as the entry point and is responsible for instantiating controllers, providers, and other core elements of the application. In Nest.js, this architecture follows the "Module Isolation" principle, ensuring that the application is organized into distinct functional or feature-based modules.

```
@Module({

 imports: [SongsModule],

 controllers: [AppController],

 providers: [AppService],

})
export  class  AppModule {}
```

We are going to build the backend of the Spotify application. We can divide our application into Modules in this way
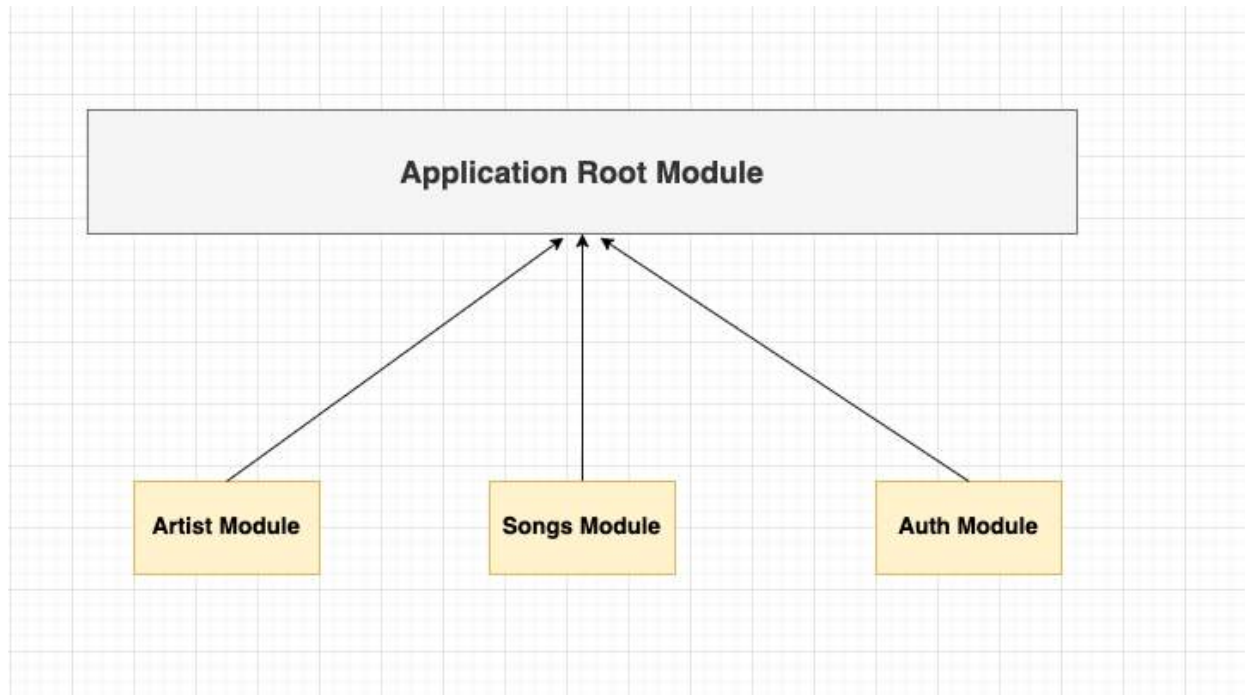
Divide your application's use cases into feature modules, such as the Artist Module, Songs Module, and Auth Module. In Nest.js, modules are a fundamental organizational unit that follow the Modularization principle, enabling better code reusability and separation of concerns. This approach streamlines development, as modules encapsulate related functionalities and can be developed or maintained independently.

Each Module will have its own providers, services, and controllers.

Now it's time to create a module. Create a songs module using nest cli

```
nest g module songs
```

A songs module will be created in your application. It will also add the SongsModule entry in AppModule

What is a Nest.js Controller
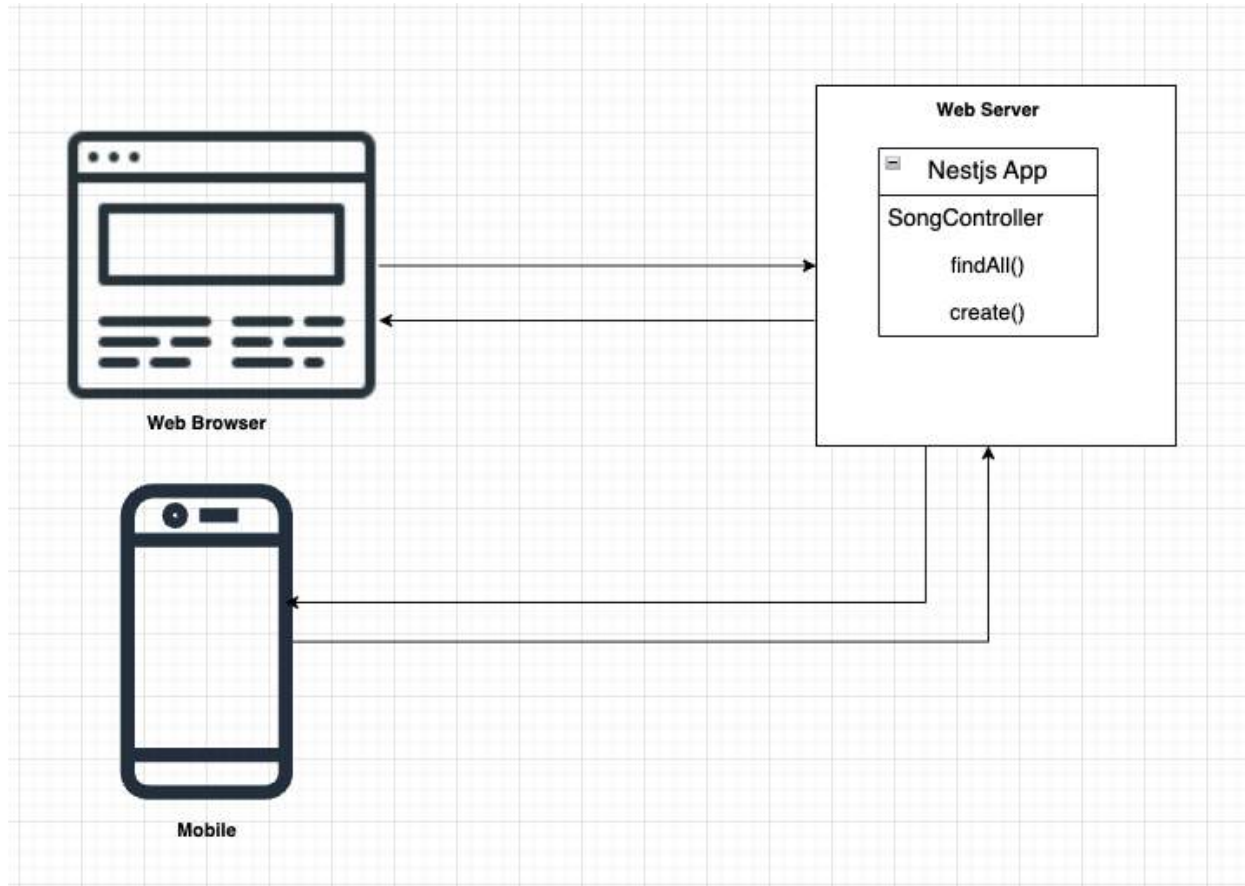
Controllers in Nest.js are responsible for handling incoming requests and managing the logic to send responses back to the client. They act as the "C" in the MVC (Model-View-Controller) pattern that Nest.js leverages for application architecture.

In the context of building the backend of a Spotify-like application, let's say you want to fetch all songs by the artist Martin Garrix. The responsibility to handle this type of request lies primarily with controllers, specific to this use case— the `SongsController`.

Your browser will initiate a request to fetch all songs. In your Nest.js application, you'll handle this through the `SongsController` and its `findAll` method, which is

specifically designed to interact with underlying services to retrieve data and send it back to the client.

Create these endpoints in the application.

```
GET http://localhost:3000/songs
```

```
GET http://localhost:3000/songs/1
```

```
POST http://localhost:3000/songs
```

```
PUT http://localhost:3000/songs/1
```

```
DELETE http://localhost:3000/songs/1
```

You can create a controller very easily. We are going to use the Nest cli to create a controller

```
nest g controller songs
```

Have the `SongsController` inside the songs directory. Nestjs will also added the entry for the SongsController in SongsModule.

Create these endpoints in the controller

```
@Controller("songs")
export class SongsController {
@Post()
create() {
```

```
    return  "create a new song endpoint";
  }

  @Get()
  findAll()  {
  return  "find all songs endpoint";
  }
  @Get(":id")
  findOne()  {
  return  "fetch song on the based on id";
  }

  @Put(":id")
  update()  {
  return  "update song on the based on id";
  }



  @Delete(":id")
  delete()  {
  return  "delete a song on the based on id";
  }
  }
```

We're sending simple messages to indicate that the route has been created. In Nest.js, you can specify dynamic parameters `id1` in your route by using a colon followed by the parameter name, like `@Get(':id')`. This follows the Nest.js principle of utilizing decorators to handle common HTTP tasks, streamlining the codebase and making it more readable.

What is Service

Services in Nest.js are providers, meaning you can inject them into modules and classes through dependency injection. In Nest.js, a service is not just a construct but a first-class citizen, managed by the framework's built-in Inversion of Control (IoC) container. Unlike in Express.js, where middleware or simple JavaScript functions

often serve the same purpose but without formal dependency management, Nest.js services offer a structured way to write business logic, making the application more maintainable and testable.
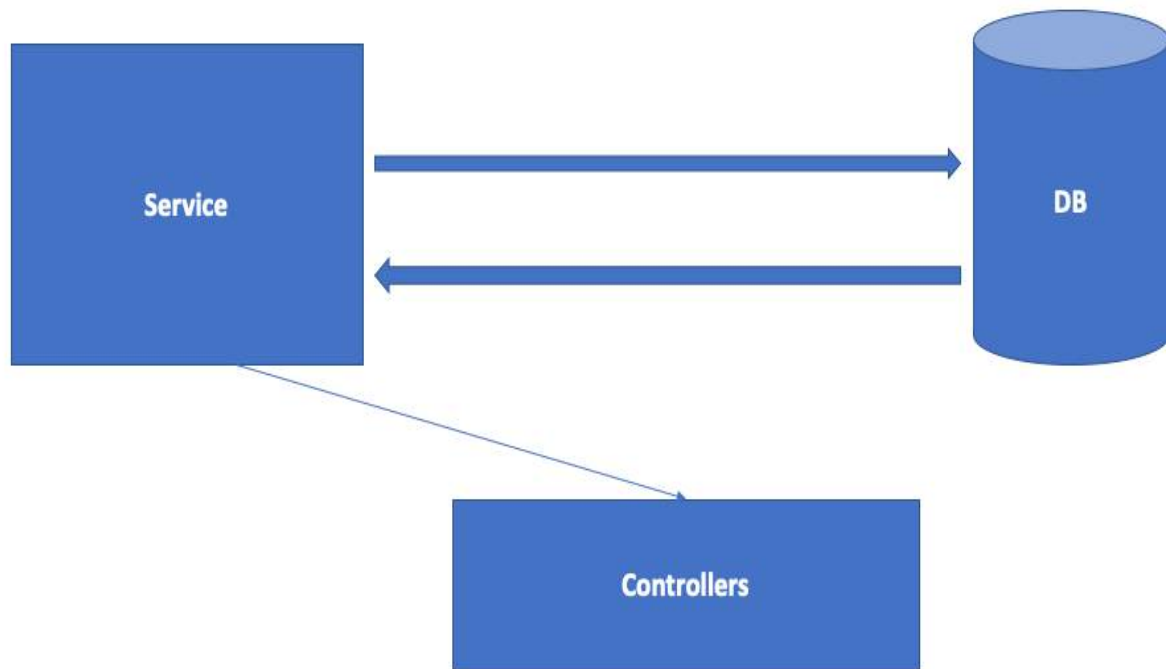
A service in Nest.js is responsible for fetching data from the database and saving data back to the database, functioning as a liaison between the controller and the database. This concept aligns with Nest.js's adherence to the Single Responsibility Principle, separating the business logic from the controller layer, a stark contrast to Express.js where the roles are often less clearly delineated.

A service can be injected into a controller using Nest.js's built-in Dependency Injection system. You can also export the service from the current module, enabling its use in other parts of that specific module. This is another feature where Nest.js differentiates itself from frameworks like Express.js, offering native support for modularity and code reuse through its export system.

In this lesson, we'll focus on creating the song service, a specific component tailored to manage song-related data. Creating specialized services for different aspects of

your application promotes better maintainability and is a cornerstone of Nest.js's modular architecture.

Creating a Service

```
@Injectable()

export class SongsService {}
```

Injecting Service

```
@Controller('songs')

export class SongsController {

constructor(private songsService: SongsService){

}
```

We can inject the service into a constructor function. Now the SongsController is dependent on SongsService

Let's create a new service by using nest-cli

nest g service songs

This command will create the SongsService inside the songs folder. One more thing, It will register the SongsService into SongsModule in the provider array

```
@Module({
```

```
  controllers: [SongsController],

  providers: [SongsService],

})

export  class  SongsModule  {}
```

Let's create an array of songs and this service will interact with the songs array. We will interact with the database next section.

```
@Injectable()
export class SongsService {
  // local DB
  // local array
  private readonly songs = [];
  create(song) {
    // Save the song in the database
    this.songs.push(song);
    return this.songs;
  }
  findAll() {
    // fetch the songs from the db
    return this.songs;
  }
}
```

Now a songs array has been created and two methods: create and findAll.

- Use findAll to send the array of songs in the response. In Nest.js, this is a common use case for a GET request handler within a controller, and it's more declarative compared to handling routes in Express.js, which often requires middleware for such functionality.
- Utilize create to add new songs to the array. This aligns with Nest.js's design philosophy of structuring code around well-defined modules and services, as opposed to Express's more flexible but less structured approach.

Inject the Service into the Controller

Use the SongsService within the SongsController. In Nest.js, this demonstrates the Dependency Injection (DI) system, which is a fundamental principle of the framework for decoupling components. Unlike in Express.js, where middleware and route handling functions often mingle with business logic, Nest.js encourages a more structured, modular approach that aligns well with SOLID principles.

```
export class SongsController {
constructor(private songsService: SongsService) {}
@Post()
create() {
    return this.songsService.create("Animals by Martin Garrix");
}
}
```

Call the the create method from SongService.

Test the GET Songs Route

Run the application and send a request to http://localhost:3000/songs. This API request will return all the songs; if no items are present in the songs array, expect a [] response with a 200 status code. Unlike in Express, where you would manually define the response code and body, Nest.js leverages decorators and dependency injection to streamline API response management.

Test the POST Songs Route

When you send this request POST http://localhost:3000/songs from rest-client. It will create a new song in the songs array.

When you send the get songs request again you will get the songs array in the response.

["Animals by Martin Garrix"];

Chapter 3 Middlewares / Exception Filters and Pipes

What is Middleware in Nest.js
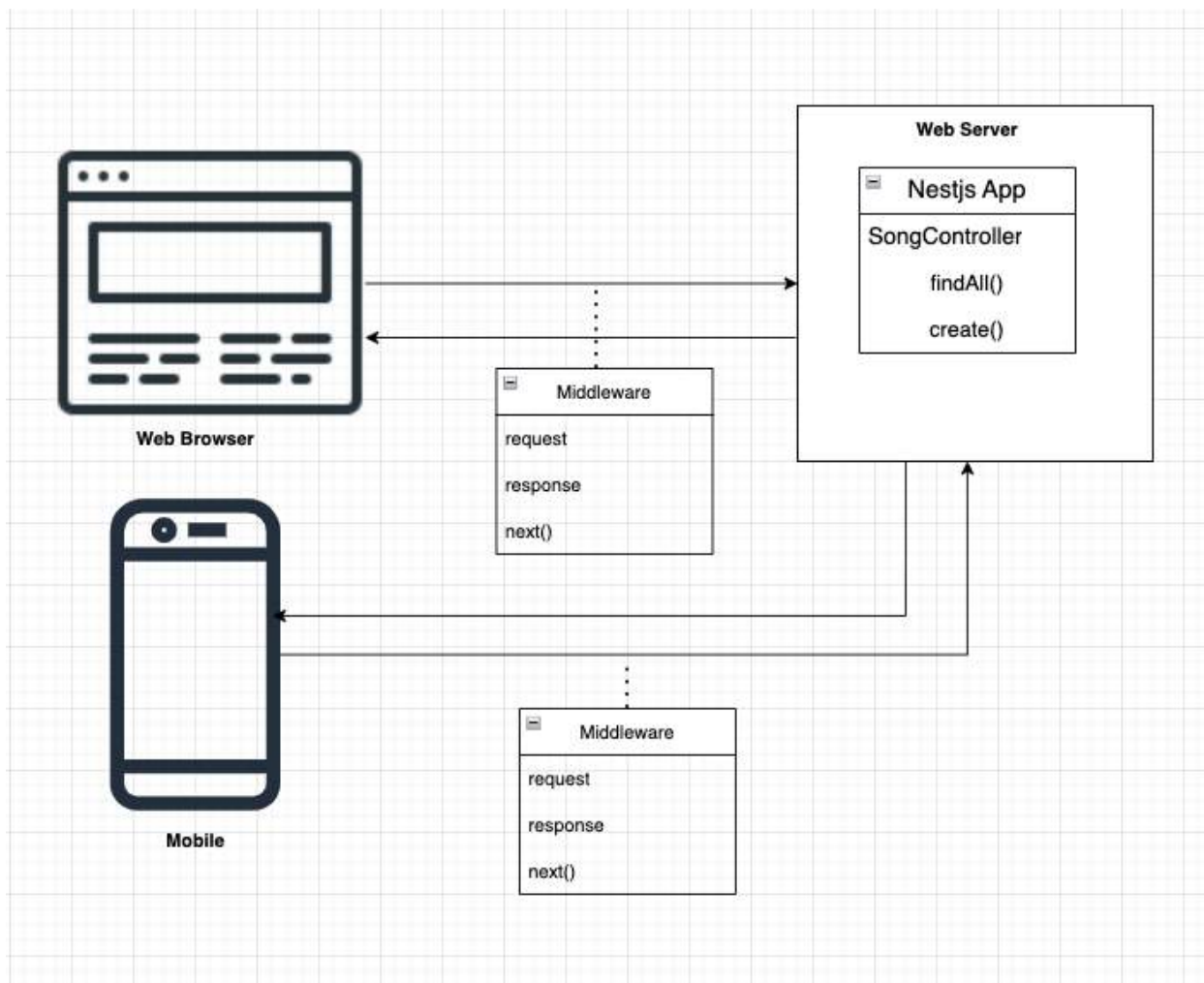
Execute a middleware function before running the route handler; for example, run it before the findAll method in SongsController. In contrast to frameworks like Express, Nest.js middleware offers a more organized and modular approach, closely aligning with object-oriented programming and functional programming paradigms.

The middleware will have access to req, res, and the next function, allowing customization of the request object. This is similar to Express, but Nest.js provides a more robust and scalable architecture for building complex applications.

What middleware can do

| Execute any code | Changes to req and res | End the req-res cycle |

| Call the next middleware |

Execute any code within middleware

In Nest.js, middleware is similar to the Express.js middleware but is more class-based and modular, fitting well within Nest's strong modular architecture. Unlike in Express, where middleware can sometimes become unmanageable in large applications, Nest.js provides a more structured way to handle middleware.

Modify the request (`req`) object

In traditional Express.js, this is often done directly within the middleware function. In Nest.js, however, you can lean more on Dependency Injection (DI) and modularity to make these changes in a more organized fashion.

End the response cycle

Just like in Express, middleware in Nest.js can terminate the request-response cycle. However, Nest.js middleware leverages async/await and decorators, offering a more modern approach and cleaner syntax for handling such operations.

Call the next middleware in the stack.

Both in Nest.js and Express, middleware can pass control to the next middleware function in the stack using the `next()` function. However, Nest.js brings type safety and DI into the picture, making it easier to build robust and maintainable applications.

Logger Middleware
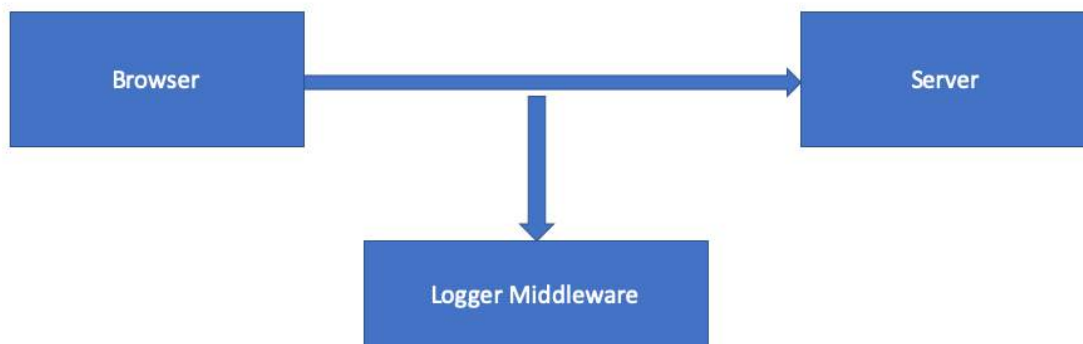
Send the request to the server via a browser. In the Nest.js application, execute the logger middleware before running the request handler. This architecture follows Nest.js's modular approach, wherein middleware-like logging functions can be

organized and re-used across different modules more effectively than in a framework like Express, which lacks such a built-in modular system.

Logger systems are essential for tracking activity, diagnosing issues, and understanding the behavior of an application. Unlike traditional setups where logging might be an afterthought, Nest.js allows the integration of sophisticated logging mechanisms due to its modular and extensible nature. This is in contrast to less opinionated frameworks like Express, where logging is often implemented via external middleware without any standard structure.

Creating Logger Middleware
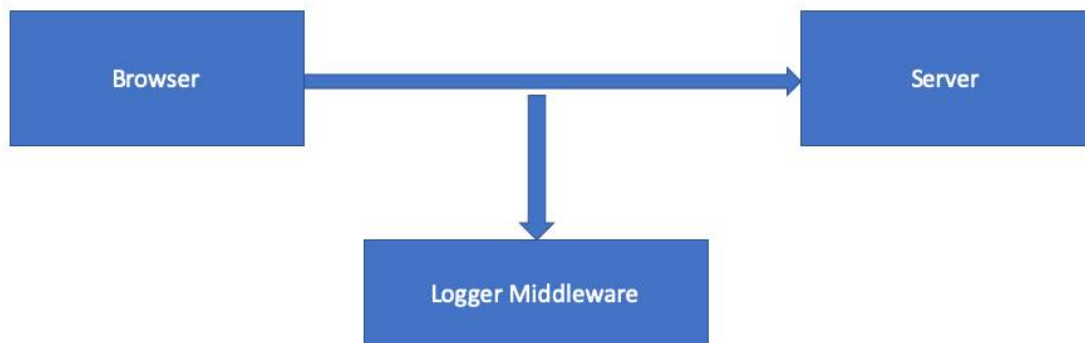
We are going to use nest cli to generate the `LoggerMiddleware`. Please create the `common` folder inside the `src` directory and also create a `middleware` folder inside the `common` directory. This could be the directory structure `src/common/middleware/`

```
nest g  module common/middleware/logger  --no-spec  --no-flat
```

- `--no-spec` means I don't want the testing file

- `--no-flat` means do not create the new directory with logger middleware. You have to create the logger.middleware.ts file

You will the `logger.middleware.ts` inside the middleware folder.

```
@Injectable()

export class LoggerMiddleware implements NestMiddleware {

use(req: any, res: any, next: () => void) {

console.log("Request ....", new Date().toDateString());

next();

}

}
```

Create a LoggerMiddleware class that implements NestMiddleware. Ensure you write the implementation for the `use` method. Customize the `req` object as needed; for example, you could log the current date.

Apply middleware in the AppModule.

```
export class AppModule implements NestModule {

configure(consumer: MiddlewareConsumer) {

// consumer.apply(LoggerMiddleware).forRoutes('songs'); // option no 1

// consumer

// .apply(LoggerMiddleware)
```

```
// .forRoutes({ path:  'songs',  method:  RequestMethod.POST }); //option no 2


  consumer.apply(LoggerMiddleware).forRoutes(SongsController); //option no 3

  }

}
```

Choose from the three provided options to apply the middleware. In the last option, implement the LoggerMiddleware for the SongsController routes.

Test the middleware

Start the application using `npm run start:dev`.

When sending a request to any `songs` API route, ensure it displays the current date.

Send a GET request to `localhost:3000/songs`.

Handling Exceptions

If an error occurs in the code, handling it becomes crucial. NestJS offers built-in HTTP exception handling that streamlines the process of sending informative, well-structured responses to the client, a feature that sets it apart from frameworks like Express, which require additional middleware for similar functionality.

Throwing an exception in the `SongsService findAll` method can be accomplished with ease. In NestJS, using `throw new HttpException('Description', HttpStatus.STATUS_CODE)` allows for both custom messages and HTTP status codes, providing a more developer-friendly and robust error-handling mechanism than some other backend frameworks like Flask, where exceptions often require more manual setup.

```
findAll() {

// fetch the songs from the db

// Errors come while fetching the data from DB

throw  new  Error('Error in Db while fetching record');

return  this.songs;

}
```

A fake error message has been sent to simulate an issue while fetching data from the database. Sending a request to fetch all songs from `http://localhost:3000/songs` will result in an error message accompanied by a 500 status code.

Handling Exception with Try/Catch

```
// SongsContoller.ts
@Get()
findAll() {
  try {
  return  this.songsService.findAll();
 }
  catch (e) {
  throw  new  HttpException(
  'server error',
  HttpStatus.INTERNAL_SERVER_ERROR,{ cause:  e },
```

```
  );

}
```

- Exception handling is possible using the `try/catch` block, a standard programming construct. Within the scope of NestJS, this is more structured and type-safe compared to Express, where error handling often relies on middleware functions and lacks native TypeScript support.
- Logging messages in the `catch` block serves as a best practice for debugging and auditing purposes. In NestJS, logging can be more streamlined thanks to its modular architecture and built-in Logger class, unlike Express, where a third-party library like `winston` or `morgan` is generally needed for robust logging.
- Sending specific HTTP status codes along with error messages is facilitated in NestJS through its built-in `HttpException` class. This provides more granularity and control over error responses compared to Express, which often requires additional libraries like `http-errors` for similar functionality.
- Opting for a 500 Internal Server Error is a choice that indicates a server-side issue. As a best practice, principal engineers might choose to map exceptions to specific HTTP status codes based on the nature of the error, a feature that is natively supported and simplified in NestJS compared to Express.

Pipes

- Transform Param using ParseInt is a feature in NestJS that allows for easy type conversion. In contrast to frameworks like Express, which lack built-in parameter transformation, NestJS's use of pipes offers a more automated and native approach to type coercion, adhering to best practices for robust type checking, which a principal engineer would highly value.
- There are two primary use cases for pipes: transforming the value and validating the input parameters. While Express requires middleware or additional libraries like `express-validator` to achieve similar functionality, NestJS pipes integrate seamlessly into the framework's ecosystem, offering a

more elegant, maintainable solution for both value transformation and input validation—aligned with the architectural best practices that a principal engineer would implement.

```typescript
// SongsContoller.ts

@Get(':id')

findOne(

    @Param(

        'id',

        new ParseIntPipe({
            errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE
        }),

    ),

    id: number,

) {

    return `fetch song on the based on id ${typeof  id}`;

}
```

- Dynamic parameters can be captured using the `@Param` decorator, where the argument name needs to be specified. In contrast to Express, where request parameters are extracted using `req.params`, NestJS provides a more declarative and type-safe way to do so, adhering to best practices by enforcing a stricter type system.
- The `id` parameter is of type string by default. Utilizing `ParseIntPipe` will automatically convert this string value to a number. Unlike in Express, which would require manual type conversion, NestJS's use of pipes allows for automatic type transformation, making the code more robust and maintainable, a practice any principal engineer would appreciate.

- Sending a request to `http://localhost:3000/songs/1` will result in logging the type of `id` as a number. This showcases NestJS's ability to utilize pipes for transformation tasks, an area where it holds an edge over frameworks like Express, which necessitate separate middleware for such operations.
- The error status code can also be provided to `ParseIntPipe`. Should a string value be provided, an error will be generated. This approach lends itself to better error handling in NestJS compared to the more manual error-checking methods required in Express.
- Sending a request to `http://localhost:3000/songs/abc` will produce an error message stating "not acceptable." In frameworks like Express, validation logic for handling such errors would generally need to be written explicitly, whereas NestJS allows for more configurable and built-in validation mechanisms. This feature aligns with best practices for maintainability and scalability.

Validate request parameters with the class validator

· To validate request parameters, class-validator is often used in NestJS. Installing two required packages initiates this feature, making validation an integral part of the request-handling process, unlike in Express where validation logic might be manually coded or pulled in via additional middleware.

· Utilizing class-validator in NestJS allows for declarative validation rules in DTO (Data Transfer Object) classes using various decorators such as `@IsString()` or `@IsNotEmpty()`. This approach promotes reusability and maintainability of validation logic, aligning with best practices for scalable application architecture, whereas in Express, separate validation libraries like `validator` or `express-validator` are often needed.

```
"class-transformer": "^0.5.1",
```

```
"class-validator": "^0.14.0"
```

Global Scope pipes

- The next action involves binding the `ValidationPipe` from the `@nestjs/common` package. This feature provides an advantage over frameworks like Express, where validation often necessitates additional libraries or custom middleware. In NestJS, pipes offer multiple scopes for validation: parameter-scoped, method-scoped, controller-scoped, or global-scoped, lending greater flexibility and modularity to the application.

- Opting for a global scope requires registration in the `main.ts` file. This is a best practice for ensuring application-wide consistency in validation, as it minimizes the chances of missing validation logic in any part of the application. Unlike Express, which would require a global middleware function for similar functionality, NestJS makes it straightforward to set global validation rules.

src/main.ts

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

// ...

app.useGlobalPipes(new ValidationPipe());
```

Create CreateSongDTO

A DTO (Data Transfer Object) serves as a blueprint for how data will be sent over the network. While both TypeScript interfaces and simple classes can define the DTO schema, classes are recommended in this context. In contrast to Express, where DTO and validation often require additional libraries or middleware, NestJS provides a more integrated approach through its decorator-based validation techniques. As a best

practice, storing DTOs in a dedicated directory ensures that the application adheres to the principle of separation of concerns, facilitating easier maintenance and future scaling.

To instantiate the DTO, a new class should be created inside the `src/songs/dto/create-song.dto.ts` file. This is a more structured approach than in frameworks like Express, where the schema and validation may be mixed with route handlers or middleware. Having a dedicated DTO file is conducive to more modular and maintainable code.

```ts
import {

IsArray,

IsDate,

IsDateString,

IsInt,

IsMilitaryTime,

IsNotEmpty,

IsString,

} from "class-validator";


export class CreateSongDto {

@IsString()

@IsNotEmpty()

readonly title;


@IsNotEmpty()
```

```
@IsArray()

@IsString({ each: true })

readonly artists;


@IsDateString()

@IsNotEmpty()

readonly releasedDate: Date;


@IsMilitaryTime()

@IsNotEmpty()

readonly duration: Date;

}
```

Four fields—title, artists, releasedDate, duration—are present.
Class-validator enables the addition of decorator-based validations. The
isDateString() function is employed to validate the date, while isMilitaryTime()
is used for time validation in the HH:MM format.

Apply CreateSongDTO as Body decorator

You have to update the create method inside the src/songs/songs.controller.ts

```
@Post()

create(@Body() createSongDTO: CreateSongDto) {

const results = this.songsService.create(createSongDTO);

return results;
```

```
        }
```

Test the Application

To initiate the application, execute the command `npm run start:dev` and proceed to send the 'create song' request. Unlike in Express, where nodemon or a similar package would be separately installed for hot-reloading, NestJS includes this feature by default with the `start:dev` script, providing a more out-of-the-box development experience. As a best practice, separating the 'create song' logic into a dedicated service method ensures cleaner, more maintainable code.

```
POST  http://localhost:3001/songs

Content-Type:  application/json




{

"title":  "Lasting Lover",

"artists": ["Siagla"],

"releasedDate"  :  "2022-07-29 00:00:00",

"duration"  :"02:34"

}
```

Chapter 4 Dependency Injection

Custom Providers

- Constructor-based dependency injection is utilized for injecting instances, often service providers like `SongsService`, into classes using `constructor(private songsService: SongService)`. This approach is in

contrast to frameworks like Express, where dependency injection is not natively supported and often requires third-party libraries for similar functionality. As a best practice, constructor-based injection is preferred for its explicitness and ease of testing.

- When the Nest IoC (Inversion of Control) container creates an instance of `SongsController`, it scans for any dependencies, such as `SongsService`. Upon identifying the dependency, Nest instantiates `SongsService`, caches it, and returns the instance, reusing the existing one if already cached. This differs from Express, which generally lacks a built-in IoC container, requiring manual instantiation or third-party solutions. Leveraging caching for service instances can lead to performance optimization.

Various techniques exist for utilizing providers in NestJS:

1. Standard Providers: These are classes that get instantiated by the NestJS dependency injection system. Unlike in Express, where dependency injection must often be implemented manually or through third-party libraries, NestJS provides this feature natively. As a best practice, leverage standard providers for services that require instantiation.
2. Value Providers: These are hard-coded values or configurations injected into other classes. This feature can replace the need for environment variables or configuration files, which in frameworks like Express, would typically be managed by separate packages. Using value providers for constants and configuration settings contributes to code maintainability.
3. Non-Class based Provider Tokens: These are custom tokens that can be used to inject values or services. Unlike in frameworks like Django, which relies more on function-based views and doesn't have a direct equivalent, NestJS allows greater flexibility in dependency injection. Use custom tokens judiciously to avoid overly complex dependency graphs.
4. Class Providers: useClass: These allow for the substitution of one class provider with another, providing polymorphism. This is distinct from Express, where such abstraction might require more manual factory patterns. Employing `useClass` is beneficial for achieving code reusability and flexibility.
5. Factory Providers: useFactory: These allow for conditional instantiation of a class, something not natively supported in frameworks like Flask. Use factory

providers when the provision of a class or value is contingent upon runtime conditions.

6. Non-service Providers: These are providers that aren't necessarily tied to a service and may provide utility functions, for example. This granularity is generally not available in basic Express setups and would usually require additional modules or utilities. Incorporating non-service providers can aid in keeping the codebase DRY (Don't Repeat Yourself).

Exploration of all six techniques will take place, each elucidated through an example.

We are going to play around with all these 6 techniques. I will explain each technique with the help of an example

Standard Providers

```
@Module({

    controllers: [SongsController],

    providers: [SongsService],

})
```

This is the standard provider technique, you have used in our application. You can also convert the above syntax into this syntax

```
@Module({

    controllers: [SongsController],

    providers: [{

            provide: SongsService,

            useClass: SongsService,

        },
```

```
    ];

})
```

Value Provider

```
const mockSongsService = {

    findAll() {

        return [{
            id: 1,
            title: "Lasting  lover"
        }];

    },

};

@Module({

    controllers: [SongsController],

    providers: [

        SongsService,

        {

            provide: SongsService,

            useValue: mockSongsService,

        },

    ],

})

export class SongsModule {}
```

The useValue syntax is useful for injecting a constant value, putting an external library into the Nest container, or replacing a real implementation with a mock object. Let's say you'd like to force Nest to use a mock `SongsService` for testing purposes.

When you send `GET` request to `localhost:3000/songs` it will run the `findAll()` method from mockSongsService instead of origianl `SongsService`

```ts
// src/common/constants/connection.ts

export const connection: Connection = {

    CONNECTION_STRING: "CONNECTION_STRING",

    DB: "MYSQL",

    DBNAME: "TEST",

};

export type Connection = {

    CONNECTION_STRING: string;

    DB: string;

    DBNAME: string;

};
```

Providers

```ts
// src/common/constants/connection.ts

export const connection: Connection = {

    CONNECTION_STRING: "CONNECTION_STRING",

    DB: "MYSQL",
```

```
    DBNAME: "TEST",

};

export type Connection = {

    CONNECTION_STRING: string;

    DB: string;

    DBNAME: string;

};
```

To inject an object as a dependency, `useValue` can be utilized. This feature distinguishes NestJS from frameworks like Express, where manual constructor-based injection is often the norm. In the case at hand, a new file named `connection.ts` has been created, containing a connection object with `CONNECTION_STRING, DB AND DBNAME`. As a best practice, keeping connection configurations in separate files and importing them as needed ensures a modular and easily configurable application setup.

```
import {
    connection
} from 'src/common/constants/connection';



@Module({

    controllers: [SongsController],

    providers: [SongsService]

    providers: [

        SongsService,

        // Non class based providers
```

```
      // You can use it to add constant values

      {

          provide: 'CONNECTION',

          useValue: connection,

      },

    ],

})

export class SongsModule {}
```

The connection object can now be injected into any controller or service within the SongsModule. This contrasts with frameworks like Express, where dependency injection isn't native and often requires third-party libraries like 'Awilix' for similar functionality. As a best practice, isolating database connections in dedicated provider files ensures more modular and maintainable code, thereby facilitating easier unit testing and separation of concerns.

```
@Injectable()

export class SongsService {

    constructor(

        @Inject("CONNECTION")

        connection: Connection

    ) {

        console.log("connection string", connection.CONNECTION_STRING);

    }

}
```

I have injected this `connection` object into `SongsService` by using the `@Inject` decorator with the token name `CONNECTION`

Class Providers: useClass

```
@Module({
    providers: [
        AppService,
        {
            provide: DevConfigService,
            useClass: DevConfigService,
        },
    ],
})
export class AppModule {}
```

A `DevConfigService` class has been created and registered as a provider. With this setup, `AppModule` is capable of injecting this provider as a dependency into any class for utilization. In contrast to Express, which typically relies on third-party libraries like `dotenv` for configuration management, NestJS offers a more integrated way to handle configurations through custom service providers. As a best practice, encapsulating configuration logic within a dedicated service class enables easier testing and maintenance, adhering to the principle of Separation of Concerns.

```typescript
// src/common/providers/DevConfigService.ts

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

@Injectable()

export class DevConfigService {

    DBHOST = "localhost";

    getDBHOST() {

        return this.DBHOST;

    }

}

@Injectable()

export class AppService {

    constructor(private devConfigService: DevConfigService) {}

    getHello(): string {

        return `Hello I am learning Nest.js Fundamentals
${this.devConfigService.getDBHOST()}`;

    }

}
```

The `DevConfigService` has been injected into `AppService`, and the `getDBHOST` method from DevConfigService is invoked. Unlike in Express, where dependency injection is not natively supported and might require third-party libraries, NestJS simplifies this with built-in Dependency Injection, leading to more modular and

testable code. As a best practice, environment-specific configurations like database host details should be abstracted away into separate configuration services, ensuring that the code adheres to the Twelve-Factor App methodology.

Non-Service Provider

A provider in NestJS has the flexibility to supply any value, making it a versatile component for dependency injection. The `useFactory` syntax facilitates the dynamic creation of providers, a feature that sets it apart from Express, which often relies on external libraries like `awilix` or manual constructor injection for similar functionality. As a best practice, isolating complex business logic within factory providers can lead to more modular and testable code.

```
const devConfig = {
    port: 3000
};

const proConfig = {
    port: 400
};
```

```
@Module({

    providers: [{

            provide: 'CONFIG',

            useFactory: () => {

                return process.env.NODE_ENV === 'development' ? devConfig :
proConfig;

            },

        },

    ],
```

```
})
export class AppModule
```

Two objects, `devConfig` and `proConfig`, have been created. Dynamic value assignment is possible through the `useFactory` function. Unlike Express, where environment-specific configurations often require separate JSON or JS files, NestJS allows for more streamlined, inline dynamic configuration. As a best practice, isolating configuration logic in dedicated modules or services fosters maintainability and scalability.

```
@Injectable()
export class AppService {
    constructor(
        @Inject("CONFIG")
        private config: {
            port: string
        }
    ) {
        console.log(config);
    }
}
```

I have injected the `CONFIG` provider inside the AppService.

Injection Scopes

Provider scopes in NestJS can be categorized into three main types:

1. **DEFAULT**
   - In this scope, a single instance of the provider is shared across the entire application, a feature similar to Express's singleton services but made explicit in NestJS. When the application requests this provider a second time, NestJS retrieves it from an internal cache, a mechanism that enhances application performance. As a best practice, caching providers to minimize computational overhead is advisable.
2. **REQUEST**
   - Contrary to the DEFAULT scope, a new instance of the provider is created for each incoming request. While this provides better isolation, it's a different model from Express's middleware-based architecture where request-level isolation usually involves function scope. This scope is ideal for scenarios where each request may need a provider with state that shouldn't affect other requests.
3. **TRANSIENT**
   - Transient providers are unique in that they are not shared across different consumers. When a transient provider is injected, the consumer receives a new, dedicated instance, unlike in Express where this level of injection granularity is not native and often must be manually managed. Employing transient providers is good practice when state isolation between consumers is crucial.

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

export class SongsService {}


@Controller({

    path: "songs",

    scope: Scope.REQUEST,
```

```
})

export class SongsController {}
```

A new instance of SongsController is created for every incoming request, offering a more stateless architecture. In contrast, frameworks like Express use a singleton pattern for controllers, thus reusing the same instance for all requests, which can introduce state-related issues. Adopting a stateless architecture is a best practice for improving scalability and maintainability.

Debugging dependencies may present challenges, but understanding the usage of different scopes is essential. Unlike Express, which lacks a built-in DI (Dependency Injection) system, NestJS has various scopes for dependency injection, making it more flexible for complex use cases. Utilizing the correct scope for dependencies is considered a best practice, as it improves performance and the overall architecture.

Chapter 5 Connecting Nest.js application with TypeORM and Postgres

In this lesson, the application will be connected to a database, utilizing TypeORM as the bridge between object-oriented code and SQL queries in a Nest.js context. As a best practice, isolating database connection configurations into environment variables ensures both security and easier environment management. Learning how to connect Nest.js applications to a Postgres database provides a comprehensive understanding of backend architecture, an area where Nest.js offers more out of the-box features compared to Express.

Install Dependencies

```
"@nestjs/typeorm": "^9.0.1",
```

```
"pg":  "^8.10.0",

"typeorm":  "^0.3.15"
```

Should TypeScript typings for TypeORM be absent, installation as a devDependency is possible.

```
"devDependencies":  {

"@types/node":  "^18.11.18"

}
```

First, installation of the required dependencies in the application is necessary. These packages can be copied into the package.json file, followed by running the command npm install.

Import TypeORM Module to App Module

AppModule is our root module, we have to configure the TypeORM module here

```
imports: [

        TypeOrmModule.forRoot({

                type: 'postgres',

                host: 'localhost',

                port: 5432,

                username: 'your_username',

                password: 'your_db_password',

                database: 'n-test',
```

```
        entities: [],

        synchronize: true,

    }),
]
```

- Importing the TypeORM module into AppModule requires calling the forRoot method. NestJS streamlines manual configuration for integrating with databases by providing built-in methods like forRoot that make it easier to set up a database connection. As a best practice, separating database connection logic into a dedicated configuration file is advised for easier management and scalability.
- The setting synchronize: true is risky for production as it can result in data loss. As a good practice, maintaining separate configurations for development and production environments ensures database integrity.
- After creating a new entity, it must be added to a specified array. NestJS promotes organization by having entities declared in a central location. A best practice is to maintain an index or a barrel file that exports all entities, thus making it easier to manage them.
- The forRoot() method accepts all configuration properties exposed by the DataSource constructor in the TypeORM package. As a good practice, encapsulating these settings in environment variables offers both security and ease of management.
- Providing a username and DB password is essential, and no database with the name n-test will exist by default. In NestJS, tools like PG-ADMIN allow for a graphical interface to interact with the Postgres Database. A good practice is to use a secure vault or environment variables for storing sensitive information like usernames and passwords.

Test the DB Connection

```
export class AppModule implements NestModule {
```

```
    constructor(private dataSource: DataSource) {

        console.log(dataSource.driver.database);

    }

 }
```

Testing the database connection can be accomplished by injecting the datasource class into AppModule, followed by logging the name of the database. Upon running the application, the database name will appear in the logs, signifying a successful connection between the NestJS application and the database.

Utilizing dependency injection for the datasource class promotes a modular and easily testable codebase, aligning with industry best practices. Additionally, logging crucial steps, such as successful database connections, aids in debugging and monitoring, also considered a best practice

Create an Entity

Entity is a class that maps to a database table, or to a collection when using MongoDB. In the context of NestJS, which often pairs with Object-Relational Mapping tools like TypeORM, an entity functions as the data model for the application, interfacing directly with the database schema. This exemplifies the software engineering principle of data source abstraction, allowing for easier swapping of databases or migration. As a best practice, encapsulating database interactions within repository classes streamlines code maintenance and enhances testability.

```
import { Column,  Entity,  PrimaryGeneratedColumn } from  "typeorm"

@Entity("songs")

export  class  Song {
```

```
@PrimaryGeneratedColumn()

id:  number;

@Column()

title:  string;

@Column("varchar", { array:  true })

artists:  string[];

@Column({ type:  "date" })

releasedDate:  Date;

@Column({ type:  "time" })

duration:  Date;

@Column({ type:  "text" })

lyrics:  string;

}
```

In the class Entity: `@Entity('songs')` indicates that `songs` is the name of the database table. `@PrimaryGeneratedColumn()` serves to automatically increment the primary key, a feature that may be specified either here or directly within the database. `@Column()` defines a column, allowing specification of various data types such as date, varchar, and time. This column definition offers precise mapping, crucial for efficient data querying and comprehensive schema definition.

```
@Column('varchar', { array:  true })

artists: string[];
```

An array of artists exists, serving as a data structure to hold multiple artist entities. To create this array field in a Postgres database, metadata is added to the `@Column` decorator in the NestJS application. As a best practice, strongly typing this array—for instance, as `string[]`—can improve code maintainability and reduce potential runtime errors. In addition, leveraging Postgres' native array data type within the `@Column` decorator provides optimizations for array operations and queries.

```
@Column({ type: 'date' })

releasedDate: Date;
```

The `date` type is specified for the release date, ensuring that no time-related properties are added to the `releasedDate` field. In NestJS, using class-validator's `@IsDate()` decorator on the `releasedDate` property would further enforce type safety and validation, adhering to best practices for robust, production-ready code

```
@Column({ type: 'time' })

duration: Date;
```

The `duration` field is configured with the date type, which might imply that it should hold date-related data. However, despite this type declaration, the field will exclusively contain time information because the specific type has been set to `time`. As a best practice in NestJS, explicitly naming the database column and its type using the `@Column` decorator can avoid ambiguity and make the code more maintainable. Additionally, using type guards or DTOs (Data Transfer Objects) with class-validator can ensure that the data in this field adheres to the intended time format, enhancing data integrity.

```
@Column({ type: 'text' })

lyrics: string;
```

To accommodate a long string or text, the `text` type in Postgres is appropriate. This stands in contrast to shorter text, for which the varchar type would be suitable.

Update CreateSongDTO

```
@IsString()

@IsOptional()

readonly lyrics: string;
```

A new field, lyrics, has been added to store the lyrics in the database. This field was not previously included in the `CreateSongDTO` file.

Register the Entity in AppModule

To integrate the Song Entity with the application, include it in the AppModule by updating the TypeORM module. Specifically, add the Song Entity to the `entities` array within the `forRoot` method. NestJS allows seamless integration of entities via its modular structure. As a best practice, organizing the `entities` in a dedicated configuration file can enhance modularity and ease of management.

```
entities: [Song],
```

Test the Application

Upon running the application, the `songs` table appears in the `n-test` database. These fields materialize automatically, an outcome facilitated by NestJS's integration with TypeORM, which handles database migrations and schema synchronization. A best practice involves leveraging NestJS's built-in Dependency Injection system for all database interactions, ensuring a modular and testable codebase.

Create and Fetch Records from DB

Repository Pattern

TypeORM supports the repository design pattern, so each entity has its own repository. In NestJS, this adherence to the repository pattern facilitates cleaner, more modular code by separating the database logic from business logic, aligning with software engineering best practices.

These repositories can be obtained from the database data source. In a NestJS application, you usually inject these repositories into your services or controllers via Dependency Injection, enabling direct interaction with the database through methods like `find`, `save`, or `delete`.

```
@Module({

imports: [TypeOrmModule.forFeature([Song])],

controllers: [SongsController],

providers: [SongsService],

})
```

To use the SongRepository we need to import the TypeORM module into the Songs module. This module uses the `forFeature()` method to define which repositories are registered in the current scope. With that in place, we can inject the `SongRepository` into the SongService using the `@InjectRepository()` decorator:

```
import { Song } from "./song.entity";

import { InjectRepository } from "@nestjs/typeorm";

export class SongsService {

constructor(
```

```
@InjectRepository(Song)

private  songRepository:  Repository<Song>

) {}

}
```

`SongRepository` provides `CRUD` methods to create, delete, update, and fetch records from the Songs table. In a NestJS application, this specialized repository is responsible for handling all operations related to the Songs entity, abstracting the database interactions and thus adhering to the repository best practice.

Create Record

Now, you should implement the create song method. This time, there is no need to add a record in the local db array. Save the new song in the database by using the `songRepository.save()` method

```
async create(songDTO: CreateSongDto): Promise<Song> {

const song =  new  Song();

song.title = songDTO.title;

song.artists = songDTO.artists;

song.duration = songDTO.duration;

song.lyrics = songDTO.lyrics;

song.releasedDate = songDTO.releasedDate;


return await this.songRepository.save(song);

// method
```

```
  }
```

Instantiate a new song instance from the Song Entity. Set the fields of the songs table using the DTO object, and finally, return a promise containing the song object from the create method.

A database promise is essential for handling asynchronous operations when interacting with a database. In a web application, tasks like querying, updating, or deleting records are not instantaneous and can take an undefined amount of time to complete. Utilizing promises allows your application to continue executing other tasks while waiting for the database operation to resolve, thereby improving performance and user experience.

In the context of NestJS and TypeORM, methods like save(), find(), or delete() often return promises. By defining a return type like Promise<Song[]> in your service or controller methods, you make it explicit that the function is asynchronous and will return data at a future point in time. This is particularly beneficial for type-checking and for setting the expectations for the developers who will consume these methods.

songRepository provides the save method to save the record in a database table. You have to provide the instance of the entity which entity record you want to save.

```
// songs.controller.ts

@Post()
create(@Body() createSongDTO: CreateSongDto): Promise<Song> {

  return this.songsService.create(createSongDTO);

}
```

Add the return type for the create method, which is a promise containing the Song entity.

Find All Records

```ts
// songs.service.ts

findAll(): Promise<Song[]> {

  return this.songRepository.find();

}
```

Use the `find1 method in the repository to fetch all records from the database table.

```ts
//songs.controller.ts

findAll(): Promise<Song[]> {

  return this.songsService.findAll();

}
```

Update the return type for the `findAll()` method in `songs.controller.ts`. Use `Promise<Song[]>` because `songService.findAll` will return an array of songs.

Find Record by ID

```ts
//songs.service.ts

findOne(id: number): Promise<Song> {

  return this.songRepository.findOneBy({ id });

}
```

You can find the record by id by using `findOneBy()` from the songsRepository.

```ts
// songs.controller.ts

@Get(':id')

findOne(@Param('id', ParseIntPipe) id: number): Promise<Song> {

  return this.songsService.findOne(id);

}
```

Delete Record

```ts
// songs.service.ts

async remove(id: number): Promise<void> {

  await this.songRepository.delete(id);

}
```

`songRepository` provides the `delete` method to delete the record based on `id`.

```ts
// songs.controller.ts

@Delete(':id')

delete(@Param('id', ParseIntPipe) id: number): Promise<void> {

  return this.songsService.remove(id);

}
```

Update Record

```ts
//songs.service.ts

import { Repository,  UpdateResult } from  'typeorm';

import { UpdateSongDto } from  './dto/update-song.dto';
```

```
  update(id: number, recordToUpdate: UpdateSongDto): Promise<UpdateResult> {

    return this.songRepository.update(id, recordToUpdate);

}
```

Create a new `update` method in `songs.service.ts`. The first argument should be the `id`, and the second argument should be `recordToUpdate`. Ensure the type of `recordToUpdate` is a DTO object.

Create another DTO object to update the record.

```ts
// src/songs/dto/update-song.dto.ts

import {

    IsArray,

    IsDateString,

    IsMilitaryTime,

    IsNotEmpty,

    IsOptional,

    IsString,

} from "class-validator";


export class UpdateSongDto {

    @IsString()

    @IsOptional()

    readonly title;
```

```
    @IsOptional()

    @IsArray()

    @IsString({
        each: true
    })

    readonly artists;


    @IsDateString()

    @IsOptional()

    readonly releasedDate: Date;


    @IsMilitaryTime()

    @IsOptional()

    readonly duration: Date;


    @IsString()

    @IsOptional()

    readonly lyrics: string;

}
```

Make all these fields optional, as it depends on the user which record they want to update. Use the @IsOptional() decorator for each field to indicate this.

```ts
// songs.controller.ts

@Put(':id')

update(

    @Param('id', ParseIntPipe) id: number,

    @Body() updateSongDTO: UpdateSongDto,

): Promise < UpdateResult > {

    return this.songsService.update(id, updateSongDTO);

}
```

Test the Application

```
PUT   http://localhost:3001/songs/3

Content-Type:  application/json



{

    "title": "You for Me 3",

    "artists": ["Siagla", "Yan", "Ny"],

    "releasedDate": "2022-09-30",

    "duration": "02:45",

    "lyrics": "Sby, you're my adrenaline. Brought out this other side of me You
don't even know Controlling my whole anatomy, oh Fingers are holding you right at
the edge You're slipping out of my hands Keeping my secrets all up in my head I'm
scared that you won't want me back, oh I dance to every song like it's about ya I
drink 'til I kiss someone who looks like ya I wish that I was honest when I had you
I shoulda told you that I wanted you for me I dance to every song like it's about
```

```
 ya I drink 'til I kiss someone who looks like ya"

 }
```

Now you can test the application by executing this command `npm run start:dev`.
You have to send a request to update the record.

Pagination

Install Pagination Package

You are going to use nestjs-typeorm-paginate to implement pagination. The
nestjs-typeorm-paginate package offers several features that make it advantageous
over implementing pagination manually:

- Type Safety: Being designed for TypeORM and Nest.js, it ensures type-safe
  queries and pagination object returns, reducing the risk of runtime errors.
- Seamless Integration: It's optimized for Nest.js and TypeORM, making
  integration simple and straightforward without needing to modify existing code
  significantly.
- Metadata-based: The package automatically computes pagination metadata like
  total items, pages, current page number, etc., saving you the time to calculate
  these manually.
- Customizable: It allows for detailed customization like specifying relations to
  be loaded, limiting fields, and more, enabling more advanced query operations.

```
 "nestjs-typeorm-paginate": "^4.0.3",
```

Add this package entry in `package.json` file and run `npm install`

Create new Paginate method

Create a new paginate method in `songs.service.ts`

```typescript
import {

paginate,

Pagination,

IPaginationOptions,

} from  'nestjs-typeorm-paginate';


async  paginate(options: IPaginationOptions): Promise<Pagination<Song>> {

  // Adding query builder

  // If you need to add query builder you can add it here

  return paginate<Song>(this.songRepository, options);

}
```

Refactor findAll method in controller

```typescript
// songs.controller.ts

findAll(

    @Query('page', new DefaultValuePipe(1), ParseIntPipe)

    page: number = 1,

    @Query('limit', new DefaultValuePipe(10), ParseIntPipe)

    limit: number = 10,
```

```
    ): Promise < Pagination < Song >> {

        limit = limit > 100 ? 100 : limit;

        return this.songsService.paginate({

            page,

            limit,

        });

    }
```

Refactor the `findAll` method in `songs.controller.ts` to obtain `page` and `limit` from the request query parameters. Use the `@Query` decorator to capture these query parameters.

Set default values for `page` and `limit`, and then call the `paginate` method from `songService`, passing in the `page` and `limit` values.

Test the Application

Now test the application by sending request `http://localhost:3001/songs/?page=2&limit=2`

Specify the page and limit value on the based on your situation/usecase

This is the response:

```
    {

        "items": [

            {

                "id": 10,
```

"title": "A",

"artists": ["New Artists"],

"releasedDate": "2022-09-29",

"duration": "05:00:00",

"lyrics": "by, you're my adrenaline. Brought out this other side of me You don't even know Controlling my whole anatomy, oh Fingers are holding you right at the edge You're slipping out of my hands Keeping my secrets all up in my head I'm scared that you won't want me back, oh I dance to every song like it's about ya I drink 'til I kiss someone who looks like ya I wish that I was honest when I had you I shoulda told you that I wanted you for me I dance to every song like it's about ya I drink 'til I kiss someone who looks like ya"

},

{

"id": 5,

"title": "Lasting Lover",

"artists": ["Siagla"],

"releasedDate": "2022-09-29",

"duration": "02:34:00",

"lyrics": "by, you're my adrenaline. Brought out this other side of me You don't even know Controlling my

```
            whole anatomy, oh Fingers are holding you right at the edge
            You're slipping out of my hands Keeping my secrets all up in
            my head I'm scared that you won't want me back, oh I dance to
            every song like it's about ya I drink 'til I kiss someone who
            looks like ya I wish that I was honest when I had you I
            shoulda told you that I wanted you for me I dance to every
            song like it's about ya I drink 'til I kiss someone who looks
            like ya"

        }

    ],

    "meta": {

        "totalItems": 7,

        "itemCount": 2,

        "itemsPerPage": 2,

        "totalPages": 4,

        "currentPage": 2

    }

}
```

Apply Sorting/Orderby

To apply sorting or filtering, utilize the QueryBuilder feature. Consult the TypeORM documentation for guidance on using QueryBuilder.

```
async  paginate(options: IPaginationOptions): Promise<Pagination<Song>> {

  const queryBuilder = this.songRepository.createQueryBuilder('c');

  queryBuilder.orderBy('c.releasedDate', 'DESC');

  return paginate<Song>(queryBuilder, options);

}
```

To sort records, either manually add the "orderBy" field or retrieve it from the query parameters, depending on your specific use case. Test the application to view records sorted by "releasedDate" in descending order.

Chapter 6 Relations

One to One Relation

One-to-one is a relational database design pattern where entity A contains only one instance of entity B, and vice versa, ensuring a bijective mapping. In a Nest.js application with TypeORM, this might manifest as a user entity having a one-to-one relation with an artist entity, meaning a user can become an artist, and an artist can have only a single user profile. This design is governed by the Single Responsibility Principle, as each entity is responsible for a distinct set of attributes and behaviors, thereby simplifying database management and application logic.

Create Artist Entity

```
import {
    User
} from "src/users/user.entity";
```

```
import {
    Entity,
    JoinColumn,
    OneToOne,
    PrimaryGeneratedColumn
} from "typeorm";


@Entity("artists")

export class Artist {

    @PrimaryGeneratedColumn()

    id: number;

}
```

Define the `Artist` entity starting with one field, using the `@Entity` decorator to map it to a database table. Extend the entity by adding more columns or fields as needed, guided by the application's requirements and domain model. This flexibility allows you to tailor the entity to fit various use cases.

Create User Entity

```
import {
    Column,
    Entity,
    PrimaryGeneratedColumn
} from "typeorm";


@Entity("users")

export class User {

    @PrimaryGeneratedColumn()
```

```
    id: number;


    @Column()

    firstName: string;


    @Column()

    lastName: string;


    @Column()

    email: string;


    @Column()

    password: string;

}
```

Create a `User` entity in the application, including fields for `firstName`, `lastName`, `email`, and `password`. Use appropriate data types and decorators to define these fields.

Add One to One Relation

```
@Entity("artists")

export class Artist {

    // A user can register as an artist
```

```
    // Each artist will have only a user profile

    @OneToOne(() => User)

    @JoinColumn()

    user: User;

}
```

Utilize TypeORM's @OneToOne decorator to specify the target relation type, which in this case is User. Include the @JoinColumn to ensure that the Artist entity possesses the relation ID or foreign key; this will result in the Artist table having userId as a foreign key.

Register User and Artist in AppModule

```
entities: [Song, Artist, User],
```

Register the newly created entities in the TypeORM module to integrate them into your Nest.js application. This can be done within the AppModule, making it a focal point for configuring these database entities. Following the Dependency Injection design pattern, this allows your application to be extensible and maintainable by centralizing the registration process.

Many to Many Relation

Establish a many-to-many relationship when a single instance of Entity A can be associated with multiple instances of Entity B and vice versa. In a musical platform built with Nest.js, consider that many artists can publish many songs, and similarly, a single song can belong to multiple artists. This relationship exemplifies the concept of high cardinality in database design, ensuring that your data structure can capture complex associations without redundancy.

Add Many to Many Relation in Artist

```
@Entity("artists")
export class Artist {
  @OneToOne(() => User)
  @JoinColumn()
  user: User;

  @ManyToMany(() => Song, (song) => song.artists)
  songs: Song[];
}
```

Add the @ManyToMany relation in the Artist entity. The first argument specifies the target entity, and the second argument represents the inverse side of the relationship.

Add Many to Many Relation in Song

```
@Entity("songs")
export class Song {
  // @Column('varchar', { array: true })
  // artists: string[];

  @ManyToMany(() => Artist, (artist) => artist.songs, { cascade: true })
  @JoinTable({ name: "songs_artists" })
  artists: Artist[];
}
```

Refactor the Artist's column; storing it as an array of strings is unnecessary. Instead, use the @ManyToMany annotation to define a relationship with another entity, specifying the target entity as the first argument and the inverse side as the second argument, found in the songs property within the Artist entity.

For cascading actions, use the cascade: boolean | ("insert" | "update")[] parameter. Setting it to true will automatically manage the related object's insertion and updates. Specific cascade options can also be set as an array.

Include the @JoinTable() annotation, which is mandatory for @ManyToMany relationships. Optionally, rename the JoinTable to create a table named songs_artists in the database. This table will house the primary keys from both the songs and artists tables, functioning as foreign keys.

Here, the @ManyToMany and @JoinTable() annotations are typical of TypeORM, which Nest.js leverages for database operations. The cascade option is an example of the software engineering principle of DRY (Don't Repeat Yourself), as it automates CRUD operations for related entities. Meanwhile, the joined table is a clear manifestation of the database normalization principle, optimizing the data structure.

Refactor CreateSongDTO

```
//@IsString({ each: true })
@IsNumber({}, { each: true })
readonly artists;
```

Expect an array of numbers, not strings, from the network request object when creating a new song. In the API request to create a song, provide the artist IDs. Use these IDs to query all corresponding artists from the database and establish a relationship with the newly created song.

In this context, using artist IDs instead of strings offers type safety and ensures data integrity, which is a cornerstone of reliable software design. The architecture of expecting IDs and establishing relations through them is often part of the RESTful API design pattern, which aims for stateless client-server communication. Nest.js makes it convenient to handle these types of operations through its integration with TypeORM, a powerful Object-Relational Mapping (ORM) library.

Refactor UpdateSongDTO

```
// @IsString({ each: true })
@IsNumber({}, { each: true })
readonly artists;
```

Refactor the `artists` field in the `CreateSongDTO`.

Register the Artist Entity in SongsModule

```
@Module({
  imports: [TypeOrmModule.forFeature([Song, Artist])],
  controllers: [SongsController],
  providers: [SongsService],
})
```

Include the `Artist` entity in `songs.service.ts` to utilize the `ArtistRepository`.

Refactor the Create Song Method

```
@Injectable()
export class SongsService {
  constructor(
    @InjectRepository(Song)
```

```
    private songRepository: Repository<Song>,
    @InjectRepository(Artist)
    private artistRepository: Repository<Artist>,
) {}


async create(songDTO: CreateSongDto): Promise<Song> {
const song = new Song();
song.title = songDTO.title;
song.artists = songDTO.artists;
song.duration = songDTO.duration;
song.lyrics = songDTO.lyrics;
song.releasedDate = songDTO.releasedDate;


const artists = await this.artistRepository.findByIds(songDTO.artists);
song.artists = artists;


return await this.songRepository.save(song);
}
```

Find the `artists` by retrieving the `ids` from the DTO object and set these to the song entity. Save the changes in the songs repository; this will automatically establish a relation in the joined table `songs_artists`.

Test the Application

To test the application, ensure that artist and user records are present in the database. If they are not, manually create these records. For testing purposes, establish relationships between at least two users and two artists.

```
POST http://localhost:3001/songs
Content-Type: application/json
{
"title": "You for me",
"artists": [1,2],
"releasedDate" : "2022-08-29",
"duration" :"02:34",
"lyrics": "by, you're my adrenaline. Brought out this other side of me You don't
even know Controlling my whole anatomy, oh Fingers are holding you right at the
edge You're slipping out of my hands Keeping my secrets all up in my head I'm
scared that you won't want me back, oh I dance to every song like it's about ya I
drink 'til I kiss someone who looks like ya I wish that I was honest when I had
you I shoulda told you that I wanted you for me I dance to every song like it's
about ya I drink 'til I kiss someone who looks like ya"
```

```
}
```

One to Many Relation

Define a Many-to-One/One-to-Many relationship in your entities to model scenarios where Entity A can have multiple instances of Entity B, but Entity B contains only one instance of Entity A. For example, each Playlist entity could have multiple Song entities, and each User entity can have multiple Playlist entities, while each Playlist belongs to a single User.

In Nest.js, leverage TypeORM's @ManyToOne and @OneToMany decorators to annotate these relationships in your entities. This enables Nest.js to automatically manage the relations through its underlying ORM capabilities.

From a software engineering standpoint, this design pattern enforces the Single Responsibility Principle (SRP) by clearly defining and separating the responsibilities of each entity. It ensures that each entity only manages the data and relationships that are directly relevant to it, thereby making the system easier to understand, debug, and maintain.

Create a Playlist Entity and add Relations

```
import { Song } from "src/songs/song.entity";

import { User } from "src/users/user.entity";

import {

  Column,

  Entity,

  ManyToOne,

  OneToMany,

  PrimaryGeneratedColumn,

} from "typeorm";


@Entity("playlists")

export class Playlist {
```

```typescript
@PrimaryGeneratedColumn()

id: number;


@Column()

name: string;


/**

 * Each Playlist will have multiple songs

 */

@OneToMany(() => Song, (song) => song.playList)

songs: Song[];


/**

 * Many Playlist can belong to a single unique user

 */


@ManyToOne(() => User, (user) => user.playLists)

user: User;

}
```

Add a @OneToMany relationship between the song and the playlist, as well as a @ManyToOne relationship between the playlist and the user. The playlist table will include userId as a foreign key.

Add Many to One Relation in Song Entity

```
export class Song {

  /**

   * Many songs can belong to the playlist for each unique user

   */

  @ManyToOne(() => Playlist, (playList) => playList.songs)

  playList: Playlist;

}
```

This Song entity will have the `playlistId` as a foreign key in the songs table.

Add One to Many Relation in User

```
export class User {

  /**

   * A user can create many playLists

   */

  @OneToMany(() => Playlist, (playList) => playList.user)

  playLists: Playlist[];

}
```

Create PlayList Module

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

import { PlayListsController } from "./playlists.controller";
```

```
import { TypeOrmModule } from "@nestjs/typeorm";

import { Playlist } from "./playlist.entity";

import { PlayListsService } from "./playlists.service";

import { Song } from "src/songs/song.entity";

import { User } from "src/users/user.entity";


@Module({

  imports: [TypeOrmModule.forFeature([Playlist, Song, User])],

  controllers: [PlayListsController],

  providers: [PlayListsService],

})

export class PlayListModule {}
```

Treat the `PlayListModule` as a feature module within your Nest.js application. In the context of software modularization, a feature module encapsulates specific functionalities, allowing for clean separation of concerns. By adopting this approach, you're following the Single Responsibility Principle, making the application easier to understand, develop, and test.

Create PlayList Service

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

import { InjectRepository } from "@nestjs/typeorm";

import { Playlist } from "./playlist.entity";

import { Repository } from "typeorm";

import { CreatePlayListDto } from "./dto/create-playlist.dto";

import { Song } from "src/songs/song.entity";
```

```typescript
import { User } from "src/users/user.entity";

@Injectable()
export class PlayListsService {
  constructor(
    @InjectRepository(Playlist)
    private playListRepo: Repository<Playlist>,

    @InjectRepository(Song)
    private songsRepo: Repository<Song>,

    @InjectRepository(User)
    private userRepo: Repository<User>
  ) {}
  async create(playListDTO: CreatePlayListDto): Promise<Playlist> {
    const playList = new Playlist();
    playList.name = playListDTO.name;

    // songs will be the array of IDs that we are getting from the DTO object
    const songs = await this.songsRepo.findByIds(playListDTO.songs);
    //Set the relation for the songs with the playlist entity
    playList.songs = songs;
```

```
    // A user will be the ID of the user we are getting from the request

    //When we implemented the user authentication this id will become the logged
in user id

    const user = await this.userRepo.findOneBy({ id: playListDTO.user });

    playList.user = user;


    return this.playListRepo.save(playList);

  }

}
```

Create a Data Transfer Object DTO for the Playlist

```
import { IsArray, IsNotEmpty, IsNumber, IsString } from "class-validator";


export class CreatePlayListDto {

  @IsString()

  @IsNotEmpty()

  readonly name;


  @IsNotEmpty()

  @IsArray()

  @IsNumber({}, { each: true })

  readonly songs;
```

```
  @IsNumber()

  @IsNotEmpty()

  readonly user: number;

}
```

To create a PlayList, send a request that includes the name, an array of song IDs, and a user ID. Ensure that these parameters are formatted correctly in the request payload.

Create PlayListsController

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

import { Playlist } from "./playlist.entity";

import { CreatePlayListDto } from "./dto/create-playlist.dto";

import { PlayListsService } from "./playlists.service";


@Controller("playlists")

export class PlayListsController {

  constructor(private playListService: PlayListsService) {}

  @Post()

  create(

    @Body()

    playlistDTO: CreatePlayListDto

  ): Promise<Playlist> {

    return this.playListService.create(playlistDTO);

  }
```

```
}
```

Create an endpoint to facilitate the addition of a new playlist. Enable the end user to send a POST request to localhost:3000/playlists for this purpose. Ensure that PlaylistRepository, SongRepository, and UserRepository are available in the PlayListService.

Register the PlayList Module in AppModule

```
@Module({

  imports: [

    TypeOrmModule.forRoot({

    entities: [Song, Artist, User, Playlist],

  }),

  SongsModule,

  PlayListModule,

  ],

})
```

Register the Playlist entity in the entities array. This step is essential for Nest.js to recognize the entity and make it available for database operations. It aligns with the software engineering principle of modularization, allowing each entity to serve as an isolated module within the larger application structure.

Test the Application

You can test the application by sending the POST api request to http://localhost:3001/playlists. You also have to provide the JSON body to save the playlist record

```
{

  "name": "Feel Good Now",
```

```
  "songs": [6],

  "user": 1

}
```

Chapter 7 Authentication

User Signup

Save the user in the database after account creation in the application. In a Nest.js environment, you'll often use a service with injected repository to handle this data-persistence layer. This follows the software engineering principle of separation of concerns, keeping the data storage logic distinct from the business logic.

Install Dependencies

Save the user password in an encrypted format. Utilize the `bcryptjs` package for password encryption within the Nest.js ecosystem. While it's generally advisable to also use a salt for added security, this project will focus solely on encryption. Install this dependency to proceed.

Encrypting user passwords is crucial for data security and aligns with the software engineering principle of confidentiality. The choice of `bcryptjs` is notable for its reliable and secure encryption algorithm.

```
"bcryptjs": "^2.4.3",
```

You can add this entry into dependencies in the `package.json` file and run `npm install`. Install typescript typing for this package

```
"@types/bcryptjs": "^2.4.2",
```

Create User and Auth Module

```
// app.module.ts
import { UsersModule } from './users/users.module';
import { AuthModule } from './auth/auth.module';


@Module({
imports: [
  CatsModule,
  SongsModule,
```

```
  PlayListModule,
  UsersModule,
  AuthModule,
],
})
```

Create two new modules with nest cli.

```typescript
//auth.module.ts
import { Module } from "@nestjs/common";
import { AuthController } from "./auth.controller";
import { UsersModule } from "src/users/users.module";
import { AuthService } from "./auth.service";

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

We are exporting the `AuthService` from `AuthModule` which means when we import the AuthModule into another module you can use the AuthService or inject the authservice into your imported module. If you don't have an `AuthService` you can create it using nest-cli

We are also importing the `UsersModule` here because we need `UserService` here.

```typescript
// users.module.ts
import { Module } from "@nestjs/common";
import { UsersService } from "./users.service";
import { TypeOrmModule } from "@nestjs/typeorm";
import { User } from "./user.entity";

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

Creating AuthController

```typescript
import { Body, Controller, Post } from "@nestjs/common";
import { CreateUserDTO } from "src/users/dto/create-user.dto";
import { User } from "src/users/user.entity";
import { UsersService } from "src/users/users.service";

@Controller("auth")
export class AuthController {
  constructor(private userService: UsersService) {}
  @Post("signup")
  signup(
    @Body()
    userDTO: CreateUserDTO
  ): Promise<User> {
    return this.userService.create(userDTO);
  }
}
```

We have created a new route for signup to handle the signup request. We did not create a `CreateUserDTO` and `create` method inside the `UsersService`

Creating UsersService

```typescript
import { Injectable } from "@nestjs/common";
import { InjectRepository } from "@nestjs/typeorm";
import { Repository } from "typeorm";
import { User } from "./user.entity";
import * as bcrypt from "bcryptjs";

import { CreateUserDTO } from "./dto/create-user.dto";

@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private userRepository: Repository<User> // 1.
  ) {}

  async create(userDTO: CreateUserDTO): Promise<User> {
    const salt = await bcrypt.genSalt(); // 2.
    userDTO.password = await bcrypt.hash(userDTO.password, salt); // 3.
    const user = await this.userRepository.save(userDTO); // 4.
    delete user.password; // 5.
    return user; // 6.
  }
```

```
}
```

1. We have imported the User Entity `imports: [TypeOrmModule.forFeature([User])]`, in the UsersModule now we can inject the UsersRepository inside the UsersService.
2. We have created the salt number to encrypt the password
3. We have encrypted the password and set it to userDTO password property
4. You have to save the user by calling the save method from the repository
5. You don't need to send the user password in the response. You have to delete the user password from the user object
6. Finally we need to return the user in the response

Create CreateUserDTO

```typescript
import { IsEmail, IsNotEmpty, IsString } from "class-validator";

export class CreateUserDTO {
  @IsString()
  @IsNotEmpty()
  firstName: string;

  @IsString()
  @IsNotEmpty()
  lastName: string;

  @IsEmail()
  @IsNotEmpty()
  email: string;

  @IsString()
  @IsNotEmpty()
  password: string;
}
```

Refactor the User Entity

```typescript
@Entity("users")
export class User {
  @Column({ unique: true })
  email: string;

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

When working with TypeORM, there might be cases where you want to exclude one or multiple columns (fields) from being selected. I don't want to send the password in the response that is why I have added Exclude.

Test the Application

Signup User


```
POST http://localhost:3000/auth/signup
Content-Type: application/json
```


```json
{
"firstName": "john",
"lastName": "doe",
"email": "john@gmail.com",
"password": "123456"
}
```


User Login

What is JSON Web Token Authentication

JSON Web Token (JWT) authentication is a method of securely transmitting information between parties as a JSON object. It is commonly used for authentication and authorization purposes in web applications and APIs. The flow of JWT authentication involves the following steps:

User Authentication: The user provides their credentials (e.g., username and password) to the authentication server. The server verifies the credentials and generates a JWT if they are valid.

JWT Generation: Upon successful authentication, the authentication server creates a JWT containing three parts: header, payload, and signature.

Header: It typically consists of two parts: the token type, which is JWT, and the hashing algorithm used to create the signature. Payload: This contains the claims or statements about the user, such as their username, role, and any additional information. The payload is not encrypted but is Base64Url encoded. Signature: The signature is created by combining the encoded header, encoded payload, and a secret key known only to the server. It ensures the integrity of the token and prevents tampering. JWT Issuance: The server responds to the user's authentication request by sending the JWT back as a response.

Token Storage: The client (usually a web browser or a mobile app) stores the received JWT securely. It can be stored in various places, such as local storage, cookies, or session storage, depending on the application's requirements.

Token Usage: For subsequent requests to protected resources, the client includes the JWT in the request headers, typically as the "Authorization" header with the "Bearer" scheme, followed by the JWT.

Token Verification: When the server receives a request with a JWT, it extracts the token from the header, payload, and signature.

Signature Validation: The server recalculates the signature using the same algorithm and the secret key. If the recalculated signature matches the signature in the token, it ensures the token's integrity. Expiration Check: The server checks the expiration time (exp) claim in the payload to ensure the token has not expired. If it has expired, the server rejects the request. Additional Validations: The server may perform additional checks based on the application's requirements, such as verifying the token's audience (aud) or checking for revoked tokens. Access Grant: If the token passes all the validations, the server grants access to the requested resource or performs the requested action on behalf of the user.

Token Renewal: If the token has an expiration time, the client can request a new JWT before the current one expires. This process is typically done using a refresh token or by re-authenticating the user.

The JWT authentication flow allows the client to include a token with each request, eliminating the need for server-side session storage. It enables stateless authentication, making it suitable for distributed systems and APIs.

Install Passport

```
"@nestjs/passport": "^9.0.3",
"passport": "^0.6.0",
```

You have to install these two dependencies

Create Login Route and Handler

```
@Post('login')
login(
@Body()
loginDTO: LoginDTO,
) {
return this.authService.login(loginDTO);
}
```

We have to create a new login route in `AuthService`. We have called the login method from Authservice. We have not created the login method in AuthService yet, let's create the login method

```typescript
import { Injectable, UnauthorizedException } from "@nestjs/common";
import { LoginDTO } from "./dto/login.dto";
import { UsersService } from "src/users/users.service";
import * as bcrypt from "bcryptjs";
import { User } from "src/users/user.entity";

@Injectable()
export class AuthService {
  constructor(private userService: UsersService) {}

  async login(loginDTO: LoginDTO): Promise<User> {
    const user = await this.userService.findOne(loginDTO); // 1.
    const passwordMatched = await bcrypt.compare(
      loginDTO.password,
      user.password
    ); // 2.
    if (passwordMatched) {
      // 3.
      delete user.password; // 4.
      return user;
    } else {
      throw new UnauthorizedException("Password does not match"); // 5.
    }
  }
}
```

1. We have to find the user based on email. We need to get the email and password from the request body.
2. We will compare the user password with an encrypted password that we saved in the last video
3. If the password matches then delete the user password and send the user back in the response. It means the user has logged in successfully
4. If the password does not match we have to send the error back in the response

Create LoginDTO

You have to create the LoginDTO file inside the `auth/dtos/login.dto.ts`

```typescript
// login.dto.ts
import { IsEmail, IsNotEmpty, IsString } from "class-validator";

export class LoginDTO {
  @IsEmail()
  @IsNotEmpty()
  email: string;
```

```
  @IsString()
  @IsNotEmpty()
  password: string;
}
```

Create findOne method inside UsersService

```
async findOne(data: Partial<User>): Promise<User> {
 const user = await this.userRepository.findOneBy({ email: data.email });
 if (!user) {
 throw new UnauthorizedException('Could not find user');
 }
 return user;
}
```

Test the Application

```
### Login User


POST http://localhost:3001/auth/login
Content-Type: application/json


{
"email": "john@gmail.com",
"password": "123456"
}
```

We did not send the JSON web token back in the response. In the next lesson, I will teach you how to send the JSOn web token in the response when the user has successfully logged in

Authenticate User

Find the user from the database based on their email and encrypt their password. If the user logs in successfully, generate a JSON Web Token (JWT) and include it in the response. This is a key step in stateless authentication, a software engineering principle that improves scalability and security. In

the previous lesson, the generation and inclusion of the JWT in the response were omitted, which left the authentication process incomplete.

Let's create the JSON web token:

Install Dependencies

```
"dependencies" : {
  "@nestjs/jwt": "^10.0.3",
  "passport-jwt": "^4.0.1",
},
"devDependencies": {
  "@types/passport-jwt": "^3.0.8",
}
```

We have to install these packages to implement complete JSON Web Token authentication and Authorization

Import JWT Module in AuthModule

```
// auth.module.ts
import { JwtModule } from '@nestjs/jwt';

imports: [UsersModule, JwtModule.register({ secret: 'HAD_12X#@' })],
controllers: [AuthController],
providers: [AuthService],
exports: [AuthService],
```

You need to register the JwtModule module in the AuthModule by providing the unique secret key. We will use this secret key to decode the token or validate the token

Refactor the AuthService

```
// auth.service.ts


export class AuthService{
constructor(
  private userService: UsersService,
  private jwtService: JwtService, // 1
) {}


async login(loginDTO: LoginDTO): Promise<{ accessToken: string }> { // 1.


/// ...
```

```
if (passwordMatched) {
  delete user.password;
  return user;
  // Sends JWT Token back in the response
  const payload = { email: user.email, sub: user.id };


  return {
    accessToken: this.jwtService.sign(payload),
  };
}
```

1. Inject JwtService as a dependency.
2. If the password matches, generate the JWT token using the `jwtService.sign` method.
3. Provide the payload, which should include the user email and userId inside the JWT token. Choose a name for the user ID field; 'sub' is used here but any name can be applied.

Create a new file to save Constant Values

You have to create a new file inside the auth folder.

```
// auth.constants.ts
export const authConstants = {
  secret: "HAD_12X#@",
};
```

Refactor Auth Module

```
// auth.module.ts


imports: [
  UsersModule,
  JwtModule.register({
  secret: authConstants.secret,
  signOptions: { // 1.
  expiresIn: '1d',
},
}),
],
```

1. The token will expire after one day

Create JWT Strategy Service

```ts
// jwt.strategy.ts

import { Injectable } from "@nestjs/common";
import { PassportStrategy } from "@nestjs/passport";
import { ExtractJwt, Strategy } from "passport-jwt";
import { AuthService } from "./auth.service";
import { authConstants } from "./auth.constants";

@Injectable()
export class JWTStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(), // 1.
      ignoreExpiration: false, // 2.
      secretOrKey: authConstants.secret, // 3.
    });
  }

  async validate(payload: any) {
    // 4.
    return { userId: payload.sub, email: payload.email }; // 5.
  }
}
```

You have to create new a file inside the auth folder with `jwt.strategy.ts`

1. Create the JWTStrategy service, extending it with PassportStrategy. When applying `@AuthGuard('jwt')`, the validate function will be called, automatically adding the `userId` and `email` to the `req.user` object.

Register the JWTStrategy in AuthModule

```ts
// auth.module.ts
import { JWTStrategy } from './jwt.strategy';
import { PassportModule } from '@nestjs/passport';


imports: [
  PassportModule,
]
providers: [AuthService, JWTStrategy],
```

You have to register the JWTStrategy as a provider in AuthModule. You also have to register the PassportModule. It will allow us to use the PassportStrategy class

Create JWT Guard

A Guard is like a middleware in express.js. You can implement role-based authentication using guards

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

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

Let's create a provider which is JwtAuthGuard. You have to apply the JWTAuthGuard to protect a private route

Protect Private Route in AppController

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

Create a new protected route inside the AppController. When sending a request to access the profile, the response will include the user ID and email. Apply JwtAuthGuard to any route in any controller to secure the endpoint.

Test the Application

```
GET http://localhost:3000/profile
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImhhaWRlcl9hbGkzQGdtYWlsLmNvbSIs
InN1YiI6NywiaWF0IjoxNjg0MjM3ODgzLCJleHAiOjE2ODQzMjQyODN9.DztMDAKZOnQjZPFkFPiWiJUm
I_VnrNfNvfwPI0yJ8MA
```

First, send a login request to obtain the token, and then provide that token in the header.

What is Role-based Authentication

Role-based authentication is a method of access control that regulates user permissions and privileges within a system based on their assigned roles. In this approach, users are categorized into roles based on their job responsibilities, functions, or levels of authority within an organization.

Each role is associated with a set of permissions that determine what actions and resources a user with that role can access. These permissions can include read, write, modify, delete, and other operations on various system resources such as files, databases, or functionalities.

We are going to implement this scenario:

- An artist can upload/create the song

We have to restrict the access of creating songs endpoint. Only artists can access this endpoint and create a song

Create an Artists Module

```
import { Module } from "@nestjs/common";
import { ArtistsService } from "./artists.service";
import { TypeOrmModule } from "@nestjs/typeorm";
import { Artist } from "./artist.entity";

@Module({
  imports: [TypeOrmModule.forFeature([Artist])],
  providers: [ArtistsService],
  exports: [ArtistsService],
})
export class ArtistsModule {}
```

Add ArtistsModule into AppModule

```
import { AuthModule } from './auth/auth.module';


@Module({
imports: [
  PlayListModule,
  UsersModule,
  AuthModule,
  ArtistsModule,
],
})
```

Create ArtistsService

```typescript
import { Injectable } from "@nestjs/common";
import { InjectRepository } from "@nestjs/typeorm";
import { Repository } from "typeorm";
import { Artist } from "./artist.entity";

@Injectable()
export class ArtistsService {
  constructor(
    @InjectRepository(Artist)
    private artistRepo: Repository<Artist>
  ) {}

  findArtist(userId: number): Promise<Artist> {
    return this.artistRepo.findOneBy({ user: { id: userId } });
  }
}
```

We will use the findArtist method to check if the current logged in user is an artist or not.

Refactor login method in AuthService

```typescript
constructor(
private userService: UsersService,
private jwtService: JwtService,
private artistService: ArtistsService,
) {}
```

You have to inject the artist's service in the `AuthService` constructor

```typescript
const payload: PayloadType = { email: user.email, userId: user.id }; // 1

// find if it is an artist then the add the artist id to payload
const artist = await this.artistService.findArtist(user.id); // 2
if (artist) {
  // 3
  payload.artistId = artist.id;
}
```

1. Refactor the payload method in the login function by changing `sub: user.id` to `userId: user.id`.
2. Find the artist based on the logged-in user.
3. If the user is an artist, save the artist ID in the payload; this artist ID will be used when decoding the token in the ArtistGuard.

Create a PayloadType

You have to create a payload type inside the auth folder. I have created a separate folder for the types to store all my types here

```
//types/payload.type.ts
export interface PayloadType {
  email: string;
  userId: number;
  artistId?: number;
}
```

Create a new JwtArtistGuard

In Nest.js, implementing role-based authentication involves the use of guard functions, which act as middleware to authorize requests. Each role can be associated with a distinct guard function that validates whether a user has the appropriate permissions to access a resource. This is an application of the "Single Responsibility Principle," as each guard function focuses solely on authorizing a specific role, making the code easier to manage and extend.

```
// jwt-artist.guard.ts
import {
  ExecutionContext,
  Injectable,
  UnauthorizedException,
} from "@nestjs/common";
import { AuthGuard } from "@nestjs/passport";
import { Observable } from "rxjs";
import { PayloadType } from "./types/payload.type";

@Injectable()
export class JwtArtistGuard extends AuthGuard("jwt") {
  canActivate(
    context: ExecutionContext
  ): boolean | Promise<boolean> | Observable<boolean> {
    return super.canActivate(context);
  }
  handleRequest<TUser = PayloadType>(err: any, user: any): TUser {
    // 1
    if (err || !user) {
      //2
      throw err || new UnauthorizedException();
    }
    console.log(user);
    if (user.artistId) {
      // 3
```

```
      return user;
    }
    throw err || new UnauthorizedException();
  }
}
```

1. When you apply the JwtAuthGuard at the controller function it will call the handleRequest function
2. If there is an error or no user then it will send the unauthorized error
3. Here we are checking the user role, if it is an artist then we have to return the user. This user will have have `email`, `userId`, and `artistId` property

Refactor the validate method in JwtStrategy

```
// jwt-strategy.ts
async validate(payload: PayloadType) { //1.
 return {
 userId: payload.userId,
 email: payload.email,
 artistId: payload.artistId, // 2
 };
}
```

1. Add the payloadType for the argument.
2. Include artistId in the response; note that artistId is an optional property and may be null.

Apply JwtArtistGuard on creating songs endpoint

```
//songs.controller.ts
@Post()
@UseGuards(JwtArtistGuard) // 1
  create(@Body() createSongDTO: CreateSongDto, @Request() req): Promise<Song> {
  console.log(req.user);
  return this.songsService.create(createSongDTO);
}
```

Now we have protected this endpoint, only artist can access this endpoint and create a new song

Test the Application

1. First of all you must have an artist record in your DB
2. If you don't have you can create an artist manually using pgAdmin
3. You have to send the login request as an artist
4. It will give you the access token you have to use that token to access to Create Songs endpoint

Artist Login User

```
POST http://localhost:3001/auth/login
Content-Type: application/json


{
  "email": "john_doe@gmail.com",
  "password": "123456"
}
```

It will give you the token, you have to use that token to create a new song

Create New SONGS REQUEST as An Artist
```
POST http://localhost:3001/songs
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImhhaWRlcl9hbGkzQGdtYWlsLmNvbSIs
InVzZXJJJZCI6NywiaWF0IjoxNjg0MzA3Mjg3LCJleHAiOjE2ODQzOTM2ODd9.A9SEhTH0O0SR5-UELhMh
ak5MVIyY2ISRwR-o2RKF_dY
Content-Type: application/json


{
  "title": "You for me",
  "artists": [1],
  "releasedDate" : "2022-08-29",
  "duration" :"02:34",
  "lyrics": "by, you're my adrenaline. Brought out this other    side of me You
don't even know Controlling my whole anatomy, oh Fingers are holding you right at
the edge You're slipping out of my hands Keeping my secrets all up in my head I'm
scared that you won't want me back, oh I dance to every song like it's about ya I
drink 'til I kiss someone who looks like ya I wish that I was honest when I had
you I shoulda told you that I wanted you for me I dance to every song like it's
about ya I drink 'til I kiss someone who looks like ya"
}
```

What is Two Factor Authentication

Two-factor authentication (2FA), also known as multi-factor authentication (MFA), is a security
mechanism that requires users to provide two or more separate forms of identification to verify their
identity and gain access to a system or account. It adds an extra layer of security beyond traditional
username and password authentication.

The two factors typically fall into three categories:

Something you know: This factor involves knowledge-based information that the user possesses, such as a password, PIN, or answers to security questions.

1. Something you have: This factor requires the user to possess a physical device or object, such as a smartphone, hardware token, or smart card. This device generates or receives a unique one-time code or a cryptographic key.
2. Something you are: This factor refers to biometric characteristics unique to the individual, such as fingerprints, facial recognition, iris scans, or voice recognition.
3. The combination of these factors increases the security of authentication because an attacker would need to compromise multiple elements to gain unauthorized access. Even if one factor is compromised, the additional factor(s) provide an extra layer of protection.

Here's a simplified example of how two-factor authentication works:

1. The user enters their username and password on a login page.
2. After successful initial authentication, the system prompts the user for a second form of verification.
3. The user may be required to provide a one-time code generated by an authentication app on their smartphone or received via SMS.
4. The user enters the one-time code to complete the authentication process.
5. If both factors are verified successfully, access is granted to the user.

Two-factor authentication is widely used across various systems, including online accounts (email, social media, banking), VPNs, cloud services, and more. It significantly reduces the risk of unauthorized access due to compromised passwords or stolen credentials, enhancing overall security and protecting sensitive information.

Install Dependencies

Speakeasy is a one-time passcode generator, ideal for use in two-factor authentication, that supports Google Authenticator and other two-factor devices.

It is well-tested and includes robust support for custom token lengths, authentication windows, hash algorithms like SHA256 and SHA512, and other features, and includes helpers like a secret key generator.

Speakeasy implements one-time passcode generators as standardized by the Initiative for Open Authentication (OATH). The HMAC-Based One-Time Password (HOTP) algorithm defined in RFC 4226 and the Time-Based One-time Password (TOTP) algorithm defined in RFC 6238 are supported. This project incorporates code from passcode, originally a fork of Speakeasy, and notp.

```
"speakeasy": "^2.0.0",
```

You have to add this entry to `dependencies` in the package.json file and run `npm install`

```
"@types/speakeasy": "^2.0.7",
```

You have to add this entry to `devDependencies` in the package.json file and run `npm install`

Update User Entity

```ts
// user.entity.ts
@Column({ nullable: true, type: 'text' })
twoFASecret: string;


@Column({ default: false, type: 'boolean' })
enable2FA: boolean;
```

You have to add two new columns for two-factor authentication. The first column will be used to store the secret key for each user. The second column will be used to enable or disable the two-factor authentication.

Add new Enable2FA type

```ts
//types/auth-types.ts
export type Enable2FAType = {
  secret: string;
};
```

Add a new method in AuthService to enable two-factor auth

```ts
//auth.service.ts
import * as speakeasy from 'speakeasy';


async enable2FA(userId: number) : Promise<Enable2FAType> {
const user = await this.userService.findById({ id: userId }); //1
if (user.enable2FA) { //2
return { secret: user.twoFASecret };
}
const secret = speakeasy.generateSecret(); //3
console.log(secret);
user.twoFASecret = secret.base32; //4
await this.userService.updateSecretKey(user.id, user.twoFASecret); //5
return { secret: user.twoFASecret }; //6
}
```

1. You have to create a new method to find the user on the based on user id
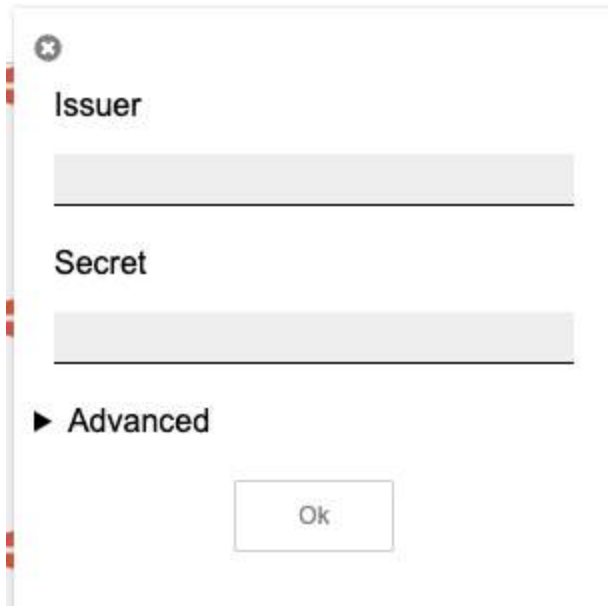
2. If user has already enabled the 2 factor authentication we have to return the secret key.
3. If the user did not enable the 2-factor authentication then we have to generate the secret key. This `speakeasy.generateSecret()` will return an object with `secret.ascii`, `secret.hex`, and `secret.base32`.
4. We are going to use the `base32` secret key
5. Finally, we have to update the `twoFAsecret` key for the specific user
6. You have to return the secret key in the response.

Use base32 secret key in QR code App

You have to install the chrome extension or Google authenticator app on your phone

You have received the secret key now you have to create a new app in your QR code application. It will ask you to add a manual secret key and the name of your app
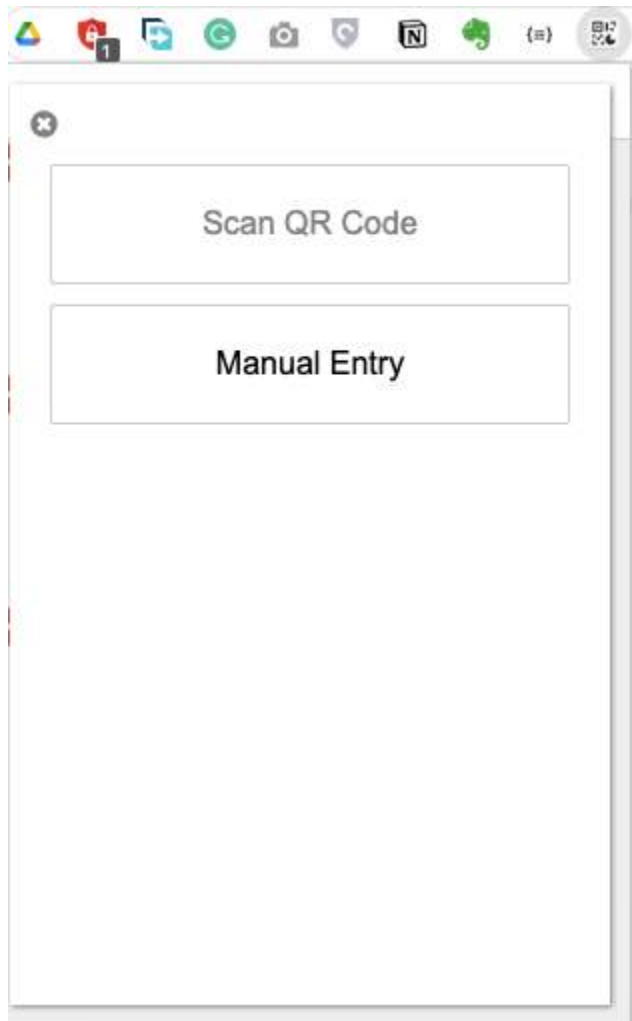
Refactor UsersService

```typescript
import { Repository, UpdateResult } from 'typeorm';


//users.service.ts
async updateSecretKey(userId, secret: string): Promise<UpdateResult> {
return this.userRepository.update(
{ id: userId },
{
twoFASecret: secret,
enable2FA: true,
},
);
}
```

You have to the secret key and enable the 2-factor authentication for a user

```typescript
//users.service.ts
async findById(id: number): Promise<User> {
return this.userRepository.findOneBy({ id: id });
}
```

Create Endpoint in AuthController to enable 2FA

```typescript
//auth.controller.ts
```

```
@Post('enable-2fa')
@UseGuards(JwtAuthGuard)
enable2FA(
@Request()
req,
): Promise<Enable2FAType> {
console.log(req.user);
return this.authService.enable2FA(req.user.userId);
}
```

Test the Enable Authentication Endpoint

```
### Enable 2FA Authentication
POST http://localhost:3000/auth/enable-2fa
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImhhaWRlcl9hbGkzQGdtYWlsLmNvbSIs
InN1YiI6NywiaWF0IjoxNjg0NDEyMjk2LCJleHAiOjE2ODQ0OTg2OTZ9.Fg0K4gJABBP3nqt8PMK72MzS
nFVK0xRaEeC_aDxnfeo
```

You have to provide your own user token

Implement Disable 2-Factor Authentication

You have to new method inside the `auth.service.ts` to disable the authentication

```
//auth.service.ts
async disable2FA(userId: number): Promise<UpdateResult> {
return this.userService.disable2FA(userId);
}
```

```
//users.service.ts
async disable2FA(userId: number): Promise<UpdateResult> {
return this.userRepository.update(
{ id: userId },
{
enable2FA: false,
twoFASecret: null,
},
);
}
```

You have to create a new route to disable authentication

```
//auth.controller.ts
@Get('disable-2fa')
@UseGuards(JwtAuthGuard)
disable2FA(
@Request()
req,
): Promise<UpdateResult> {
return this.authService.disable2FA(req.user.userId);
}
```

Now you can test the disabled authentication endpoint

```
GET http://localhost:3000/auth/disable-2fa
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImhhaWRlcl9hbGkzQGdtYWlsLmNvbSIs
InN1YiI6NywiaWF0IjoxNjg0NDkyOTk1LCJleHAiOjE2ODQ1NzkzOTV9.vhAHpdyuQHWvsET2sSLvQpr3
3vpk8K089NLiENgh7pM
```

Verify One-time password/token

```
// auth.controller.ts
@Post('validate-2fa')
@UseGuards(JwtAuthGuard)
validate2FA(
@Request()
req,
@Body()
ValidateTokenDTO: ValidateTokenDTO,
): Promise<{ verified: boolean }> {
return this.authService.validate2FAToken(
req.user.userId,
ValidateTokenDTO.token,
);
}
```

You have to create an endpoint to validate the one-time password/token

```
// auth/dto/validate-token.dto.ts
import { IsNotEmpty, IsString } from "class-validator";

export class ValidateTokenDTO {
  @IsNotEmpty()
  @IsString()
  token: string;
}
```

Now you have to create a new method inside the `auth.service.ts` to verify the token

```typescript
// validate the 2fa secret with provided token
async validate2FAToken(
userId: number,
token: string,
): Promise<{ verified: boolean }> {
try {
// find the user on the based on id
const user = await this.userService.findById(userId);


// extract his 2FA secret


// verify the secret with a token by calling the speakeasy verify method
const verified = speakeasy.totp.verify({
secret: user.twoFASecret,
token: token,
encoding: 'base32',
});


// if validated then sends the json web token in the response
if (verified) {
return { verified: true };
} else {
return { verified: false };
}
} catch (err) {
throw new UnauthorizedException('Error verifying token');
}
}
```

Let's test the validate token endpoint.

Validate 2FA Token


```
POST http://localhost:3000/auth/validate-2fa
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InNhbUBnbWFpbC5jb20iLCJzdWIiOjks
```

ImlhdCI6MTY4NDQ5ODg0MSwiZXhwIjoxNjg0NTg1MjQxfQ.dKgmLSsGctbWR9HKz3ByfS2ZpUGiNR234u
qEgs0pgtQ
Content-Type: application/json

```
{
"token": "054603"
}
```

The token I have provided is one-time password/token from the authenticator app. You have to provide your unique token from your authenticator app

If user enabled the 2FA refactor the login method

```
async login(
loginDTO: LoginDTO,
): Promise<{ accessToken: string } | { validate2FA: string; message: string }>
```

You have to refactor the return type. If the user has enabled the 2FA it will use the second return type with a custom message

```
const user = await this.userService.findOne(loginDTO);
const passwordMatched = await bcrypt.compare(
loginDTO.password,
// Sends JWT Token back in the response
const payload = { email: user.email, sub: user.id };


// If user has enabled 2FA and have the secret key then
if (user.enable2FA && user.twoFASecret) { //1.
// sends the validateToken request link
// else otherwise sends the json web token in the response
return { //2.
validate2FA: 'http://localhost:3000/auth/validate-2fa',
message:
'Please send the one-time password/token from your Google Authenticator App',
};
}
return {
accessToken: this.jwtService.sign(payload),
};
```

1. You have to add a new code here to check user enabled the 2FA.
2. If the user enabled the 2FA then we have to send the link to validate the token from your QR code app

What is an API Key Authentication

API key authentication is a method of authenticating and securing access to an application programming interface (API). It involves the use of an API key, which is a unique identifier that grants access to specific API resources and operations.

The purpose of API key authentication is to control and monitor access to an API by assigning and managing unique keys for individual users or applications.

API keys serve as a form of credential, allowing the API provider to track and control API usage, enforce rate limits, and monitor and identify unauthorized access attempts.

When do you need API Keys

- You want to control the number of calls made to your API.
- You want to identify usage patterns in your API's traffic

API key authentication is typically used in scenarios where multiple users or applications need to access an API.

It enables the API provider to identify and track the usage of different clients, which can be beneficial for managing access levels, implementing usage quotas, and identifying potential abuse or security breaches.

Where can we use API Keys

API key authentication can be used in a variety of contexts, including web and mobile applications, microservices, and third-party integrations.

It is commonly employed by API providers to ensure secure and controlled access to their services. By requiring API keys, providers can track usage, enforce restrictions, and have the flexibility to revoke access for specific keys if necessary.

Complete Flow of API Key Authentication

1. The client initiates a request to access an API resource or perform an operation.
2. The client includes the API key as part of the request. This can be done in various ways, such as including the key in the request header, query parameters, or as part of the request body.
3. The API server receives the request and extracts the API key.
4. The API server verifies the API key's validity and authenticity. This verification process may involve checking against a database or performing cryptographic operations.

5. If the API key is valid, the server grants access to the requested resources or operations based on the permissions associated with that key. If the key is invalid or expired, the server denies access and returns an appropriate error response.
6. The API server processes the client's request and returns the requested data or performs the requested operation.
7. The client receives the response from the API server and can continue interacting with the API if the authentication was successful.

Steps

1. Step 1: Generate API Keys
2. Step 2: Create and Store API key
3. Step 3: Create an API Key strategy
4. Step 4: Register API key strategy in Auth Module
5. Step 5: Validate the User by API key
6. Step 6: Apply API key Authentication on protected Route

Step 1: Generate API Keys

First of all, we have to generate API keys. I am going to use a third third-party package `uuid` to create unique api keys.

You have to install it

```
npm install uuid
```

Step 2: Create and Store the API key

Now we need to create an API Key and store it in the database. Each user will have its own API key. We need to add api key logic in the signup function. When the user is registered we have to assign the unique api key

```
// user.entity.ts


@Column()
apiKey: string;
```

You have to add the `apiKey` column in the user entity. Let's create the api key inside the signup function

```
// users.service.ts


import { v4 as uuid4 } from "uuid";


const user = new User();
```

```
user.firstName = userDTO.firstName;
user.lastName = userDTO.lastName;
user.email = userDTO.email;
user.apiKey = uuid4();
user.password = userDTO.password;

const savedUser = await this.userRepository.save(user);
delete savedUser.password;
return savedUser;
```

I have called the uuid4() to create the Api key.

Let's test the application by sending a signup api request

```
POST http://localhost:3001/auth/signup
Content-Type: application/json


{
"firstName": "sam",
"lastName": "oven",
"email": "sam@gmail.com",
"password": "123456"
}

{
  "firstName": "sam",
  "lastName": "oven",
  "email": "sam@gmail.com",
  "apiKey": "853d94a2-f760-43e3-b384-a9ba94542bf0",
  "twoFASecret": null,
  "id": 1,
  "enable2FA": false
}
```

Step 3: Create an API Key strategy

You have to create a new file ApiKeyStrategy.ts in the auth folder

```
import { Injectable, UnauthorizedException } from "@nestjs/common";
import { PassportStrategy } from "@nestjs/passport";
import { Strategy } from "passport-http-bearer";
import { AuthService } from "./auth.service";


@Injectable()
```

```
export class ApiKeyStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super();
  }
  async validate(apiKey: string) {
    const user = await this.authService.validateUserByApiKey(apiKey);
    if (!user) {
      throw new UnauthorizedException();
    } else {
      return user;
    }
  }
}
```

We have installed the `passport-http-bearer` package and applied this strategy to validate the API keys. It means you need to provide the Api key in the authorization header:

`Authorization: Bearer 853d94a2-f760-43e3-b384-a9ba94542bf0`

When you will apply the AuthGuard `@UseGuards(AuthGuard('bearer'))` to the protected route. It will call the `validate` method from the `ApiKeyStrategy`

Step 4: Register API key strategy in Auth Module

```
import { ApiKeyStrategy } from './api-key.strategy';


providers: [AuthService, JWTStrategy, ApiKeyStrategy],
```

Step 5: Validate the User by API key

```
//auth.service.ts


async validateUserByApiKey(apiKey: string): Promise<User> {
return this.userService.findByApiKey(apiKey);
}
```

I have created a new function inside the `auth` service and validated the user by api key

```
// user.service.ts
async findByApiKey(apiKey: string): Promise<User> {
return this.userRepository.findOneBy({ apiKey });
```

```
}
```

Let's create a new function inside the `user.service.ts` to fetch the user from the DB based on API Key

Step 6: Apply API key Authentication on protected Route

```typescript
//auth.controller.ts
@Get('profile')
@UseGuards(AuthGuard('bearer'))
getProfile(
@Request()
req,
) {
 delete req.user.password;
 return {
  msg: 'authenticated with api key',
  user: req.user,
 };
}
```

A user can access his/her profile. He has to provide the API key to access the protected route. You can protect any route by applying the `AuthGuard('bearer')` to the route

```
GET http://localhost:3001/auth/profile
Authorization: Bearer 853d94a2-f760-43e3-b384-a9ba94542bf0
```

You have to provide the API key in the authorization header to execute the request successfully.

Create a launch.json file in the .vscode folder

Create a `launch.json` file in the `.vscode` folder to configure debugging settings specifically for your Nest.js application. This file allows you to set breakpoints and inspect variables directly in the Visual Studio Code editor, enhancing your development experience. Utilizing a `launch.json` aligns with the software engineering principle of "Configuration as Code," making your development environment easily reproducible and version-controllable.

```json
{
  "version": "0.2.0",
  "configurations": [{
    "name": "Attach",
    "port": 9229,
    "request": "attach",
```

```
        "skipFiles": ["<node_internals>/**"],
        "type": "node"
    }
    ]
}
```

Start the Application using the Debug script

Nestjs provides the debug command to start debugging the application. Find the debug command in the `package.json` file

```
"start:debug": "nest start --debug --watch",
```

Run the application using `npm run start:debug`

In TypeORM, migrations are a way to manage and apply changes to your database schema over time. A migration is a file that contains a set of instructions for creating, modifying, or deleting database tables, columns, constraints, and other schema elements. Migrations help you keep your database schema in sync with your application's models or entities.

When you develop an application using TypeORM, your database schema evolves as you add new features, modify existing ones, or fix issues. Migrations provide a structured and controlled approach to apply these changes to your database without losing existing data.

Here's a general workflow of how migrations work in TypeORM:

1. Creating a Migration: When you make changes to your entities or models, you generate a migration file using TypeORM's CLI command or programmatically using the provided API. The migration file contains both the "up" and "down" methods. The "up" method specifies how to apply the changes to the database, while the "down" method defines how to revert those changes.
2. Applying Migrations: To apply a migration, you execute the migration runner provided by TypeORM, either through the CLI or programmatically in your code. The runner reads the migration files and executes the "up" method, which performs the necessary changes to the database schema.
3. Reverting Migrations: If you encounter issues or need to roll back changes, you can use the migration runner to revert migrations. The runner executes the "down" method of the migration file, which undoes the changes made by the corresponding "up" method.
4. Managing Migration History: TypeORM keeps track of the executed migrations in a table within your database. This table records which migrations have been applied, allowing the runner to determine which migrations are pending or need to be reverted.

By using migrations, you can version control your database schema, collaborate with other developers effectively, and easily deploy schema changes across different environments. Migrations provide a structured and reliable approach to evolving your database schema while preserving data integrity.

Chapter 8 Migrations

Why do we need Migrations

Once you get into production you'll need to synchronize model changes into the database. Typically, it is unsafe to use `synchronize: true` for schema synchronization on production once you get data in your database. Here is where migrations come to help.

A migration is just a single file with SQL queries to update a database schema and apply new changes to an existing database.

Step 1: Move TypeORM config into a separate file

You have to create a new folder with the db name in the root directory and create a new file `data-source.ts`

```
//db/data-source.ts
import { DataSource, DataSourceOptions } from "typeorm";

export const dataSourceOptions: DataSourceOptions = {
  type: "postgres",
  host: "localhost",
  port: 5432,
  username: "postgres",
  password: "root",
  database: "n-test",
  entities: ["dist/**/*.entity.js"], //1
  synchronize: false, // 2
  migrations: ["dist/db/migrations/*.js"], // 3
};

const dataSource = new DataSource(dataSourceOptions); //4
export default dataSource;
```

1. Now you don't need to register the entity manually. TypeORM will find the entities by itself.
2. When you are working with migrations you have to set the synchronize to false because our migration file will update the changes in the database

3. You have to provide the path of migration where you want to store. I chose the dist folder. I will run the migrations as a js file. That's why we need to build the project before running typeorm migrations
4. We will use this data source object when we generate/run the migrations with typeorm cli

Step 2: Refactor TypeORM config in AppModule

```ts
//app.module.ts
import { dataSourceOptions } from "db/data-source";

imports: [TypeOrmModule.forRoot(dataSourceOptions)];
```

Step 3: Migration scripts in package.json file

```json
//package.json
"typeorm": "npm run build && npx typeorm -d dist/db/data-source.js",
"migration:generate": "npm run typeorm -- migration:generate",
"migration:run" : "npm run typeorm -- migration:run",
"migration:revert" : "npm run typeorm -- migration:revert"
```

Step 4: Add a new column in any Entity

I am thinking about adding a `phone` column in the user entity. Maybe, the requirements changed after 3 months and you have to add a new column in the user entity.

```ts
//user.entity.ts
@Column()
phone: string
```

Now you have to update the user table using migrations.

```
npm run migration:generate -- db/migrations/add-user-phone
```

- `db/migrations` : I am telling typeorm I want to save migrations in this folder
- `add-user-phone` : This is the name of the migration

It will generate a new migration file inside the migrations folder

Now you have to run the migration by using this command

```
npm run migration:run
```

This command will alter/update the user table in the database

You can see the users table in the database and see the phone column there.

What is Data Seeding

Data seeding is the process of populating a database with an initial set of data. Applying seed data to a database refers to the process of inserting initial data into a database, usually when the database is first created. This data serves as a baseline and can be used for testing, and development, and to provide some context for the application that will be built on top of the database.

1. Install Dependencies

I am going to use an external package to generate fake/mock data.

```
npm install @faker-js/faker
```

2. Create a seed-data.ts file in the db/seeds directory

```ts
import { Artist } from "src/artists/artist.entity";
import { User } from "src/users/user.entity";
import { EntityManager } from "typeorm";
import { faker } from "@faker-js/faker";
import { v4 as uuid4 } from "uuid";
import * as bcrypt from "bcryptjs";
import { Playlist } from "src/playlist/playlist.entity";

export const seedData = async (manager: EntityManager): Promise<void> => {
  //1
  // Add your seeding logic here using the manager
  // For example:

  await seedUser();
  await seedArtist();
  await seedPlayLists();

  async function seedUser() {
    //2
    const salt = await bcrypt.genSalt();
    const encryptedPassword = await bcrypt.hash("123456", salt);

    const user = new User();
    user.firstName = faker.person.firstName();
    user.lastName = faker.person.lastName();
    user.email = faker.internet.email();
```

```javascript
    user.password = encryptedPassword;
    user.apiKey = uuid4();

    await manager.getRepository(User).save(user);
  }

  async function seedArtist() {
    const salt = await bcrypt.genSalt();
    const encryptedPassword = await bcrypt.hash("123456", salt);

    const user = new User();
    user.firstName = faker.person.firstName();
    user.lastName = faker.person.lastName();
    user.email = faker.internet.email();
    user.password = encryptedPassword;
    user.apiKey = uuid4();

    const artist = new Artist();
    artist.user = user;
    await manager.getRepository(User).save(user);
    await manager.getRepository(Artist).save(artist);
  }

  async function seedPlayLists() {
    const salt = await bcrypt.genSalt();
    const encryptedPassword = await bcrypt.hash("123456", salt);

    const user = new User();
    user.firstName = faker.person.firstName();
    user.lastName = faker.person.lastName();
    user.email = faker.internet.email();
    user.password = encryptedPassword;
    user.apiKey = uuid4();

    const playList = new Playlist();
    playList.name = faker.music.genre();
    playList.user = user;

    await manager.getRepository(User).save(user);
    await manager.getRepository(Playlist).save(playList);
  }
};
```

1. I have created a seedData method with an entity manager argument. I will get the repository
   for each entity by calling the getRepository. This is how you can create mock data

2. I have used the Faker package to generate mock data. You can see the functions `faker.person.firstName()`. You can explore methods from Faker by looking at the documentation

3. Create a new seed module

```
nest g module seed
```

```typescript
import { Module } from "@nestjs/common";
import { SeedService } from "./seed.service";

@Module({
  providers: [SeedService],
})
export class SeedModule {}
```

Make sure you have imported the `SeedModule` in `AppModule`

4.Create a SeedService

```typescript
import { Injectable } from "@nestjs/common";
import { DataSource } from "typeorm";
import { seedData } from "../../db/seeds/seed-data";

@Injectable()
export class SeedService {
  constructor(private readonly connection: DataSource) {}

  async seed(): Promise<void> {
    const queryRunner = this.connection.createQueryRunner(); //1

    await queryRunner.connect(); //2
    await queryRunner.startTransaction(); //3
    try {
      const manager = queryRunner.manager;
      await seedData(manager);

      await queryRunner.commitTransaction(); //4
    } catch (err) {
      console.log("Error during database seeding:", err);
      await queryRunner.rollbackTransaction(); // 5
    } finally {
      await queryRunner.release(); //6
    }
  }
}
```

1. A Query Runner can be used to manage and work with a single real database data source. Each new QueryRunner instance takes a single connection from the connection pool if RDBMS supports connection pooling. For databases not supporting connection pools, it uses the same connection across data source.
2. Use the `connect` method to actually obtain a connection from the connection pool.
3. QueryRunner provides a single database connection. Transactions are organized using query runners. Single transactions can only be established on a single query runner. You can manually create a query runner instance and use it to manually control transaction state.
4. Commit the Transaction
5. If we have errors let's rollback changes we made
6. Make sure to release it when it is not needed anymore to make it available to the connection pool again

5. Run the Seeds

```
// main.ts
import { NestFactory } from "@nestjs/core";
import { AppModule } from "./app.module";
import { ValidationPipe } from "@nestjs/common";
import { SeedService } from "./seed/seed.service";

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  const seedService = app.get(SeedService);
  await seedService.seed();
  await app.listen(3001);
}
bootstrap();
```

This is an entry file, Nestjs will run this file to bootstrap the application, whenever you run the application it will create new data and save it to DB. When you don't need new data you can disable these two lines:

```
// const seedService = app.get(SeedService);
// await seedService.seed();
```

Chapter 9 Configuration

Applications often run in different environments. Depending on the environment, different configuration settings should be used. For example, usually, the local environment relies on specific database credentials, valid only for the local DB instance. The production environment would use a

separate set of DB credentials. Since configuration variables change, best practice is to store configuration variables in the environment.

In Nest.js, configurations refer to the settings and parameters that define the behavior of your application. These configurations can include various aspects, such as server settings, database connections, API keys, logging options, and more.

Nest.js provides a flexible way to manage configurations, allowing you to easily customize the behavior of your application based on different environments (e.g., development, production, testing) or specific deployment scenarios. By separating configuration settings from your code, you can make your application more portable and adaptable to different environments.

Configurations in Nest.js typically consist of key-value pairs, where each key represents a specific setting and the corresponding value represents its configuration value. These configurations are often stored in environment variables or configuration files.

By utilizing configurations, you can ensure that your application can be easily configured without modifying the code itself. This makes it simpler to deploy and maintain your application in different environments or when working with multiple teams.

Nest.js provides various mechanisms to load and use configurations, including the popular dotenv package for loading environment variables from files, as well as custom configuration modules and services that encapsulate the configuration logic and provide access to the configuration values throughout your application.

Overall, configurations in Nest.js help you manage the behavior of your application in a flexible and modular way, enabling you to adapt it to different environments and deployment scenarios without modifying the underlying code.

Step 1: Install Dependencies

```
"@nestjs/config": "^2.3.2",
```

Nest.js provides built-in config package for configuration. We are going to use this package. The `@nestjs/config` package internally uses `dotenv`.

Step 2: Import ConfigModule in AppModule

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


@Module({
imports: [
ConfigModule.forRoot(),
})
```

You have to provide the path of your env file in the `ConfigModule.forRoot()` method. We will have two env files, one is for development and the other is for production level

Step 3: Creating Custom Env Files

You have to `.development.env` and `.production.env` files in the root directory

`.development.env`

```
PORT = 3000;
```

`.production.env`

```
PORT = 3000;
```

You also have to register your env files in AppModule.

```
ConfigModule.forRoot({ envFilePath: ['.development.env', '.production.env'] }),
```

Step 4: Make ConfigModule to global

```
ConfigModule.forRoot({
  isGlobal: true,
});
```

When you want to use `ConfigModule` in other modules, you'll need to import it (as is standard with any Nest module). Alternatively, declare it as a global module by setting the options object's `isGlobal` property to `true`, as shown below. In that case, you will not need to import `ConfigModule` in other modules once it's been loaded in the root module (e.g., `AppModule`)

Step 5: Creating CustomConfiguration.ts file

You have to create a new `config` folder inside the `src` directory and create new `configuration.ts` file there. You can create multiple custom configuration files for the database, app settings, and jwt, etc. A custom configuration file exports a factory function that returns a configuration object.

```
//config/configuration.ts
export default () => ({
  port: parseInt(process.env.PORT),
});
```

Step 6: Load Configuration File

```
ConfigModule.forRoot({
envFilePath: ['.development.env', '.production.env'],
isGlobal: true,
load: [configuration],
}),
```

We load this file using the load property of the options object we pass to the ConfigModule.forRoot() method. The value assigned to the load property is an array, allowing you to load multiple configuration files (e.g. `load: [databaseConfig, authConfig]`

Step 7: Test the env variable

```
//auth.service.ts
import { ConfigService } from "@nestjs/config";

export class AuthService {
  constructor(private configService: ConfigService) {}

  getEnvVariables() {
    return {
      port: this.configService.get<number>("port"),
    };
  }
}
```

You can inject `ConfigService` as a dependency in any service. I wanted to show you the usage of `ConfigService` in a service. To access configuration values from our `ConfigService`, we first need to inject `ConfigService`. As with any provider, we need to import it's containing module - the ConfigModule - into the module that will use it (unless you set the `isGlobal` property in the options object passed to the `ConfigModule.forRoot()` method to `true`).

We don't need to import the `ConfigModule` in `AuthModule` because we have made it global in the AppModule

You can get the environment variables by using the `get` function from `ConfigService`. I have provided the type `<number>` and the `key` of the variable

Let's create the test method in the `AuthController` to get the env variable. I have created this route for only testing purposes

```
//auth.controller.ts
@Get('test')
testEnv() {
return this.authService.getEnvVariables();
}
```

```
### TEST ENV VARIABLES
GET http://localhost:3001/auth/test
```

When you send a request to this URL you will get the port number in the response

Step 8: Use ConfigService in main.ts file

When you open the `main.ts` file you will see the manual port value which is `3000`. Let's get the value from `ConfigService` and add the port to the `listen` method. One more thing you can get the `ConfigService` instance from the app by calling `app. get(ConfigService)`

```
//main.ts
import { ConfigService } from "@nestjs/config";

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const configService = app.get(ConfigService); // get the instance of
ConfigService using app.get
  await app.listen(configService.get<number>("port"));
}
bootstrap();
```

Step 9: Add JWT secret in Configuration

```
.development.env
```

SECRET=HAD_12X#@

```
.production.env
```

SECRET=HAD_12X#@

You also have to add the new value in the `configuration.ts` file for the `secret`.

```
//configuration.ts
secret: process.env.SECRET,
```

Now we need to add the jwt secret inside the `AuthModule` while registering the `JwtModule`

```
//auth.module.ts
import { ConfigModule, ConfigService } from '@nestjs/config';
```

```
JwtModule.registerAsync({
imports: [ConfigModule],
useFactory: async (configService: ConfigService) => ({
secret: configService.get<string>('secret'),
signOptions: {
expiresIn: '1d',
},
}),
inject: [ConfigService],
}),
```

The `registerAsync` method will return the `DynamicModule`. In Nest.js, a dynamic module is a feature that allows you to dynamically configure and register modules at runtime based on dynamic conditions or external factors. It provides a way to encapsulate complex configuration logic and allows modules to be created and registered programmatically.

Dynamic modules are useful when you have modules that require some dynamic configuration or when you want to conditionally load modules based on runtime conditions or variables.

```
static registerAsync(options: JwtModuleAsyncOptions): DynamicModule;
```

Step 10: Setup DB Configuration

`.development.env`

```
# Database Configuration for Development
DB_HOST=localhost
DB_PORT=5432
USERNAME=postgres
PASSWORD=root
DB_NAME=spotify-clone
```

We have used database configuration manually, now we need to get all the configuration from the env file. We can have a separate db configuration for the development and production environment. You have to use your database configuration like dbName, username, and password

You have to refactor the `data-source.ts` file

```
//data-source.ts
import { ConfigModule, ConfigService } from "@nestjs/config";
import {
  TypeOrmModuleAsyncOptions,
  TypeOrmModuleOptions,
} from "@nestjs/typeorm";
```

```typescript
export const typeOrmAsyncConfig: TypeOrmModuleAsyncOptions = {
  imports: [ConfigModule],
  inject: [ConfigService],
  useFactory: async (
    configService: ConfigService
  ): Promise<TypeOrmModuleOptions> => {
    return {
      type: "postgres",
      host: configService.get<string>("dbHost"),
      port: configService.get<number>("dbPort"),
      username: configService.get<string>("username"),
      database: configService.get<string>("dbName"),
      password: configService.get<string>("password"),
      entities: ["dist/**/*.entity.js"],
      synchronize: false,
      migrations: ["dist/db/migrations/*.js"],
    };
  },
};

export const dataSourceOptions: DataSourceOptions = {
  type: "postgres",
  host: process.env.DB_HOST,
  port: parseInt(process.env.DB_PORT),
  username: process.env.USERNAME,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
  entities: ["dist/**/*.entity.js"],
  synchronize: false,
  migrations: ["dist/db/migrations/*.js"],
};
```

I have created a new object `typeOrmAsyncConfig` inside the `data-source.ts`

Let's refactor the TyepOrm Module registration in`AppModule`

```typescript
//app.module.ts
TypeOrmModule.forRootAsync(typeOrmAsyncConfig),
```

You have to add more properties in the `configuration.ts` file

```typescript
//configuration.ts
dbHost: process.env.DB_HOST,
dbPort: parseInt(process.env.DB_PORT),
username: process.env.USERNAME,
password: process.env.PASSWORD,
```

```
dbName: process.env.DB_NAME,
```

Application Configurations

Add validation for environment variables by utilizing the `class-validator` package in your Nest.js application. This ensures that the application doesn't start with incorrect or missing configurations, adhering to the "Fail-fast" principle in software engineering. Throw the validation error before the application starts to avoid runtime issues.

Step 1: Add Node_ENV variable in .env files

Implement the validation for `env` variables. I am going to add the new property `NODE_ENV=development` in the `.env.development` file and `NODE_ENV=production`. We have to validate these variables

```
# .env.development
NODE_ENV=development

# .env.production
NODE_ENV=production
```

Step 2: Create .env.validation.ts

You have to add validation logic in this file. I have created this file inside the root folder.

```
import { plainToInstance } from "class-transformer";
import { IsEnum, IsNumber, IsString, validateSync } from "class-validator";

enum Environment {
  Development = "development",
  Production = "production",
  Test = "test",
  Provision = "provision",
}

class EnvironmentVariables {
  // 1
  @IsEnum(Environment)
  NODE_ENV: Environment;

  @IsNumber()
  PORT: number;
```

```typescript
  @IsString()
  DB_HOST: string;

  @IsString()
  USERNAME: string;

  @IsString()
  PASSWORD: string;

  @IsString()
  DB_NAME: string;

  @IsString()
  SECRET: string;
}

export function validate(config: Record<string, unknown>) {
  //plainInstance converts plain (literal) object to class (constructor) object.
Also works with arrays.
  const validatedConfig = plainToInstance(EnvironmentVariables, config, {
    /**
     * enableImplicitConversion will tell class-transformer that if it sees a
primitive that is currently a string (like a boolean or a number) to assume it
should be the primitive type instead and transform it, even though @Type(() =>
Number) or @Type(() => Boolean) isn't used
     */
    enableImplicitConversion: true,
  });

  /**
   * Performs sync validation of the given object.
   * Note that this method completely ignores async validations.
   * If you want to properly perform validation you need to call validate method
instead.
   */
  const errors = validateSync(validatedConfig, {
    skipMissingProperties: false,
  });

  if (errors.length > 0) {
    throw new Error(errors.toString());
  }
  return validatedConfig;
}
```

1. We have added the validations for all our env variables. Whenever you need to create a new env variable in `.env.development` or `.env.production` file. You can create a validation rule inside the `EnvironmentVariables` class
2. `Class-transformer` allows you to transform a plain object to some instance of class and versa. Also, it allows the serialization/deserializing of objects based on criteria. This tool is super useful on both the front end and backend.
3. You can read the documentation of `class-transformer` here

Step 3: Add validate method in ConfigModule

You can add validate method in `ConfigModule` in `AppModule`

```
//app.module.ts

import { validate } from "env.validation";

ConfigModule.forRoot({
  validate: validate,
});
```

I have noticed that when you make changes in your application it will take too much time to reload the application we can make this process faster by using webpack HMR

Step 1: Create webpack-hmr.config.js

First of all, you have to install this dev dependency

```
npm install -D run-script-webpack-plugin
```

You have to create this file in the root directory

```
// eslint-disable-next-line @typescript-eslint/no-var-requires
const nodeExternals = require("webpack-node-externals");
// eslint-disable-next-line @typescript-eslint/no-var-requires
const { RunScriptWebpackPlugin } = require("run-script-webpack-plugin");

module.exports = function (options, webpack) {
  return {
    ...options,
    entry: ["webpack/hot/poll?100", options.entry],
    externals: [
      nodeExternals({
        allowlist: ["webpack/hot/poll?100"],
      }),
    ],
```

```
    plugins: [
      ...options.plugins,
      new webpack.HotModuleReplacementPlugin(),
      new webpack.WatchIgnorePlugin({
        paths: [/\.js$/, /\.d\.ts$/],
      }),
      new RunScriptWebpackPlugin({
        name: options.output.filename,
        autoRestart: false,
      }),
    ],
  };
};
```

Step 2: Refactor bootstrap function in main.ts

```
declare const module: any;

async function bootstrap() {
  ////....
  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}
```

Step 3: Refactor start:dev script in package.json

```
"start:dev": "nest build --webpack --webpackPath webpack-hmr.config.js --watch",
```

You have to specificy that you want to reload the application using webpack. Now you test the application by making some changes and your application reload time would be faster

Chapter 9 API Documentation with Swagger

What is Swagger?

The components of Swagger include:

1. **Swagger Specification:** This forms the blueprint of the API, documenting its structure in a format that can be both human and machine interpreted. NestJS supports Swagger by offering packages such as `@nestjs/swagger` to auto-generate specifications from code,

which aligns with best practices by ensuring that the documentation remains synchronized with the codebase.

2. **Swagger Editor:** This browser-based editor allows for the crafting and fine-tuning of Swagger specifications. NestJS users can import their auto-generated Swagger configuration into the Swagger Editor to make further refinements or visualize changes, streamlining API design iterations.

3. **Swagger UI:** Interactive documentation created by Swagger UI facilitates clear communication and understanding of API functionalities. NestJS leverages Swagger UI to enable developers and API consumers to interact with the API endpoints directly from the browser, encouraging a hands-on approach to exploring API capabilities.

4. **Swagger Codegen:** This tool provides the means to scaffold server structures and client-side code from the API specifications. Within the NestJS framework, Swagger Codegen can expedite development by auto-creating API-related code, thus embodying a best practice of reducing manual coding and potential human error.

Employing Swagger within NestJS promotes a systematic, well-documented, and collaborative API development process, reflecting best practices and serving as a foundation for robust software architecture.

Step 1: Install Dependencies

```
"@nestjs/swagger": "^6.3.0",
```

The Nest.js built-in package will be utilized to implement API documentation with Swagger. This integration facilitates the automatic generation of interactive API documentation, which is a best practice for maintaining clear and structured endpoint documentation for developers and users alike.

Step 2: Configure SwaggerModule in bootstrap function

SwaggerModule configuration takes place within the bootstrap function. This NestJS-specific module automatically generates interactive API documentation, and as a best practice, it's advised to configure Swagger in development environments to aid in API design and testing without compromising the production environment's security.

```
async function bootstrap() {
  app.useGlobalPipes(new ValidationPipe());
  ///......
  //Configure the swagger module here
  const config = new DocumentBuilder() //1
    .setTitle("Spotify Clone")
    .setDescription("The Spotify Clone Api documentation")
    .setVersion("1.0")
    .build();

  const document = SwaggerModule.createDocument(app, config); //2
  SwaggerModule.setup("api", app, document); //3
}
```

1. The `DocumentBuilder` is utilized to configure the title, description, and version of the API documentation.
2. This document is then created with the help of the SwaggerModule, which is specific to NestJS for API design and testing.
3. Subsequently, the Swagger document is hosted at the `/api` endpoint, making it accessible via `http://localhost:3000/api`, offering a visual interface for interacting with the API. As a best practice, maintaining up-to-date and comprehensive Swagger documentation ensures that APIs are understandable and usable, aiding in both development and API consumption.

Step 1: Add auth Tags in the Auth Controller

In the initial step, authentication tags are added to the Auth Controller. This labeling within NestJS facilitates Swagger documentation generation, which helps to categorically group and distinguish authentication endpoints. It is considered a best practice to annotate controllers with appropriate Swagger tags to enhance API documentation and maintain clarity for developers interfacing with the backend services.

```
@ApiTags("auth")
export class AuthController {}
```

Now you will see all the auth routes in the `auth` section

Step 2: Add Api Operation and Response for the Signup flow

In Step 2, ApiOperation and ApiResponse annotations are added for the signup process. These annotations, part of the Swagger module in NestJS, facilitate API documentation by describing the operation and its possible responses, enhancing understandability for developers and end-users alike. It is considered good practice to thoroughly document API endpoints with such annotations, as this can significantly improve the development experience and future maintenance.

```
@ApiOperation({ summary: 'Register new user' })
@ApiResponse({
  status: 201,
  description: 'It will return the user in the response',
})
signup(){}
```

The `@ApiOperation` decorator instructs Swagger to generate documentation for a particular endpoint, enriching the API's interactive exploration interface. Meanwhile, `@ApiResponse` defines the expected response for an endpoint, including the status code, which improves clarity and client-side handling expectations. NestJS's integration with Swagger simplifies API documentation and it's considered best practice to use these decorators to provide clear, self-documenting API endpoints that align with OpenAPI specifications.

Step 3: Refactor the TypeORM config in data-source.ts

```
export const typeOrmAsyncConfig: TypeOrmModuleAsyncOptions = {
  entities: [User, Playlist, Artist, Song],
};
```

The syntax `entities: ['dist/**/*.entity.js']` has been utilized for entity registration, yet it is incompatible with Webpack hot module reloading, necessitating manual entity registration within the `typeOrmAsyncConfig` object. As a best practice, specifying each entity class directly in the TypeORM configuration ensures clarity and reliability, especially during the development phase when hot reloading is frequently used.

Show User Schema

Step 1: Add @ApiProperty in the User Entity

In the User Entity, the `@ApiProperty` decorator is added to enhance Swagger documentation automatically. This inclusion is a NestJS-specific feature that aids in generating interactive API documentation, and it exemplifies the practice of incorporating documentation as a part of the coding process to maintain clarity and up-to-date API information for developers.

```
export class User {
  @ApiProperty({
    example: "Jane",
    description: "Provide the first name of the user",
  })
  @Column()
  firstName: string;

  @ApiProperty({
    example: "Doe",
    description: "provide the lastName of the user",
  })
  @Column()
  lastName: string;

  @ApiProperty({
    example: "jane_doe@gmail.com",
    description: "Provide the email of the user",
  })
  @Column({ unique: true })
  email: string;
```

```
  @ApiProperty({
    example: "test123#@",
    description: "Provide the password of the user",
  })
  @Column()
  @Exclude()
  password: string;
}
```

The `@ApiProperty` decorator enhances swagger documentation by explicitly declaring the user schema. Utilizing this decorator is considered a best practice within NestJS for API documentation, as it provides clear and interactive API endpoints for testing and inspection.

Step 2: Register the Swagger plugin in nest-cli.json

To register the Swagger plugin, it is specified in the `nest-cli.json` file. This NestJS-specific step integrates Swagger automatically, generating API documentation and providing interactive testing utilities. Employing this plugin is a recommended practice as it promotes standardized API documentation and simplifies developer onboarding and frontend integration efforts.

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src",
  "compilerOptions": {
    "deleteOutDir": true,
    "plugins": [
      {
        "name": "@nestjs/swagger",
        "options": {
          "introspectComments": true
        }
      }
    ]
  }
}
```

The plugin array must be created, followed by the registration of the `@nestjs/swagger` package as a plugin. This step is crucial for enabling Swagger documentation in a NestJS application, a best practice for automatically generating interactive API documentation that enhances developer experience and API usability.

User Authentication

Step 1: Add API Operation for Login

The addition of an API operation for login is achieved through creating an authentication controller with a dedicated login route. Implementing this within NestJS typically involves using Guards and Strategies, leveraging Passport.js under the hood for a robust and secure authentication process. As a best practice, it is advisable to use environment variables for sensitive information such as secret keys and to apply validation pipelines to ensure the integrity of user input before processing authentication.

```
@ApiOperation({ summary: 'Login user' })
@ApiResponse({
  status: 200,
  description: 'It will give you the access_token in the response',
})
```

Step 2: Enable Bearer Auth

```
const config = new DocumentBuilder()
  .setTitle("Spotify Clone")
  .setDescription("The Spotify Clone Api documentation")
  .setVersion("1.0")
  .addBearerAuth(
    // Enable Bearer Auth here
    {
      type: "http",
      scheme: "bearer",
      bearerFormat: "JWT",
      name: "JWT",
      description: "Enter JWT token",
      in: "header",
    },
    "JWT-auth" // We will use this Bearer Auth with the JWT-auth name on the
controller function
  )
  .build();
```

Step 3: Update Secret Key in JWTStrategy

In the JWTStrategy, the secret key must be updated to ensure the security of token verification. NestJS recommends encapsulating such sensitive information within environment variables or configuration services to maintain a secure and scalable codebase. Best practices suggest the use of a robust secret management system to handle secrets, which aids in avoiding hard-coded credentials within the application code.

```
export class JWTStrategy extends PassportStrategy(Strategy) {
```

```
constructor() {
  // super();
  super({
    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
    ignoreExpiration: false,
    secretOrKey: process.env.SECRET, //1
  });
}
}
```

The decision has been made to not utilize the authconstants file. The secret will be retrieved from the env file, hence the use of `process.env.SECRET` is observed. NestJS recommends managing configuration in a robust and accessible manner, and utilizing environment variables through the ConfigModule or custom configuration services aligns with this best practice, ensuring sensitive information is securely managed.

Step 3: Apply ApiBearerAuth on the protected Route

```
//app.controller.ts
@Get('profile')
@ApiBearerAuth('JWT-auth')
@UseGuards(JwtAuthGuard)
getProfile(
  @Request()
  req,
) {
  return req.user;
}
```

The `@ApiBearerAuth` decorator has been applied to `getProfile` within `AppController`, designating it as a protected route that returns the user profile in the response. In NestJS, securing routes with decorators aligns with the framework's philosophy of declarative programming, enhancing readability and security - a practice in line with advanced software design principles.

Chapter 10 MongoDB Database

Install MongoDB using Docker Compose

MongoDB serves as an open-source NoSQL database management program, particularly adept at handling extensive sets of distributed data. Learning to integrate MongoDB within a Nest.js application involves using the Mongoose package, which is recommended for its robust modeling and validation tools that are absent in the native driver.

Mongoose operates as an object-oriented JavaScript library that facilitates a connection between MongoDB and the Node.js runtime environment. It simplifies the interaction with MongoDB by providing schema validation and the ability to translate between objects in code and their representation within MongoDB, which is a best practice for maintaining data integrity and consistency.

Step 1: Create a new Project

Open your terminal and create a new project with nest-cli

```
nest new project-name
```

Choose any name for your project.

Step 2: Install Dependencies

Install these two packages to connect with MongoDB database

```
npm install @nestjs/mongoose mongoose
```

Step 3: Setup MongoDB using Docker Compose

Docker Compose is utilized to install MongoDB, streamlining the setup process. Should Docker be unavailable, the MongoDB driver requires manual installation on the machine. It is recommended to use containerization with Docker for database services in development environments, as it offers consistency across different systems and can be integrated seamlessly with NestJS applications.

You have to create a `docker-compose.yml` file in your root project directory

```
version: "3"
services:
mongodb:
image: mongo:latest
environment:
  - MONGODB_DATABASE="test"
ports:
  - 27017:27017
```

You start the MongoDB server by opening the terminal in your root directory and run:

```
docker-compose up
```

You can stop the MongoDB driver using this command:

```
docker-compose down
```

Step 4: Connect MongoDB Database using GUI Tool

MongoDB Compass is utilized to interact with the MongoDB database. Installation is necessary for those who do not currently have it. Download MongoDB Compass

Connect with MongoDB

Step 1: Create Mongoose Module in the AppModule

The previous lesson involved the installation of two packages, `@nestjs/mongoose` and `mongoose`, for MongoDB integration. The creation of a Mongoose Module within AppModule follows, demonstrating NestJS's module-driven architecture which encapsulates functionality. It is considered a best practice to define schema models and their corresponding modules in separate files to enhance modularity and maintain a clear project structure.

```typescript
//app.module.ts
@Module({
  imports: [MongooseModule.forRoot("mongodb://localhost:27017/spotify-clone")],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

The `forRoot()` method in NestJS's Mongoose module requires a configuration object analogous to the one used in `mongoose.connect()` from the Mongoose package. This method provides a streamlined approach to configuring the database connection at the root module level, ensuring all sub-modules can interact with the database consistently. A best practice entails validating the database connection parameters and handling connection errors gracefully to maintain the stability of the application.

Create Schema

In Mongoose, everything commences with a Schema, which maps to a MongoDB collection and dictates the structure of the documents in that collection. Schemas serve to establish Models, with Models being accountable for the creation and retrieval of documents from the MongoDB database. Adopting Schemas and Models aligns with NestJS's modular approach, enhancing maintainability and promoting adherence to an application's defined data architecture. Utilizing Mongoose with NestJS adheres to a best practice of encapsulating database interactions, which improves code reusability and testability.

Step 1: Create a Schema

Create a `songs` folder and create a `schemas` folder inside the songs. This structure, recommended in NestJS, promotes a modular architecture by encapsulating schema definitions within their respective domain context, which aligns with domain-driven design principles for maintainable code organization.

```
//src/songs/schemas/song.ts
import { Prop, Schema, SchemaFactory } from "@nestjs/mongoose";
import { HydratedDocument } from "mongoose";

export type SongDocument = HydratedDocument<Song>;

@Schema()
export class Song {
  @Prop({
    required: true,
  })
  title: string;

  @Prop({
    required: true,
  })
  releasedDate: Date;

  @Prop({
    required: true,
  })
  duration: string;

  lyrics: string;
}

export const SongSchema = SchemaFactory.createForClass(Song);
```

1. The `SongDocument` is utilized upon injecting the `Model` into the `SongService`. It is a NestJS-specific approach to apply TypeScript interfaces for Mongoose models, promoting type safety and IntelliSense in the service layer.
2. Applying the `@Schema()` decorator designates a class as a schema definition, associating the `Song` class with a MongoDB collection named `songs`. This decorator is part of NestJS's Mongoose integration, which simplifies working with MongoDB by automatically pluralizing the model name for the collection.
3. The `@Prop()` decorator is employed to declare a property within the document. This decorator is crucial in defining the schema's data structure and ensuring the fields align with the intended types in the MongoDB collection.

4. SchemaFactory is tasked with generating the bare schema definition. Employing `console.log` on `SongSchema` will reveal the structured outcome, demonstrating the schema's conversion to a format that Mongoose can use to enforce document structure in MongoDB.

```
Schema {
  obj: {
    title: { required: true, type: [Function: String] },
    releasedDate: { required: true, type: [Function: Date] },
    duration: { required: true, type: [Function: String] }
  },
  paths: {
    title: SchemaString {
      enumValues: [],
      regExp: null,
      path: 'title',
      instance: 'String',
      validators: [Array],
      getters: [],
      setters: [],
      _presplitPath: [Array],
      options: [SchemaStringOptions],
      _index: null,
      isRequired: true,
      requiredValidator: [Function (anonymous)],
      originalRequiredValue: true,
      [Symbol(mongoose#schemaType)]: true
    },
    releasedDate: SchemaDate {
      path: 'releasedDate',
      instance: 'Date',
      validators: [Array],
      getters: [],
      setters: [],
      _presplitPath: [Array],
      options: [SchemaDateOptions],
      _index: null,
      isRequired: true,
      requiredValidator: [Function (anonymous)],
      originalRequiredValue: true,
      [Symbol(mongoose#schemaType)]: true
    },
    duration: SchemaString {
      enumValues: [],
      regExp: null,
      path: 'duration',
```

```
      instance: 'String',
      validators: [Array],
      getters: [],
      setters: [],
      _presplitPath: [Array],
      options: [SchemaStringOptions],
      _index: null,
      isRequired: true,
      requiredValidator: [Function (anonymous)],
      originalRequiredValue: true,
      [Symbol(mongoose#schemaType)]: true
    },
    _id: ObjectId {
      path: '_id',
      instance: 'ObjectId',
      validators: [],
      getters: [],
      setters: [Array],
      _presplitPath: [Array],
      options: [SchemaObjectIdOptions],
      _index: null,
      defaultValue: [Function],
      [Symbol(mongoose#schemaType)]: true
    }
  },
  aliases: {},
  subpaths: {},
  virtuals: {},
  singleNestedPaths: {},
  nested: {},
  inherits: {},
  callQueue: [],
  _indexes: [],
  methods: {},
  methodOptions: {},
  statics: {},
  tree: {
  title: { required: true, type: [Function: String] },
  releasedDate: { required: true, type: [Function: Date] },
  duration: { required: true, type: [Function: String] },
  _id: { auto: true, type: 'ObjectId' }
  },
  query: {},
  childSchemas: [],
  plugins: [],
```

```
  '$id': 1,
  mapPaths: [],
  s: { hooks: Kareem { _pres: Map(0) {}, _posts: Map(0) {} } },
  _userProvidedOptions: {},
  options: {
    typeKey: 'type',
    id: true,
    _id: true,
    validateBeforeSave: true,
    read: null,
    shardKey: null,
    discriminatorKey: '__t',
    autoIndex: null,
    minimize: true,
    optimisticConcurrency: false,
    versionKey: '__v',
    capped: false,
    bufferCommands: true,
    strictQuery: false,
    strict: true
  }
}
```

Save Record in MongoDB

The record must be saved in the MongoDB collection, necessitating the creation of a `POST` endpoint to store the songs. Utilizing NestJS's @Controller and @Post decorators, the endpoint can be efficiently set up, showcasing the framework's streamlined approach to REST API development. It is considered a best practice to abstract the interaction with MongoDB into a service, which allows for cleaner controllers and easier maintenance of database operations.

Step 1: Create Songs Module

```
nest g module songs
```

Step 2: Create Songs Controller

```
nest g controller songs
```

Step 3: Create Songs Service

```
nest g service songs
```

Step 4: Add a create method in SongsController

```typescript
import { Body, Controller, Post } from "@nestjs/common";
import { CreateSongDTO } from "./dto/create-song-dto";
import { SongsService } from "./songs.service";

@Controller("songs")
export class SongsController {
  constructor(private songService: SongsService) {}
  @Post()
  create(
    @Body()
    createSongDTO: CreateSongDTO
  ) {
    return this.songService.create(createSongDTO);
  }
}
```

Step 5: Create CreateSongDTO

```typescript
// songs/dto/create-song-dto.ts
export class CreateSongDTO {
  title: string;
  releasedDate: Date;
  duration: Date;
  lyrics: string;
}
```

Step 6: Add a create method in SongsService

```typescript
import { Injectable } from "@nestjs/common";
import { InjectModel } from "@nestjs/mongoose";
import { Song, SongDocument } from "./schemas/song.schema";
import { Model } from "mongoose";
import { CreateSongDTO } from "./dto/create-song-dto";

@Injectable()
export class SongsService {
  constructor(
    @InjectModel(Song.name) //1
    private readonly songModel: Model<SongDocument> //2
  ) {}
```

```
  async create(createSongDTO: CreateSongDTO): Promise<Song> {
    const song = await this.songModel.create(createSongDTO); //3.
    return song;
  }
}
```

1. Registration of the song model within the `SongsModule` allows for its injection into the service, demonstrating NestJS's modular design that promotes loose coupling and high cohesion.
2. Utilization of the `SongDocument` type within the song schema file ensures that the object adheres to the defined schema, a practice that enforces type safety and reduces runtime errors.
3. The `songModel` incorporates a method to persist records in MongoDB, which illustrates the encapsulation of database operations within models, a practice that enhances maintainability and scalability of the application.

Step 7: Register the Song Model in SongsModule

```
@Module({
  imports: [
    MongooseModule.forFeature([{ name: Song.name, schema: SongSchema }]), //1
  ],
  controllers: [SongsController],
  providers: [SongsService],
})
export class SongsModule {}
```

The MongooseModule employs the `forFeature()` method to configure itself, allowing specific models to be registered within the current scope. It is a NestJS-specific mechanism that ensures model encapsulation and modularity, a recommended approach for maintaining clean and manageable database-related code.

Step 8: Test the Application

The application's functionality can tested. See if it works.

```
POST http://localhost:3000/songs
Content-Type: application/json

{
"title": "Lasting Lover",
"releasedDate" : "2023-05-11",
"duration" :"02:33",
"lyrics": "I don't know why I can't quite get you out my sight You're always just
behind"
```

```
}
```

Find and Delete Record

Step 1: Create a new find method in SongService

A new find method is crafted within the SongService, encapsulating the logic for retrieving song data. As a best practice within NestJS, this method should be designed as a service that can be injected into controllers, promoting a clean separation of concerns and enhanced testability.

```
//songs.service.ts
async find(): Promise<Song[]> {
  return this.songModel.find();
}
```

Step 2: Create a Route in SongController

A route in the SongController is established to handle specific music-related requests. In NestJS, it's advisable to use decorators like @Get, @Post, or @Put to clearly define the purpose and nature of the route, enhancing the readability and structure of the code.

```
//songs.controller.ts
@Get()
find(): Promise<Song[]> {
  return this.songService.find();
}
```

Now test this route by starting the application and sending the request at GET http://localhost:3000/songs

Step 3: Create a findById method in SongService

The creation of a `findById` method within `SongService` serves to encapsulate the logic for retrieving a specific song by its identifier. Implementing such methods aligns with NestJS's philosophy of modular services, ensuring that each service has a single responsibility and that the application remains scalable and maintainable. It is considered a best practice to abstract database queries within services to isolate them from controllers, thereby promoting clean separation of concerns.

```
async findById(id: string): Promise<Song> {
  return this.songModel.findById(id);
}
```

Step 4: Create findOne Route in the Controller

The creation of a `findOne` route in the controller is implemented to facilitate the retrieval of a single song entity by its unique identifier. In NestJS, best practices suggest utilizing decorators like `@Get` with the route path and `@Param` to capture route parameters, enhancing the modularity and declarative nature of routing mechanisms.

```
@Get(':id')
findOne(
  @Param('id')
  id: string,
):
Promise<Song> {
  return this.songService.findById(id);
}
```

Step 5: Create a delete song method in SongService

A delete song method within SongService can be established to handle removal operations for songs. Ensuring the method is idempotent, meaning it can be called multiple times without changing the result beyond the initial application, is considered a best practice for robust API design in NestJS applications.

```
async delete(id: string) {
  return this.songModel.deleteOne({ _id: id });
}
```

Step 6: Create a Route for deleting a song

A route for deleting a song is established through a controller's method decorated with NestJS's `@Delete()` decorator, which maps HTTP DELETE requests to the corresponding service function. In terms of best practices, implementing soft deletion, where records are flagged as inactive rather than removed from the database, can be advantageous for data recovery and audit purposes.

```
@Delete(':id')
delete(
  @Param('id')
  id: string,
) {
  return this.songService.delete(id);
}
```

Populate

This lesson covers the addition of relations between two models in Mongoose, illustrating the association where each song is linked to a single album and each album may encompass numerous songs. The demonstration focuses on Mongoose's `populate` feature, which is pivotal for managing document relationships in MongoDB, akin to JOINs in relational databases. NestJS supports Mongoose's features directly through its dedicated `@nestjs/mongoose` package, and employing `populate` is a common practice to efficiently retrieve related documents.

Step 1: Create an album schema

A new folder named `albums` must be created to house the schema file. This action conforms to NestJS's modular architecture, where separating concerns by feature enhances maintainability and scalability—a best practice in software development.

```typescript
import { Prop, Schema, SchemaFactory } from "@nestjs/mongoose";
import { HydratedDocument, Schema as MongooseSchema, Types } from "mongoose";
import { Song } from "src/songs/schemas/song.schema";

export type AlbumDocument = HydratedDocument<Album>;

@Schema()
export class Album {
  @Prop({
    required: true,
  })
  title: string;

  @Prop({ type: [Types.ObjectId], ref: "songs" }) //1
  songs: Song[];
}

export const AlbumSchema = SchemaFactory.createForClass(Album);
```

1. The `songs` property is defined in the Album model with the type specified as an array of MongoDB ObjectIDs, intended to store references to the IDs of the song collection within the albums table.
2. The `ref` attribute establishes a reference to the `songs` collection, enabling the creation of relational data structures within a MongoDB database, a feature NestJS leverages through its Mongoose module integration.

As a best practice, it is recommended to clearly document such relationships within the code to improve maintainability and provide clarity for future development efforts. Additionally, ensuring that the ObjectIDs used in references are validated for their format can prevent runtime errors and maintain data integrity.

Step 2: Add a relation in the song schema

Inclusion of a relation within the song schema can be executed through the use of decorators that define the relationship type, such as @OneToMany or @ManyToOne, which NestJS leverages from TypeORM. This practice encapsulates the relational aspect directly within the entity, thus promoting a clear and maintainable structure within the application's domain model. Best practice dictates careful planning of entity relationships to optimize query performance and database integrity.

```typescript
// song.schema.ts
@Prop({
  type: Types.ObjectId,
  ref: Album.name,
})
album: Album;
```

Step 3: Create Album Module, Controller and Service

The creation of an Album Module, Controller, and Service in a NestJS application encapsulates the album-related functionalities, aligning with the framework's modular architecture. Implementing each as a separate entity follows NestJS's single-responsibility principle, ensuring that the application remains scalable and maintainable. As a best practice, defining strict interfaces and DTOs (Data Transfer Objects) for service methods enhances type safety and validation, which is a crucial aspect of robust software design.

```typescript
import { Module } from "@nestjs/common";
import { AlbumController } from "./album.controller";
import { AlbumService } from "./album.service";
import { MongooseModule } from "@nestjs/mongoose";
import { Album, AlbumSchema } from "./schemas/album.schema";

@Module({
  imports: [
    MongooseModule.forFeature([{ name: Album.name, schema: AlbumSchema }]),
  ],
  controllers: [AlbumController],
  providers: [AlbumService],
})
export class AlbumModule {}

import { Injectable } from "@nestjs/common";
import { Album, AlbumDocument } from "./schemas/album.schema";
import { Model } from "mongoose";
import { InjectModel } from "@nestjs/mongoose";
import { CreateAlbumDTO } from "./dto/create-album-dto";
import { Song } from "src/songs/schemas/song.schema";
```

```
@Injectable()
export class AlbumService {
  constructor(
    @InjectModel(Album.name)
    private readonly albumModel: Model<AlbumDocument>
  ) {}
  async createAlbum(createAlbumDTO: CreateAlbumDTO): Promise<Album> {
    return this.albumModel.create(createAlbumDTO);
  }

  async findAlbums() {
    return this.albumModel.find().populate("songs", null, Song.name); //1
  }
}
```

The `populate` method is applied on the album model to retrieve all songs associated with each album. Utilizing this feature in NestJS effectively allows for eager loading of related entities, which streamlines data retrieval processes. It is considered a good practice to carefully manage such operations to optimize query performance and maintain data integrity.

```
//create-album-dto.ts
export class CreateAlbumDTO {
  title: string;
  songs: string[];
}
```

```
//album.controller.ts

import { Body, Controller, Get, Post } from "@nestjs/common";
import { Album } from "./schemas/album.schema";
import { AlbumService } from "./album.service";
import { CreateAlbumDTO } from "./dto/create-album-dto";

@Controller("albums")
export class AlbumController {
  constructor(private albumService: AlbumService) {}
  @Post()
  create(
    @Body()
    createAlbumDTO: CreateAlbumDTO
  ): Promise<Album> {
    return this.albumService.createAlbum(createAlbumDTO);
  }

  @Get()
  find(): Promise<Album[]> {
```

```
    return this.albumService.findAlbums();
  }
}
```

Refactor the CreateSongDTO

```
//create-song-dto.ts
album: string;
```

Chapter 11 Deploy Nest.js Application

Configure Development and Production Environment

This lesson covers the deployment of a Nest.js project to Railway, detailing the necessary configurations. It requires setting up distinct `.env` files for development and production environments. Nest.js best practices include using environment-specific configuration files to manage different settings for development, testing, and production, ensuring sensitive data is not exposed. Proper management of environment variables is crucial for security and flexibility in different deployment scenarios.

Step 1: Create New Account at the Railway

Create a new account at Railway.

Step 2: Create a new Project

After creating an account at Railway you have to create a new project, connect your Github account with Railway

Step 3: add NODE_ENV in configuration file

The `NODE_ENV` has not been included in `configuration.ts`. Including it would enable access to `NODE_ENV` through `ConfigService`, which is a best practice in NestJS for maintaining environment-based configurations, thereby facilitating easier management and scalability of the application's settings.

```
//configuration.ts
NODE_ENV: process.env.NODE_ENV;
```

Step 4: Refactor the envFile path in AppModule

A separate environment file is required for each development stage, necessitating the addition of a dynamic environment file path. NestJS supports environment-specific configuration through its configuration module, a best practice that ensures the separation of concerns and facilitates different settings for development, staging, and production environments.

```
//app.module.ts
ConfigModule.forRoot({
  envFilePath: [`${process.cwd()}/.env.${process.env.NODE_ENV}`],
  isGlobal: true,
  load: [configuration],
  validate: validate,
}),
```

`process.cwd()` gives you the current working directory

Step 4: Add NODE_ENV for development and production script

Set the `NODE_ENV` configuration in package.json file

```
"start:dev": "NODE_ENV=development nest build --webpack --webpackPath
webpack-hmr.config.js --watch",
"start:prod": "NODE_ENV=production node dist/main",
```

Step 5: Test the Application

The application is executable by initiating the project in either development or production mode. In NestJS, this is typically managed by environment-specific configuration files, which is a best practice for maintaining separation between development and production settings, ensuring that environment variables and settings are correctly applied for each scenario.

Pushing your source code

In this lesson, the focus is on pushing the source code to GitHub and establishing a connection between the GitHub repository and the Railway project. As a best practice in NestJS development, it's recommended to include a `.gitignore` file to prevent the versioning of environment-specific files and node_modules, thus ensuring the repository remains clean and manageable. Additionally, integrating continuous integration tools can streamline deployment workflows, a strategy aligned with high-quality software development processes.

Step 1: Create a GitHub Repository

create a new GitHub repository and push the source code to that repository

```
git remote add origin <Your repo Link>
git branch -M main
git push -u origin main
```

Don't push the files: `.env`, `.env.development`, `.env.production` to your GitHub repository. Add this as a restriction in the `.gitignore`

```
.env
.env.development
.env.production
```

Deploy Nest.js Project

The deployment of a Nest.js project to Railway requires configuring the project's environment variables and settings within the Railway platform. Ensuring that the database and other services align with the Railway deployment specifications is a critical step for smooth operation, which aligns with best practices for cloud deployment.

Step 1: Connect your GitHub Repo to Railway Project

A new account has been created at Railway, and it requires the selection of a GitHub repository. Connecting the repository is necessary at this stage.

Upon selecting the repository, the application deployment will commence.

Additionally, the application status can be monitored by checking the logs.

Step 2: Create a Database

Upon navigating to the project on Railway, a new database can be created with a simple right-click on the project dashboard; the Postgres Database must be selected.

Step 3: Copy the Database Configurations

The database has been created successfully. The database credentials can be obtained by visiting the connection tab.

The deployment of a Nest.js project to Railway requires configuring the project's environment variables and settings within the Railway platform. Ensuring that the database and other services align with the Railway deployment specifications is a critical step for smooth operation, which aligns with best practices for cloud deployment.

Step 1: Connect your GitHub Repo to Railway Project

A new account has been created at Railway, and it requires the selection of a GitHub repository. Connecting the repository is necessary at this stage.

Upon selecting the repository, the application deployment will commence.

Additionally, the application status can be monitored by checking the logs.

Step 2: Create a Database

Upon navigating to the project on Railway, a new database can be created with a simple right-click on the project dashboard; the Postgres Database must be selected.

Step 3: Copy the Database Configurations

The database has been created successfully. The database credentials can be obtained by visiting the connection tab.


```
DB_HOST=containers-us-west-89.railway.app

DB_PORT=5673

USERNAME=postgres
```

```
PASSWORD=PaX7MXpvWogZoxvcunkL

DB_NAME=railway
```

You have to copy your configurations and save it to `.env.production` file

Step 4: Set the Environment Variables for your Project

Now you have to go to your Nest.js project and you will find the `Variable` tab here
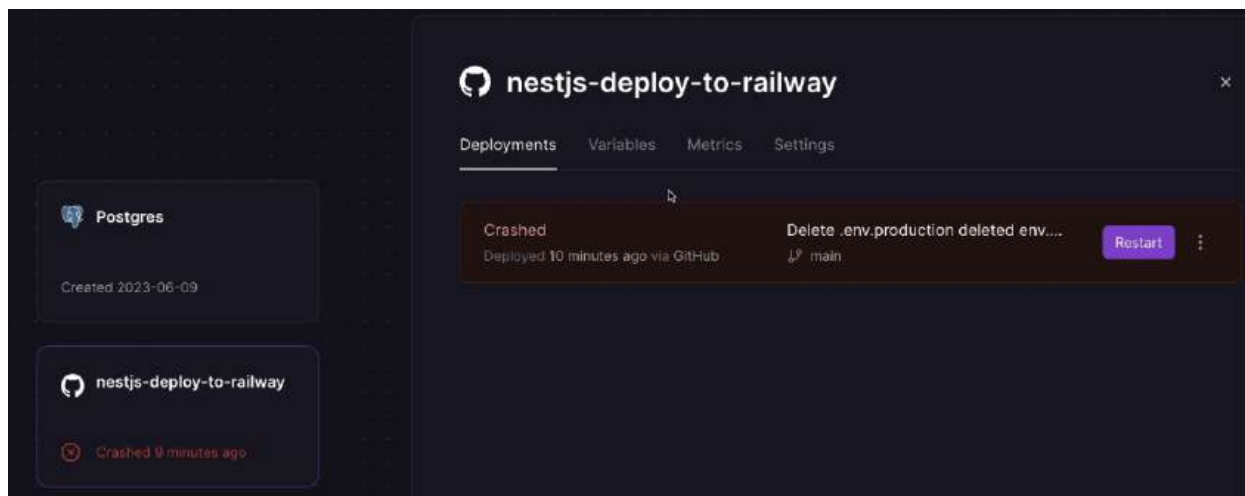
Raw Editor

Add, edit, or delete your project variables

ENV	JSON

```
PORT=3000
SECRET=HAD_12X#@

DB_HOST=containers-us-west-89.railway.app
DB_PORT=5673
USERNAME=postgres
PASSWORD=PaX7MXpvWogZoxvcunkL|          I
DB_NAME=railway
```

⟆ Copy ENV   Cancel   Update Variables

You have to copy your `.env.production` and save it here

Now it is going to start deploying the project and your project will be running successfully

Step 5: Test the Application

Now you can test the application by sending the signup request from your live api
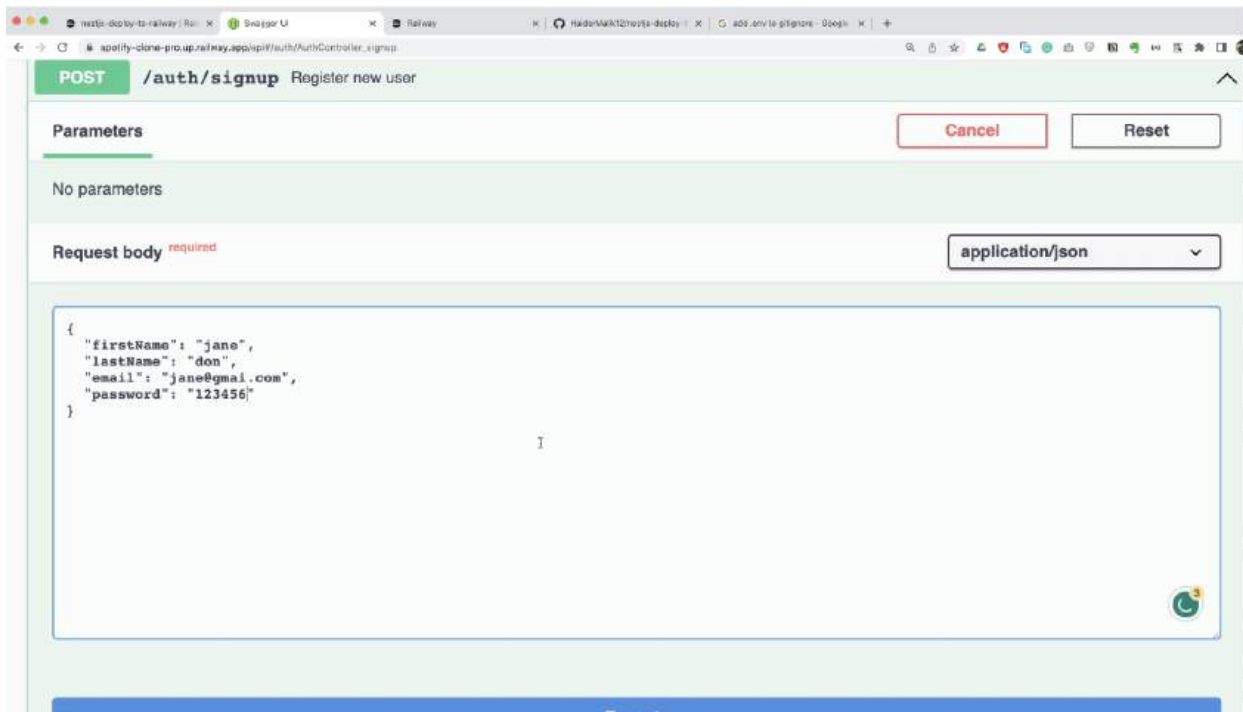
You will see this error
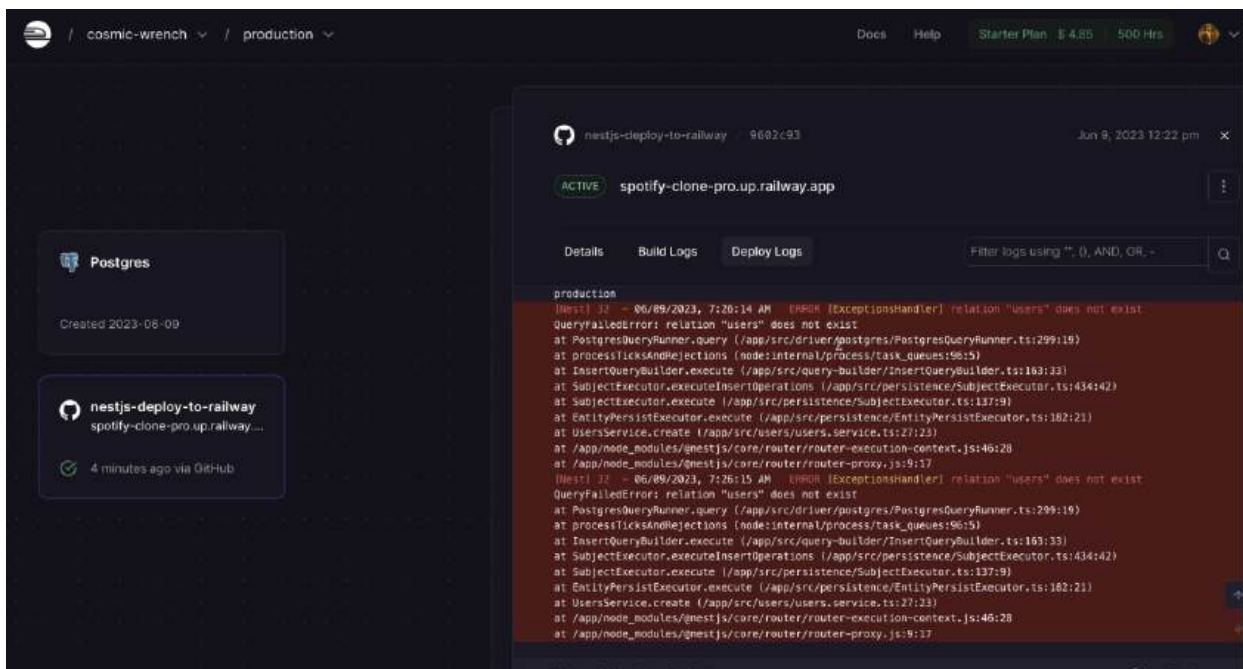
It means there is no users table in your Railway Postgres database. We have to migrate your database to the production

In the next lesson, you are going to learn how to migrate the database in the production

Install DotEnv

In the previous video, we got this issue

Step 1: Run the Migration

You will get this error when you run the migration

Step 2: Set the Environment to Production using the export command

You can set the `NODE_ENV` to `production` to run the build command for the production database

Step 3: Run the migration again

When you run the migration again after setting up production `NODE_ENV` you will get the undefined env variables in the `data-source.ts` file. Now the `NODE_ENV` is the production, you can see below

Step 4: Install the dotenv package

The issue has been identified within the `data-source.ts` file, which fails to retrieve values from the `.env.production` file. Installation of the `dotenv` package in the application is required to ensure environment variables are managed and loaded correctly, which aligns with best practices for maintaining a secure and configurable codebase.

```
npm install dotenv
```

Step 5: Run the migration again to test it

When the migration is executed again, the same error persists. The forthcoming lesson will address the resolution of this error, highlighting a best practice which entails examining and refining the migration scripts to ensure idempotency, thereby preventing repetitive failure upon re-execution.

```
> npm run build && npx typeorm -d dist/db/data-source.js "migration:generate" "db/migrations/init"


> n-fundamentals-pro@0.0.1 build
> nest build

production
undefined
undefined
Error during migration generation:
```

Fixing Env Bugs

Step 1: Create a .env file

Attempting to resolve the issue by specifying a custom path for the configuration may not yield the desired results. This approach, when unsuccessful, suggests that the problem lies elsewhere, potentially in the environment setup or file referencing.

```
import path from 'path';
const envPath = path.resolve(__dirname, `.env.${process.env.NODE_ENV}`);
require('dotenv').config({ path: path.resolve(__dirname, '../.env') });
```

Running the migration with the current setup continues to result in an error, suggesting that the env variables in `data-source.ts` are not being correctly defined or accessed. A best practice in such cases is to ensure that environment variables are being loaded properly, often by using a configuration management library or by verifying the path and export mechanism of the environment configuration file.

```
# USE IT FOR TYPEORM MIGRATION
PORT=3000
SECRET=HAD_12X#@

DB_HOST=containers-us-west-89.railway.app
DB_PORT=5673
USERNAME=postgres
PASSWORD=PaX7MXpvWogZoxvcunkL
```

```
DB_NAME=railway
```

You have to create a new `.env` file in the root directory of your project

Step 2: Generation the Migrations

```
apple@apples-MacBook-Pro nest-js-deployment-railway % npm run migration:generate -- db/migrations/init

> n-fundamentals-pro@0.0.1 migration:generate
> npm run typeorm -- migration:generate "db/migrations/init"
```

```
production
containers-us-west-89.railway.app
PaX7MXpvWogZoxvcunkL
Migration /Volumes/Macinotsh HD/programming-section/Newline/n-f-code/module-11-deployment-to-railway/nest-js-deployment-railway/d
b/migrations/1686309181524-init.ts has been generated successfully.
apple@apples-MacBook-Pro nest-js-deployment-railway %
```

Now you run the migration it will work like migration will be generated successfully

Step 3: Run the migration command

After the migration file has been generated successfully, execute the run migration command.

```
npm run migration:run
```

You will see this output

```
ListId") REFERENCES "playlists"("id") ON DELETE NO ACTION ON UPDATE NO ACTION
query: ALTER TABLE "artists" ADD CONSTRAINT "FK_f7bd9114dc2849a90d39512911b" FOREIGN KEY ("us
erId") REFERENCES "users"("id") ON DELETE NO ACTION ON UPDATE NO ACTION
query: ALTER TABLE "songs_artists" ADD CONSTRAINT "FK_971d95bf6df45f2b07c317b6b34" FOREIGN KE
Y ("songsId") REFERENCES "songs"("id") ON DELETE CASCADE ON UPDATE CASCADE
query: ALTER TABLE "songs_artists" ADD CONSTRAINT "FK_3f43a7e4032521e4edd2e7ecd29" FOREIGN KE
Y ("artistsId") REFERENCES "artists"("id") ON DELETE NO ACTION ON UPDATE NO ACTION
query: INSERT INTO "migrations"("timestamp", "name") VALUES ($1, $2) -- PARAMETERS: [16863095
49613,"Init1686309549613"]
Migration Init1686309549613 has been  executed successfully.
query: COMMIT
```

Step 4: Push the code to Repo

Source code must be pushed to the GitHub repository to trigger Railway's automatic redeployment of the application. Testing the application can then be conducted by sending a signup request, ensuring that continuous integration and delivery practices are followed for efficient workflow.

It's considered a best practice to include unit tests for the signup feature to validate functionality before pushing code changes. This ensures that the deployment pipeline maintains high-quality code promotion to production environments.

Chapter 12 Testing

Getting started with Jest

What is Jest

Jest, developed by Facebook, stands as a prominent testing framework within the JavaScript ecosystem. It finds extensive use across various JavaScript applications, fostering robust testing practices in environments built upon React, Vue, Angular, and Node.js.

The framework encompasses a myriad of features that aid in crafting thorough tests:

1. Test Runner and Assertion APIs: Jest's integrated test runner efficiently executes tests, while its assertion APIs allow for meticulous verification of expected results, ensuring the reliability of tests.
2. Organized Test Suites and Matchers: Jest enables structuring tests into suites for better organization, coupled with a diverse range of matchers for precise value assertions, contributing to clearer and more maintainable test code.
3. Mock Functions and Spies: Its built-in mocking utilities support the creation of stand-in functions, modules, and dependencies. Spies are also available to monitor and verify the invocation of functions, an essential aspect of testing behavior and interactions.
4. Snapshot Testing: This unique Jest feature facilitates capturing and comparing snapshots of component outputs or data structures, playing a pivotal role in identifying unintended alterations, a practice that greatly enhances change detection workflows.
5. Coverage Analysis: The included code coverage tool quantifies test coverage, highlighting untested parts of a codebase, and prompts a thorough testing discipline that strives for comprehensive coverage.
6. Asynchronous Code Testing: The framework caters to the complexity of asynchronous operations through dedicated utilities, streamlining the testing of promises, callbacks, and other asynchronous patterns.

Jest's straightforward setup, combined with its execution efficiency and expansive support documentation, positions it as a highly favored framework for JavaScript testing. Its seamless

integration with widespread JavaScript tools further solidifies its status as a foundational element of modern JavaScript development workflows.

Step 1: Create Package.json

We are going to play around with Jest to learn the basics of the Jest framework. I want you to open the new directory in your vs code editor and create a `package.json` file

```json
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watchAll"
  },
  "devDependencies": {
    "jest": "^29.5.0",
    "@types/jest": "^29.5.2"
  }
}
```

Install the dependencies by running `npm install`

Step 2: Create a sum.js file

```js
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

Step 3: Write a test case

1. A new file named `sum.test.js` should be created. The Jest framework is designed to recognize files with `.test.js` as testing files automatically.
2. This detection triggers the execution of tests within that file when the test suite is run. It is a best practice to name test files with this convention, as it ensures clarity and a standard across the project's test environment.

```js
const sum = require("./sum");

test("add 1+2 to equal 3", () => {
  expect(sum(1, 2)).toBe(3);
  expect(sum(1, 2)).toBeGreaterThan(2);
});
```

1. Writing the first test in Jest involves crafting test cases that are meant to assert the expected functionality of the application code. It is a good practice to explore various assertions that Jest provides to cover different scenarios and edge cases.

2. Engaging with a variety of Jest assertions can ensure a comprehensive testing suite. Effective testing strategies involve utilizing matchers like `.toBe(), .toEqual()`, and others that can verify the accuracy of different data types and structures within the application.

Auto-Mocking

What is Auto Mocking

Auto Mocking in Jest is the process of automatically creating mock versions of modules or dependencies for unit testing. This feature in Jest, known as "automocking," simplifies test setup by negating the need for manual mocks, which is considered a best practice for maintaining test isolation and efficiency.

With automocking enabled, Jest preemptively generates mock facades for modules imported into the test environment. This substitute of genuine modules with their mock counterparts allows for precise control over their responses and the simulation of various test conditions.

Employing auto-mocking is advantageous for segregating the unit under test from its external dependencies. Mocking these dependent modules ensures that the unit's functionality can be assessed in isolation, a technique recommended for accurate unit testing.

Incorporating Auto Mocking can streamline testing of intricate systems, easing dependency management. It shifts the focus to the unit of code in question, while interactions with its dependencies are managed in a predictable manner, a strategy that aligns with advanced software testing methodologies.

What are Mock Functions

Mock functions, also known as mocks or mock implementations, serve as stand-ins for real functions or dependencies during testing. Within NestJS, these are integral for unit testing to ensure that the code under test is isolated from external interactions.

1. Control Behavior: Mock functions in NestJS are crafted to dictate precise behaviors and outcomes, crucial for simulating various scenarios to evaluate code branches. They are especially important when the real implementations involve unpredictability or non-deterministic behavior.
2. Remove Dependencies: NestJS advocates for mocks to replace actual implementations, thus eliminating the need for external systems or services during testing. This approach is fundamental for creating tests that are both robust and self-contained.
3. Track Function Calls: Mock functions in a NestJS context are designed to record their usage statistics, such as invocation count and argument details. Such information is pivotal for verifying that components interact as expected.

4. Simplify Testing: In NestJS, mock functions streamline the testing process by emulating elaborate or resource-intensive functions. For instance, they can mimic database calls or external API interactions, facilitating more efficient and focused testing environments.

In NestJS, Jest is a favored testing framework where mock functions are either created via the Jest mocking APIs or generated through Jest's auto-mocking feature. Utilizing these mock functions in tandem with assertions is a best practice that confirms the correct behavior of components under test.

Step 1: Basic Mock Function

Now we are going to write examples on Mock functions. I want you to create a test folder and create a new file with `mock-function.spec.js`

```
//mock-function.spec.js
describe("Mock Function Examples", () => {
  it("should create a basic mock function", () => {
    const mockFn = jest.fn(); //1
    // mockFn.mockReturnValue('HELLO WORLD');
    mockFn.mockReturnValue(3); //2
    console.log(mockFn());
    expect(mockFn()).toBe(3);
    expect(mockFn.mock.calls.length).toBe(2); //3
    expect(mockFn).toHaveBeenCalled();
  });
});
```

1. An empty mock function has been created. This serves as a placeholder in testing environments to simulate the behavior of real functions without implementing their logic.
2. The function is configured to return a value of 3. Returning a static value is a common practice for simplifying tests, ensuring that the output is predictable and the function's behavior is consistent.
3. Tracking the calls to mock functions is achievable. This allows for the verification of interactions within the code, ensuring that functions are invoked as expected, which aligns with the best practices of test-driven development.

Analogously, think of a mock function as a stunt double in a film: it performs the actions required for the scene (test case) without being the actual actor (real function), and its performance can be precisely controlled and reviewed.

Step 2: Basic Mock Function with Arguments

A basic mock function with arguments is designed to simulate the behavior of a real function, allowing for controlled responses and interactions. It takes predefined arguments and, when invoked within tests, returns a controlled output without executing any actual logic, akin to a rehearsal before a live performance.

Best practice dictates that such mock functions should be as close to the real implementation as possible to ensure reliable tests. Additionally, specifying clear return values or behaviors based on the arguments received allows for a comprehensive testing strategy that can catch a wide array of potential issues before deployment.

```
it("should create a mock function with argument", () => {
  const createSongMock = jest.fn((createSongDTO) => ({
    id: 1,
    title: createSongDTO.title,
  }));

  console.log(createSongMock({ title: "Lover" }));
  expect(createSongMock).toHaveBeenCalled();
  expect(createSongMock({ title: "Lover" })).toEqual({ id: 1, title: "Lover" });
});
```

You can also pass an argument to the mock function just like we did `createSongDTO`

Step 3: Create a Mock Function

```
it("create mock function using mockImplementation", () => {
  const mockFn = jest.fn();
  mockFn.mockImplementation(() => {
    //1
    console.log("Mock function called");
  });

  mockFn(); // Output: 'Mock function called'
  expect(mockFn).toHaveBeenCalled();
  expect(mockFn).toHaveBeenCalledTimes(1);
  expect(mockFn).toHaveBeenCalledWith();
});
```

1. You can also create a mock function with `mockImplementation` method.

Step 4: Create Mock Function with a Promise

Creating a mock function with a Promise in NestJS facilitates the simulation of database calls or external service responses during testing. By returning a resolved or rejected promise, the mock replicates the asynchronous behavior of real-world scenarios, ensuring comprehensive test coverage.

It is considered a best practice to isolate and mock external dependencies in tests to verify that the system's reaction to various potential responses is handled correctly. Analogous to a flight simulator for pilots, this technique allows developers to safely test the behavior of the application in a controlled environment before deployment.

```
it("it should create a mock function with promise", () => {
  const mockFetchSongs = jest.fn();
  mockFetchSongs.mockResolvedValue({ id: 1, title: "Dancing Feat" }); //1

  mockFetchSongs().then((result) => {
    console.log(result);
  });

  expect(mockFetchSongs).toHaveBeenCalled();
  expect(mockFetchSongs()).resolves.toEqual({ id: 1, title: "Dancing Feat" });
//2
});
```

1. The operation will resolve to a promise that yields the object `{ id: 1, title: 'Dancing Feat' }`. This approach is aligned with best practices, encapsulating the asynchronous nature of operations which may involve I/O processes, such as database interactions.
2. To verify the resolution of a promise, one can utilize the `resolves` matcher in testing frameworks. This facilitates the writing of clean, robust tests that ensure promises are fulfilled as expected, which is an essential aspect of reliable software delivery.

SpyOn Functionality

Spies play a crucial role in testing scenarios where there is a need to monitor and manipulate the behavior of function calls. They enable the validation of how functions are used without altering their actual implementation, aligning with best practices for maintaining clean and reliable test suites.

The `jest.spyOn()` function in Jest is employed to construct a spy for any method of an existing object, allowing for a granular control and assertion of that method's interactions. This utility provides the advantage of asserting calls and responses, facilitating a mock implementation when necessary, which is particularly useful when needing to simulate real-world scenarios in a controlled test environment.

Step 1: SpyOn existing Object

Creating a new file named `spyon-demo.spec.js` in the test directory initiates the process for implementing test spies. In NestJS, utilizing `jest.spyOn()` allows monitoring and asserting the behavior of functions within existing objects, ensuring that best practices for test coverage and quality assurance are adhered to.

Incorporating spies into unit tests by using `jest.spyOn()` is analogous to deploying surveillance equipment; it discreetly observes the interactions without obstructing the

```
const songRepository = {
```

```
  create: (createSongDTO) => {
    // Original method implementation
  },
};

describe("spyOn Demo", () => {
  it("should spyon the existing object", () => {
    const spy = jest.spyOn(songRepository, "create"); //1

    // Call the method
    songRepository.create({ id: 1, title: "Lover" }); //2

    // Assertions
    expect(spy).toHaveBeenCalled(); //3
    expect(spy).toHaveBeenCalledWith({ id: 1, title: "Lover" });
    expect(spy).toHaveBeenCalled();
    expect(spy).toHaveBeenCalledTimes(1);

    // Restore the original method
    spy.mockRestore();
  });
});
```

1. A mock function has been implemented for the `create` property in the `SongRepository` to simulate functionality without invoking the actual implementation. Utilizing `spyOn` provides the additional benefit of creating a mock function while also monitoring the call activities, which aligns with best practices for testing where monitoring interactions with certain components is essential.
2. Upon invoking `songRepository.create`, the underlying mechanism defers to the `spy` function equipped with the bespoke fake implementation. This approach ensures that tests can verify that the repository's `create` function is being called as expected, a technique considered a best practice to validate the integration point without relying on the actual database operation, enhancing test reliability and execution speed.

By employing these techniques, one mimics the delicate workings of a timepiece, ensuring each gear (or function) interacts correctly with the others, while still allowing for inspection of each individual movement (or function call).

Step 2: Spy on the class Method

Simulating the behavior of a class method requires employing a spy. In NestJS, spies are commonly utilized in testing scenarios where the actual execution of a method should be observed without triggering its real effects, akin to a surveillance camera monitoring activity discreetly.

A best practice is to use spies from robust testing libraries like Jest, which can keep track of calls to the method, arguments passed, and even allow specifying return values. This strategy ensures that

unit tests remain isolated and are not affected by external factors, much like using a test dummy in car crash simulations to understand impacts without risking actual harm.

An additional tip is to always restore or clean up spies after each test to prevent unexpected behavior in subsequent tests. This is similar to resetting a chessboard after a game; it ensures that the starting conditions are consistent every time.

```
class ArtistRepository {
  save(createArtistDTO) {
    // Original method implementation
  }
}
it("should spy on the class method", () => {
  const artist = new ArtistRepository();
  const spy = jest
    .spyOn(artist, "save")
    .mockImplementation((createArtistDTO) => createArtistDTO);

  // Call the method
  artist.save({ name: "Martin Garrix" }); //1

  console.log(spy({ name: "Martin Garrix" }));
  // Assertions
  expect(spy).toHaveBeenCalled();
  expect(spy).toHaveBeenCalledWith({ name: "Martin Garrix" });

  // Restore the original method
  spy.mockRestore();
});
```

1. Invoking `artist.save` engages the implementation of the spy function, allowing for monitoring of function calls. The spy function accepts `createArtistDTO` as an argument and returns the identical `createArtistDTO`, enabling the confirmation of data integrity and flow through the function.

Best practice dictates the use of such spy functions in test cases to ensure that methods are called with the correct arguments, effectively simulating the behavior of complex dependencies. Analogous to a checkpoint in a relay race, it verifies the baton (in this case, `createArtistDTO`) is passed correctly, ensuring the next runner (function or method in the application) can proceed without issues.

Step 3: restore All mocks usuing beforeEach hook

```
describe('spyOn Demo', () => {
  afterEach(() => jest.resetAllMocks());
```

```
}
```

Removing the manual `mockRestore` calls from each test case and utilizing `jest.resetAllMocks` within the `afterEach` hook streamlines the testing process. This approach ensures that all mocks are reset automatically after each test, promoting cleaner test architecture and reducing the chance of state leakage between tests, much like a blackboard is wiped clean after each lesson to ensure a fresh start for new information.

Incorporating this method is considered a best practice, as it adheres to the principle of test isolation, akin to how surgeons sterilize their tools after each procedure to prevent cross-contamination. Additionally, this tactic enhances test reliability and maintainability, simplifying future updates to the test suite.

Unit Test Controller

Unit Testing

Unit testing represents a pivotal software testing methodology where the smallest testable parts of an application are evaluated independently. It is designed to affirm that each code segment, be it functions, methods, or classes, operates as intended, delivering the correct outcomes for distinct inputs.

Insights into the practice of unit testing include:

1. Unit tests facilitate the early discovery of issues, typically during the development phase. This preemptive measure allows for the rectification of problems prior to their expansion throughout the software, thereby streamlining the development process.
2. The application of unit tests enhances code integrity and fosters the development of code that is both modular and maintainable. Such tests instill confidence in the functionality of code segments and serve as a bulwark against future regressions when the codebase evolves.
3. When a unit test signals a failure, it precisely pinpoints the troubled segment, significantly expediting the debugging process. This targeted approach to problem-solving is integral to efficient software maintenance.
4. As code undergoes refactoring, unit tests provide a framework that assures the consistency of unit functionality. They are the custodians of code behavior, ensuring that modifications do not inadvertently affect existing features.

Step 1: Run the Test

Initiating the test process requires executing `npm run test:watch`, which activates the test watcher. Upon initiation, pressing `p` allows for pattern matching where entering `song.controller.spec.ts` focuses the test watcher on this specific file.

As a best practice, targeting specific test files by pattern can significantly streamline the debugging process, akin to using a surgical strike to identify and resolve issues. This focused approach is more efficient than a broader testing strategy, which could be compared to casting a wide net and sifting through irrelevant information.

In addition, it's advisable to keep the test files named consistently with their corresponding modules. This convention simplifies the identification and execution of related tests, much like arranging books by genre simplifies the search in a library.

```typescript
import { Test, TestingModule } from "@nestjs/testing";
import { SongController } from "./song.controller";

describe("SongController", () => {
  let controller: SongController;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      controllers: [SongController],
    }).compile(); //1

    controller = module.get<SongController>(SongController); //2
  });

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

1. Nest.js automatically generates a fake testing module. Upon execution of the compile function, it instantiates all dependencies required for the testing module, ensuring a controlled environment akin to a laboratory where experiments can be conducted in isolation.
2. Obtaining an instance of the controller dependency is a straightforward process. It involves using the testing module's get method, which serves as a direct line to the desired dependencies, much like retrieving a specific book from a well-organized shelf using a catalog.

When the test is executed, the error message is revealed. This error message acts as an early indicator, a lighthouse warning ships of potential danger, enabling developers to pinpoint the fault and implement corrective measures. As a best practice, thorough inspection and understanding of the error message are crucial; it should be seen as a map leading to the bug that needs to be fixed. It's advisable to treat tests as first-class citizens, crafting them with the same care as production code, and reviewing error messages from tests with a keen, diagnostic eye to maintain robustness in the application.

```
SongController

X should be defined (41 ms)
```

- SongController › should be defined

Nest can't resolve dependencies of the SongController (?). Please make sure that the argument SongService at index [0] is available in the RootTestModule context.

Potential solutions:
- Is RootTestModule a valid NestJS module?
- If SongService is a provider, is it part of the current RootTestModule?
- If SongService is exported from a separate @Module, is that module imported within RootTestModule?
@Module({
imports: [/* the Module containing SongService */]
})

When you check the `SongController` it is dependent on the `SongService`. We have to unit test the controller functions individually. That's why we need to mock the `SongService` to individually test the controller functions

Step 2: Creating a Mock SongService

```
let service: SongService;

const module: TestingModule = await Test.createTestingModule({
  controllers: [SongController],
  providers: [
    SongService,
    {
      provide: SongService,
      useValue: {
        getSongs: jest
          .fn()
          .mockResolvedValue([{ id: "123131", title: "Dancing" }]),
        getSong: jest.fn().mockImplementation((id: string) => {
          return Promise.resolve({ id: id, title: "Dancing" });
        }),
        createSong: jest
          .fn()
          .mockImplementation((createSongDTO: CreateSongDTO) => {
            return Promise.resolve({ id: "a uuid", ...createSongDTO });
          }),
        updateSong: jest
          .fn()
          .mockImplementation((updateSongDTO: UpdateSongDTO) => {
```

```
        return Promise.resolve({ affected: 1 });
      }),

    deleteSong: jest.fn().mockImplementation((id: string) => {
      return Promise.resolve({ affected: 1 });
    }),
    },
  },
  ],
}).compile();

controller = module.get<SongController>(SongController);
service = module.get<SongService>(SongService);
```

The `useValue` syntax in NestJS's dependency injection system allows for substituting real implementations with fake ones, useful for testing purposes. For instance, when configuring a provider in a module, setting `useValue` to an object with methods mirroring those of `SongService` can simulate database interactions, aligning with the best practice of using fakes or mocks during unit testing to ensure tests run quickly and reliably without real database dependencies.

As an analogy, using `useValue` is like using a stunt double in a movie; it stands in for the real actor (the `SongService`), performing necessary actions during testing scenes, which allows the real service to remain unaffected and free from the potential side effects of rigorous testing procedures. This approach is an elegant solution to maintaining a controlled test environment.

Step 3: Test Controller functions

```
describe("getSongs", () => {
  it("should give me the array of songs", async () => {
    const songs = await controller.getSongs();
    expect(songs).toEqual([{ id: "123131", title: "Dancing" }]);
  });
});

describe("getSong by id", () => {
  it("should give me the song by id", async () => {
    const song = await controller.getSong("123131");
    expect(song.id).toBe("123131");
  });
});

describe("createSong", () => {
  it("should create a new song", async () => {
    const newSongDTO: CreateSongDTO = {
      title: "Runaway",
    };
```

```
    const song = await controller.createSong(newSongDTO);
    expect(song.title).toBe("Runaway");
  });
});

describe("updateSong", () => {
  it("should update the song DTO", async () => {
    const updatesongDTO: UpdateSongDTO = {
      title: "Animals",
    };
    const updateResults = await controller.updateSong("a uuid", updatesongDTO);
    expect(updateResults).toBeDefined();
    expect(updateResults.affected).toBe(1);
  });
});

describe("deleteSong", () => {
  it("should delete the song", async () => {
    const deleteResult = await controller.deleteSong("a uuid");
    expect(deleteResult.affected).toBe(1);
  });
});
```

Unit Test Service

Step 1: Run the test for SongService

Initially run the test for Song Service.

```
import { Test, TestingModule } from "@nestjs/testing";
import { SongService } from "./song.service";

describe("SongService", () => {
  let service: SongService;

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

    service = module.get<SongService>(SongService);
```

```
  });

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

npm run test:watch

Now there's a pattern that's established for testing. `SongService` needs similar testing help as the controller.

```
FAIL  src/song/song.service.spec.ts (34.922 s)
  SongService
    ✕ should be defined (42 ms)

  ● SongService › should be defined

    Nest can't resolve dependencies of the SongService (?). Please make sure that
the argument SongRepository at index [0] is available in the RootTestModule
context.

    Potential solutions:
    - Is RootTestModule a valid NestJS module?
    - If SongRepository is a provider, is it part of the current RootTestModule?
    - If SongRepository is exported from a separate @Module, is that module
imported within RootTestModule?
      @Module({
        imports: [ /* the Module containing SongRepository */ ]
      })

      6 |
      7 |    beforeEach(async () => {
  >   8 |      const module: TestingModule = await Test.createTestingModule({
        |                                                  ^
      9 |        providers: [SongService],
     10 |      }).compile();
     11 |
```

Now there's the same problem as the conroller. To fix this, create the mock `SongRepository` to run the test for SongService.

Step 2: Create the Mock Repository

```
let service: SongService;
```

```typescript
let repo: Repository<Song>;

const oneSong = { id: "a uuid", title: "Lover" };
const songArray = [{ id: "a uuid", title: "Lover" }];

const module: TestingModule = await Test.createTestingModule({
  providers: [
    SongService,
    {
      provide: getRepositoryToken(Song),
      useValue: {
        find: jest
          .fn()
          .mockImplementation(() => Promise.resolve(songArray)),
        findOneOrFail: jest
          .fn()
          .mockImplementation((options: FindOneOptions<Song>) => {
            return Promise.resolve(oneSong);
          }),
        create: jest
          .fn()
          .mockImplementation((createSongDTO: CreateSongDTO) => {
            return Promise.resolve(oneSong);
          }),
        save: jest.fn(),
        update: jest
          .fn()
          .mockImplementation(
            (id: string, updateSongDTO: UpdateSongDTO) => {
              return Promise.resolve(oneSong);
            },
          ),
        delete: jest
          .fn()
          .mockImplementation((id: string) =>
            Promise.resolve({ affected: 1 }),
          ),
      },
    },
  ],

service = module.get<SongService>(SongService);
repo = module.get<Repository<Song>>(getRepositoryToken(Song));
```

Step 3: Test SongService

```
it("should give me the song by id", async () => {
  const song = await service.getSong("a uudi");
  const repoSpy = jest.spyOn(repo, "findOneOrFail");
  expect(song).toEqual(oneSong);
  expect(repoSpy).toBeCalledWith({ where: { id: "a uudi" } });
});

it("should create the song", async () => {
  const song = await service.createSong({ title: "Lover" });
  expect(song).toEqual(oneSong);
  expect(repo.create).toBeCalledTimes(1);
  expect(repo.create).toBeCalledWith({ title: "Lover" });
});

it("should update the song", async () => {
  const song = await service.updateSong("a uuid", { title: "Lover" });
  expect(repo.update).toBeCalledTimes(1);
  expect(song).toEqual(oneSong);
});

it("should delete the song", async () => {
  const song = await service.deleteSong("a uuid");
  const repoSpyOn = jest.spyOn(repo, "delete");
  expect(repo.delete).toBeCalledTimes(1);
  expect(song.affected).toBe(1);
  expect(repoSpyOn).toBeCalledWith("a uuid");
});
```

End To End Testing

What is E2E Testing?

End-to-end testing (E2E testing) encompasses a comprehensive software testing approach, simulating real user scenarios to ensure the application operates effectively from start to finish. It scrutinizes the system's behavior and functionality through the lens of user interactions, extending across various components and systems.

In-depth analysis of end-to-end testing reveals:

1. Integration issues become apparent through E2E testing, as it uncovers problems that surface during the interaction between disparate application segments. Adopting this testing

strategy is a best practice for identifying data inconsistencies, communication breakdowns, and API compatibility issues, which are often obscured within the confines of unit and integration testing stages.

2. E2E testing is instrumental in validating user journeys, ensuring that the application reliably navigates the critical pathways a user might traverse. It acts as a safeguard, confirming that the software responds with the correct outputs across a multitude of user interactions, which is a testament to meticulous software craftsmanship.

3. The user experience is enhanced when E2E testing is employed, as it encompasses a holistic view of the application's operation. This approach is akin to a conductor overseeing an orchestra, ensuring every section performs harmoniously, resulting in a seamless performance—mirrored in the software by uniform navigation, fluid data transactions, accurate UI representations, and responsive user engagements.

4. The deployment process is fortified with E2E testing, functioning as the final rehearsal before the live performance of the application in the production environment. It instills a level of assurance that all systems operate as expected, analogous to a dress rehearsal in theater, ensuring that every act and scene transitions flawlessly before the curtain rises.

Step 1: Add E2E script

```
"test:e2e:watch": "jest --watch --detectOpenHandles --config ./test/jest-e2e.json"
```

In Nest.js, a watch script for end-to-end (E2E) testing is not part of the default setup. For continuous feedback during development, it is recommended to implement a watch mechanism, such as using Jest's `--watch` flag, to automatically re-run E2E tests when changes are detected.

Step 2: Add TypeORM Module

The TypeORM module must be registered while creating the TestingModule.

```
imports: [
     TypeOrmModule.forRoot({
       type: 'postgres',
       url: 'postgres://postgres:root@localhost:5432/test-dev',
       synchronize: true,
       entities: [Song],
       dropSchema: true,
     }),
     SongModule,
   ],
 }).compile();
```

Ensuring the creation of a separate database for application testing is essential. Setting the `dropSchema` option to `true` within the TypeORM configuration ensures the schema is automatically dropped post-testing, maintaining a clean state.

Step 3: Clear SongRepository

```
afterEach(async () => {
  // Fetch all the entities
  const songRepository = app.get("SongRepository");
  await songRepository.clear();
});
```

The `SongRepository` instance can be obtained by invoking the `app.get` method.

Step 4: Test Get Songs endpoints

Registering the TypeORM module is a necessary step in the configuration of the TestingModule in NestJS, ensuring that the data layer is appropriately integrated for testing environments. This process aligns with established best practices, enabling the simulation of database interactions and the assessment of data persistence within the service under test.

When testing Get Songs endpoints, the approach includes simulating client requests and asserting the expected responses, which should reflect the retrieval of song data. It's a common practice to mock the service layer to return predefined data, thereby isolating the controller's response logic for accurate and efficient validation.

```
import * as request from "supertest";

const createSong = (createSongDTO: CreateSongDTO): Promise<Song> => {
  const song = new Song();
  song.title = createSongDTO.title;
  const songRepo = app.get("SongRepository");
  return songRepo.save(song);
};

it(`/GET songs`, async () => {
  const newSong = await createSong({ title: "Animals" });
  const results = await request(app.getHttpServer()).get("/songs");
  expect(results.statusCode).toBe(200);
  expect(results.body).toHaveLength(1);
  expect(results.body).toEqual([newSong]);
});
```

The `supertest` package, pre-installed in NestJS, facilitates the testing of HTTP APIs by simulating server behavior and handling HTTP requests and responses. This allows for the creation of robust test suites that ensure APIs behave as expected under various conditions, adhering to best practices for maintaining high-quality software standards.

Step 5: Test GET Song Endpoint

When conducting tests for the GET Song endpoint, it is essential to ensure that the request retrieves the correct song data and handles potential errors gracefully. The test should mimic a client's request for a song, verifying that the endpoint returns the expected song data with the correct HTTP status code.

```
it("/GET songs/:id", async () => {
  const newSong = await createSong({ title: "Animals" });
  const results = await request(app.getHttpServer()).get(
    `/songs/${newSong.id}`
  );
  expect(results.statusCode).toBe(200);
  expect(results.body).toEqual(newSong);
});
```

Step 6: Test PUT Song Endpoint

In testing the PUT Song endpoint, the focus is on the endpoint's ability to update an existing song's data accurately and to validate any changes made. The test should simulate a client updating song details, confirming that the endpoint processes the update correctly and returns a success response or appropriate error message.

```
it("/PUT songs/:id", async () => {
  const newSong = await createSong({ title: "Animals" });
  const updateSongDTO: UpdateSongDTO = { title: "Wonderful" };
  const results = await request(app.getHttpServer())
    .put(`/songs/${newSong.id}`)
    .send(updateSongDTO as UpdateSongDTO);
  expect(results.statusCode).toBe(200);
  expect(results.body.affected).toEqual(1);
});
```

Step 7: Test Create Song Endpoint

To effectively test the 'Create Song' endpoint, the TestingModule should instantiate with the TypeORM module integrated, ensuring database interactions are part of the test environment. This integration is crucial for replicating the application's behavior in a controlled testing scenario, allowing for accurate verification of the endpoint's functionality.

A best practice in this context is to utilize mock repositories or in-memory databases, which provide isolation for tests, thus preventing side effects on the actual database and ensuring that each test case runs in a consistent state. This technique not only accelerates testing procedures but also maintains the integrity of test scenarios.

```
it("/POST songs", async () => {
  const createSongDTO = { title: "Animals" };
  const results = await request(app.getHttpServer())
    .post(`/songs`)
    .send(createSongDTO as CreateSongDTO);
  expect(results.status).toBe(201);
  expect(results.body.title).toBe("Animals");
});
```

Step 8: Test Delete Song Endpoint

For testing the Delete Song endpoint within a NestJS application, the TestingModule should be configured to replicate the application's behavior in a controlled environment. The endpoint's robustness is ensured by creating test cases that cover all possible scenarios, including valid deletions, attempts to delete non-existent songs, and handling of unauthorized requests.

Best practices dictate the utilization of TypeORM's transactional operations during testing to maintain database integrity, which allows for each test to run in isolation without impacting the database state. Additionally, incorporating service mocks within the test suite is recommended to simulate interactions with the database or external services, ensuring tests are fast and reliable.

```
it("/DELETE songs", async () => {
  const createSongDTO: CreateSongDTO = { title: "Animals" };
  const newSong = await createSong(createSongDTO);
  const results = await request(app.getHttpServer()).delete(
    `/songs/${newSong.id}`
  );
  expect(results.statusCode).toBe(200);
  expect(results.body.affected).toBe(1);
});
```

Chapter 13 WebSocket and SocketIo Integration

Using Speedy Web Compiler

Nest.js version 10 introduces enhancements to the compiler speed, resulting in accelerated project execution times. This advancement leads to a more efficient development experience when building Nest.js applications, as faster compilation directly improves the feedback loop for developers.

Best practices suggest leveraging such improvements to compiler speed by regularly updating to the latest version, ensuring that the development environment is as performant as possible. Additionally, it's advisable to monitor the project's build times after upgrading, to quantify the improvement and identify any areas where the newer version can be further optimized for speed.

Step 1: Uninstall the @nestjs/cli

```
npm uninstall -g @nestjs/cli
```

Step 2: Install @nestjs/cli

```
npm install -g @nestjs/cli
```

Step 3: Create a new Project

```
nest new <project-name>
```

Step 4: Make speedy web compiler

You have to add these two properties in `nest-cli.json`

```
"compilerOptions": {
    "builder": "swc",
    "typeCheck": true
  }
```

Step 5: Install the SWC packages

```
npm i -D @swr/cli @swr/core
```

Step 6: Run the Application

```
npm run start:dev
```

Create Web Socket Server

What are WebSockets?

WebSockets are a communication protocol that enables real-time, bidirectional communication between a client (such as a web browser) and a server. Unlike traditional HTTP requests, where the client sends a request and the server responds, WebSockets establish a persistent connection between the client and the server, allowing for ongoing communication in both directions.

Here are some key characteristics of WebSockets:

1. Full-duplex communication: WebSockets enable simultaneous two-way communication between the client and the server. Both parties can send messages to each other independently and in real-time.
2. Persistent connection: Unlike traditional HTTP connections that are stateless and short-lived, WebSockets maintain a persistent connection between the client and the server. This eliminates the need for repeated connection setup and teardown with each request.
3. Real-time updates: With WebSockets, data can be pushed from the server to the client or vice versa as soon as it becomes available. This allows for real-time updates and notifications without the need for the client to continuously poll the server for updates.
4. Low overhead: WebSockets have a minimal overhead compared to other communication protocols like HTTP. Once the initial connection is established, the overhead of subsequent messages is significantly reduced.
5. WebSockets are commonly used in various applications and use cases that require real-time communication, such as chat applications, collaborative editing tools, real-time analytics, online gaming, and live streaming. They provide a more efficient and responsive alternative to techniques like long-polling or frequent AJAX requests for real-time updates.

Step 1: Install Dependencies

```
"@nestjs/platform-socket.io": "^10.0.3",
"@nestjs/websockets": "^10.0.3",
```

First of all, install these two dependencies to work with the websocket.

Step 2: Create Events Module and Gateway

```
nest g module events
nest g gateway events
```

In the `events.module.ts` file, `EventsGateway` is included as a provider to facilitate real-time bi-directional event-based communication. Best practice dictates isolating such gateways within their dedicated module to streamline scalability and maintain clear boundaries of responsibility within the application.

```
// Events.module.ts
import { Module } from "@nestjs/common";
import { EventsGateway } from "./events.gateway";
```

```ts
@Module({
  providers: [EventsGateway],
})
export class EventsModule {}
```

Step 3: Create a new Event

```ts
//events.gateway.ts
import { OnModuleInit } from "@nestjs/common";
import {
  MessageBody,
  SubscribeMessage,
  WebSocketGateway,
  WebSocketServer,
  WsResponse,
} from "@nestjs/websockets";
import { Observable, of } from "rxjs";
import { Server } from "socket.io";

@WebSocketGateway({
  cors: {
    origin: "*",
  },
})
export class EventsGateway implements OnModuleInit {
  @WebSocketServer()
  server: Server;

  onModuleInit() {
    //3
    this.server.on("connection", (socket) => {
      console.log(socket.id);
      console.log(socket.connected);
    });
  }
  // You can listen to this event
  // Client can send message to me by using the message key/event name
  @SubscribeMessage("message")
  newMessage(
    @MessageBody()
    data: any
  ) {
    console.log("Message is received from the client");
    console.log(data); //Here we did not send a message back to the server
```

```
    // Save the message to the database and return the reply in the response
    // Call the service method to save the record in the DB
  }
}
```

Gateways operate on an event-driven architecture, where events are emitted and consumed by clients and the server. Clients can listen for specific events and the server can emit events to trigger actions on the client side.

1. Gateways in Nest.js are implemented using decorators and a class-based approach. A gateway class is decorated with the @WebSocketGateway() decorator, which marks it as a WebSocket gateway. Within the gateway class, you can define event handlers to handle WebSocket events like connection, disconnection, and message reception.

Gateways can be treated as providers; this means they can inject dependencies through the class constructor. Also, gateways can be injected by other classes (providers and controllers) as well. 2. You can access to webSocket server by using this decorator. You can find out who has connected to your Gateway and their socket ID 3. I choose `onModuleInit` to check the WebSocket connection with the client 4. The `@SubscribeMessage('message')` decorator is applied to the newMessage method. This decorator specifies that this method should be executed when a WebSocket message with the event name 'message' is received from the client.

The `@MessageBody()` decorator is applied to the data parameter. This decorator instructs Nest.js to extract the message payload from the incoming WebSocket message and bind it to the data parameter.

In summary, it demonstrates a method in a WebSocket gateway that is triggered when a WebSocket message with the event name 'message' is received from the client. It logs the received message to the console and potentially performs further actions such as saving the message to a database using a service method.

Send Message from Frontend Client

Step 1: Consuming Event from the Frontend

When consuming the WebSocket 'message' event, plain JavaScript can be utilized effectively. This approach ensures that the event handling is kept lightweight and free from additional dependencies.

A best practice in this scenario is to ensure robust error handling and validation of incoming messages to prevent potential issues stemming from malformed data. It's also advisable to structure the code for easy maintenance and scalability, possibly by abstracting the event handling into separate, testable functions or classes.

```
<html>
```

```
  <head>
    <body>
      <p>Please check the console for the message reply</p>
    </body>
    <script
      src="https://cdn.socket.io/4.3.2/socket.io.min.js"

integrity="sha384-KAZ4DtjNhLChOB/hxXuKqhMLYvx3b5MlT55xPEiNmREKRzeEm+RVPlTnAn0ajQN
s"
      crossorigin="anonymous"
    ></script>
    <script>
      const socket = io('http://localhost:3000');
      socket.on('connect', function () {
        console.log('Connected');
        // Send Message to Websocket Server
        socket.emit('message', { msg: 'THIS IS THE MESSAGE FROM THE CLIENT' });
      });

      socket.on('message', function (data) {
        console.log('event ', data);
      });
      socket.on('exception', function (data) {
        console.log('event', data);
      });
      socket.on('disconnect', function () {
        console.log('Disconnected');
      });
    </script>
  </head>

  <body></body>
</html>
```

Step 2: Return data from the message event using Observable

Returning data from the message event utilizing Observables in Nest.js aligns with reactive
programming principles, facilitating a stream-based approach to handle data sequences over time.
This method is crucial for scenarios requiring real-time updates or complex asynchronous
operations, where Observables can efficiently manage and deliver data payloads as events occur.

As a best practice, it's advisable to leverage the RxJS library, which Nest.js integrates seamlessly
with, to create Observables. This enables fine-grained control over event handling, allowing for the

composition of sophisticated data handling and transformation strategies, essential for robust and reactive application architectures.

```
 @SubscribeMessage('message')
  newMessage(
    @MessageBody()
    data: any,
  ): Observable<WsResponse<any>> { //1
    console.log('Message is receieved from the client');
    console.log(data);
    return of({ event: 'message', data: 'Learn Node' });
  }
```

Summary

In the context of Nest.js, the integration of Socket.IO for WebSocket communication is vital for real-time, bi-directional communication between web clients and servers. The setup involves establishing a persistent connection that allows clients to send messages to the server and listen for responses. Best practices include handling exceptions gracefully and ensuring that disconnection events are managed to maintain robust communication. Additionally, logging received data to the console is a useful technique for monitoring and debugging the flow of messages. When implementing this in Nest.js, one would typically encapsulate WebSocket management within a dedicated module or service, leveraging Nest.js's dependency injection to manage lifecycle events and provide a clean separation of concerns.

Chapter 14 Build GraphQL APIs

GraphQL Server Setup

Step 1: Install Dependencies

npm install @apollo/server @nestjs/apollo @nestjs/graphql graphql ts-morph

@apollo/server: This package is part of the Apollo GraphQL ecosystem and provides a GraphQL server implementation. It allows you to create a GraphQL server using Node.js, enabling you to define your GraphQL schema, and resolvers, and handle incoming GraphQL queries and mutations.

@nestjs/apollo: This package is an integration package provided by the NestJS framework. It allows you to use Apollo Server seamlessly within a NestJS application. NestJS is a powerful

Node.js framework that provides a modular and structured approach to building scalable and maintainable server-side applications.

`@nestjs/graphql`: This package is another part of the NestJS ecosystem and provides GraphQL support for NestJS applications. It allows you to define GraphQL schemas, and resolvers, and use decorators to annotate your classes and methods to create GraphQL endpoints. It works in conjunction with @apollo/server or other GraphQL server implementations.

`graphql`: This package is the core GraphQL library that provides the fundamental building blocks for working with GraphQL. It includes utilities for parsing, validating, and executing GraphQL queries and mutations. It is a widely used package in the GraphQL community and is required by most GraphQL server implementations.

`ts-morph`: This package is a TypeScript AST (Abstract Syntax Tree) manipulation library. It allows you to programmatically analyze, modify, and generate TypeScript code. It can be particularly useful when working with code generation or refactoring tools, such as generating TypeScript types from GraphQL schemas or modifying existing TypeScript files programmatically.

In summary, the packages we mentioned are commonly used in the context of building GraphQL servers with NestJS and Apollo. They provide the necessary tools and integrations to define GraphQL schemas, handle incoming queries and mutations, and manipulate TypeScript code when needed.

Step 2: Import GraphQL Module

```
GraphQLModule.forRoot<ApolloDriverConfig>({
    driver: ApolloDriver,
    typePaths: ['./**/*.graphql'],
    definitions: {
      path: join(process.cwd(), 'src/graphql.ts'),
      outputAs: 'class',
    },
  }),
```

Step 3: Create a Schema File

A schema file named `song.graphql` must be created within the `songs` directory. This file defines the GraphQL schema, outlining the types, queries, mutations, and subscriptions for song-related operations, serving as a blueprint for the GraphQL API structure. It is considered good practice to maintain schema files within their respective domain folders to encapsulate and modularize the API definitions, facilitating better maintainability and scalability of the application's GraphQL architecture.

```
type Song {
  id: ID!
  title: String!
}
```

```
type Query {
  songs: [Song!]!
}
```

Utilizing this schema, one can execute GraphQL queries to retrieve data regarding songs. The sole query operation provided is `songs`, yielding an array of song objects, with each object including properties such as `ID` and `title`.

Best practices recommend ensuring that each query is optimized for performance, possibly by leveraging indexes on the `ID` field to expedite lookup times. Additionally, implementing proper error handling and security measures, such as validation and authentication on queries, can protect against inefficient data retrieval and unauthorized access.

For example, to retrieve all the songs, you can execute the following GraphQL query:

```
query {
  songs {
    id
    title
  }
}
```

The query retrieves all songs from the GraphQL server, delivering both IDs and titles to the client.

For the query to function in a production environment, resolvers must be developed to establish the methods of data retrieval upon execution of the songs query, which is not depicted in this schema outline.

Best practices suggest the implementation of resolvers should be designed to be efficient and scalable, often including pagination and filtering capabilities to handle large datasets. Additionally, implementing caching mechanisms can significantly enhance performance by reducing redundant database calls for frequently requested data.

Step 4: Generate Typings

When setting up a NestJS project, a `generate-typings.ts` file must be created in the root directory. This file is instrumental in generating TypeScript typings that align with the data models and provides strong typing benefits throughout the application.

A best practice in this process involves ensuring that the script within `generate-typings.ts` is both maintained with the current project's structure and executed as part of the build process. This maintains type accuracy, aiding in early detection of potential type mismatches or errors during development.

```
import { GraphQLDefinitionsFactory } from "@nestjs/graphql";
import { join } from "path";
```

```
const definitionsFactory = new GraphQLDefinitionsFactory();
definitionsFactory.generate({
  typePaths: ["./src/**/*.graphql"],
  path: join(process.cwd(), "src/graphql.ts"),
  outputAs: "class",
});


export class Song {
  id: string;
  title: string;
}

export abstract class IQuery {
  abstract songs():
    | Nullable<Nullable<Song>[]>
    | Promise<Nullable<Nullable<Song>[]>>;
}

type Nullable<T> = T | null;
```

In the context of Nest.js, when defining a resolver for GraphQL, specific types must be declared to ensure the proper resolution of queries and mutations. It is essential to accurately define the types to map the application's data structures to the GraphQL schema, facilitating clear and type-safe data exchange.

A best practice includes leveraging TypeScript's strong typing system within Nest.js to reinforce the integrity of GraphQL resolvers. By doing so, one ensures that the code aligns with the defined schema, reducing the potential for runtime errors and improving maintainability.

Step 4: Create Script for the Typings

```
"generate:typings": "ts-node generate-typings.ts",
```

You have to add a script in the `package.json` file. You can run this command by executing this script

```
npm run generate:typings
```

Define Queries and Mutations

Step 1: Define a query for a single song

In the given GraphQL schema definition, a `Query` type is defined to facilitate fetching of specific `Song` data by its unique identifier, ensuring a structured and type-safe way of querying the data. This approach optimizes the data retrieval process by allowing precise and efficient querying for only the required song information.

```
type Query {
    song(id: ID!): Song!
}
```

In the context of a NestJS application, extending the GraphQL schema to include queries for specific items, such as songs, enhances the API's capabilities. Such an extension allows clients to retrieve a single song using its unique identifier, which optimizes data retrieval by fetching only the required item rather than the entire collection.

It is a widely accepted best practice to provide such granular query capabilities, as it can significantly improve the performance of the application by reducing the amount of data transferred over the network and the load on the server. Additionally, crafting precise and efficient queries is a hallmark of well-designed GraphQL services, allowing for flexible and optimized client-server interactions.

```
query {
    song(id: "123") {
        id
        title
    }
}
```

Step 2: Define Mutations

The provided GraphQL schema defines mutations, which are operations that modify data on the server. Specifically, it includes mutations for creating, updating, and deleting a song, with clearly defined input types and expected result types.

In the context of a NestJS application utilizing GraphQL, it is crucial to design mutations to be precise in their intent and to return types that provide meaningful feedback about the outcome of the operation, such as the number of affected records. Best practices include careful naming conventions for clarity and the use of non-nullable types (!) where appropriate to enforce the presence of return data, enhancing the robustness of the API. It's also advisable to validate input objects to maintain data integrity and to use custom scalars for complex data types when needed.

```
type Mutation {
    createSong(title: String!): Song!
    updateSong(id: ID!, updateSongInput: UpdateSongInput!): UpdateResult!
    deleteSong(id: ID!): DeleteResult!
}
```

```
type UpdateSongInput {
  title: String
}

type UpdateResult {
  affected: Int!
}

type DeleteResult {
  affected: Int!
}
```

In NestJS, mutations in the schema reflect the capability to alter data within the application, encapsulating operations such as creating, updating, and deleting records. These mutation operations are critical for maintaining dynamic data states within the application and directly correspond to CRUD (Create, Read, Update, Delete) operations that are fundamental to persistent storage management.

Employing the `createSong`, `updateSong`, and `deleteSong` mutations allows for granular control over the song records, with the operations returning types `UpdateResult` and `DeleteResult` that furnish insights into the result of these mutations. As a best practice, ensuring these mutations are well-tested and transactional can help maintain data integrity and provide rollback mechanisms in case of failures. This provides robustness to the data management layer of a NestJS application.

Step 3: Generate Typings

```
npm run generate:typings
```

Resolve Queries and Mutations

Step 1: Refactor type UpdateSongInput to input

In NestJS, when using GraphQL, the `UpdateSongInput` type is defined to specify the shape of data that can be used as input for mutations, particularly for updating songs in this context. This GraphQL input type is crucial for validating the structure of client-provided data before it is processed by the server.

As a standard procedure, generating typings through the `npm run generate:typings` command ensures that the GraphQL schema changes are reflected in the TypeScript types. This step helps to maintain type safety across the application, preventing runtime errors due to type mismatches and facilitating a robust development environment.

```
input UpdateSongInput {
  title: String
}
```

Make sure to generate the typings by running `npm run generate:typings`.

Step 2: Create Song Resolver

This has a GraphQL resolver in a NestJS application, which is a key component for defining how data is fetched. The `SongResolver` class is decorated with `@Resolver()`, indicating its role in handling GraphQL queries, specifically for fetching songs using a service.

It adheres to best practices by leveraging dependency injection to incorporate the `SongService`, ensuring that the resolver remains modular and testable. Utilizing async/await for the `getSongs` query method, it promises a smooth, non-blocking operation that aligns with the scalable and efficient nature of a well-structured NestJS application.

```
import { Query, Resolver } from "@nestjs/graphql";
import { Song } from "src/graphql";
import { SongService } from "./song.service";

@Resolver()
export class SongResolver {
  constructor(private readonly songService: SongService) {}

  @Query("songs")
  async getSongs(): Promise<Song[]> {
    return this.songService.getSongs();
  }
}
```

Step 3 Register the Song Resolver in the Song Module

```
providers: [SongService, SongResolver],
```

Step 4: Resolve Song By a given Id

The `@Query` decorator in a NestJS GraphQL resolver indicates a query operation that fetches data according to the defined schema, with 'song' being the specific query field. The `getSong` method utilizes dependency injection to access the `songService` and execute its `getSong` method, providing a promise that resolves with a `Song` entity, identified by the unique 'id' argument passed to it.

Implementing such methods following the Single Responsibility Principle ensures that each function performs a specific task, improving maintainability and testability. Including comprehensive error

handling within service methods helps to gracefully manage exceptions, ensuring reliable application behavior.

```
@Query('song')
  async getSong(
    @Args('id')
    id: string,
  ): Promise<Song> {
    return this.songService.getSong(id);
  }
```

Resolver with Song Mutation

Step 1: Add CreateSongInput

The GraphQL schema definition below th outlines a mutation operation for creating a song, specifying the structure for client requests. This mutation, `createSong`, takes a non-nullable `CreateSongInput` object as an argument and returns a non-nullable `Song` object, enforcing a contract where both input and output are required.

Incorporating non-nullable fields in GraphQL schema design ensures robustness by explicitly defining required data, which prevents erroneous data entry and streamlines client-server communication. As a best practice, precise input types and return types should be defined, making the API predictable and the codebase maintainable, as well as facilitating smoother frontend development with clearer expectations.

```
createSong(createSongInput: CreateSongInput!): Song!

input CreateSongInput {
  title: String!
}
```

```
npm run generate:typings
```

Step 2: Resolver with createSong mutation

This is a GraphQL resolver for song creation. It illustrates the resolver pattern that is part of NestJS's GraphQL module. The `SongResolver` class contains a mutation operation, which is an essential part of GraphQL's data manipulation language, allowing clients to create new data entries, in this case, a new song.

Incorporating Data Transfer Objects (DTOs), as seen with `CreateSongDTO`, is a recommended practice for ensuring the integrity of the data being transferred between the client and server. This DTO pattern aids in validating incoming data for mutations, enforcing a clear contract for the expected structure of the data, which is especially critical in GraphQL where the schema defines the API's capabilities and constraints.

```
import { CreateSongDTO } from "./dto/create-song-dto";

export class SongResolver {
  @Mutation("createSong")
  async createSong(
    @Args("createSongInput")
    args: CreateSongDTO
  ): Promise<Song> {
    return this.songService.createSong(args);
  }
}
```

Step 3: Resolver with Update Song Mutation

In the context of a GraphQL API with NestJS, the `@Mutation` decorator indicates an operation that modifies server-side data and corresponds to a POST/PUT/PATCH request in a RESTful API. This particular mutation, `updateSong`, is designed to update an existing song resource when provided with an identifier and the new data to apply.

Best practices suggest that mutation operations should be explicit in their intent and tightly scoped to ensure maintainability and clear understanding of their impact. Furthermore, the use of DTOs (Data Transfer Objects) like `UpdateSongDTO` for encapsulating the update data is recommended to enforce validation and encapsulation, which helps to maintain the integrity of the data being sent to the server.

```
@Mutation('updateSong')
  async updateSong(
    @Args('updateSongInput')
    args: UpdateSongDTO,
    @Args('id')
    id: string,
  ): Promise<UpdateResult> {
    return this.songService.updateSong(id, args);
  }
```

Step 4: Resolver with Delete Song Mutation

The `@Mutation` decorator indicates that `deleteSong` is a mutation operation in the GraphQL API, which allows clients to perform write operations, in this case, deleting a song by its identifier.

For maintainability and adherence to best practices, the mutation is handled through a service, encapsulating business logic away from the controller layer. This separation of concerns ensures that the controller is kept lean, with the service being responsible for interfacing with the data access layer to execute the deletion operation.

```
@Mutation('deleteSong')
  async deleteSong(
    @Args('id')
    id: string,
  ): Promise<DeleteResult> {
    return this.songService.deleteSong(id);
  }
```

Error Handling

Error handling is a crucial aspect of developing resilient applications, and within the context of a NestJS GraphQL service, throwing an error from a resolver, such as `getSongs` in `SongResolver`, allows for graceful failure management. This approach can encapsulate business logic validation and operational faults, providing clear feedback to the client about what went wrong.

Incorporating error handling at the resolver level adheres to best practices by localizing error management and maintaining clean separation of concerns. Structured correctly, it can streamline debugging and maintenance, ensuring that the GraphQL API communicates effectively with clients and enhances the overall robustness of the service.

```
@Query('songs')
  async getSongs(): Promise<Song[]> {
    // return this.songService.getSongs();
    // throw new Error('Unable to fetch songs!');
    throw new GraphQLError('Unable to fetch the songs', {
      extensions: {
        code: 'INTERNAL_SERVER_ERROR',
      },
    });
  }
```

In practice, structured error handling is crucial as it aids client applications in deciphering the nature of errors. The inclusion of an error code within the extensions of `GraphQLError` allows for consistent error processing and can be used to trigger specific client-side behavior. It's recommended to handle errors gracefully and provide meaningful error messages to maintain a seamless user experience. Moreover, logging such errors in a manner that they can be monitored and analyzed can lead to improved system robustness and quicker resolution of underlying issues.

Chapter 15 Authenticate GraphQL APIs

Define Schema for Authentication

Error handling is a crucial aspect of developing resilient applications, and within the context of a NestJS GraphQL service, throwing an error from a resolver, such as `getSongs` in `SongResolver`, allows for graceful failure management. This approach can encapsulate business logic validation and operational faults, providing clear feedback to the client about what went wrong.

Incorporating error handling at the resolver level adheres to best practices by localizing error management and maintaining clean separation of concerns. Structured correctly, it can streamline debugging and maintenance, ensuring that the GraphQL API communicates effectively with clients and enhances the overall robustness of the service.

```
@Query('songs')
  async getSongs(): Promise<Song[]> {
    // return this.songService.getSongs();
    // throw new Error('Unable to fetch songs!');
    throw new GraphQLError('Unable to fetch the songs', {
      extensions: {
        code: 'INTERNAL_SERVER_ERROR',
      },
    });
  }
```

In practice, structured error handling is crucial as it aids client applications in deciphering the nature of errors. The inclusion of an error code within the extensions of `GraphQLError` allows for consistent error processing and can be used to trigger specific client-side behavior. It's recommended to handle errors gracefully and provide meaningful error messages to maintain a seamless user experience. Moreover, logging such errors in a manner that they can be monitored and analyzed can lead to improved system robustness and quicker resolution of underlying issues.

Resolve Auth Queries and Mutations

Step 1: Refactoring SignupInput

```
signup(signupInput: SignupInput!): SignupResponse!
```

Renaming from `SingupInput` to `SignupInput` corrects a typographical error, ensuring that the identifier accurately reflects its purpose and adheres to naming conventions. Similarly, updating the field from `acessToken` to `accessToken` rectifies a spelling mistake, which is crucial for maintaining the integrity of the codebase and avoiding potential bugs related to misnamed properties.

The generation of TypeScript typings is an essential step for enforcing type safety within the application, which facilitates the early discovery of errors and enhances the development experience. Implementing a script to automate the generation of these typings ensures consistency and efficiency, which is a recommended practice for maintaining a robust and scalable codebase.

```
npm run generate:typings
```

Step 2: Creating the Auth Resolver

The Auth Resolver in NestJS functions as a bridge between the client's requests and the authentication services, handling operations such as login, registration, and user authentication. Within this resolver, decorators and method handlers work in conjunction to parse client requests, validate user credentials, and return appropriate responses, effectively encapsulating authentication logic in a clean, modular way.

Best practices include implementing robust security measures within the resolver, such as password hashing and token-based authentication, to protect user data. Additionally, it's advantageous to employ middleware for consistent session management across the application, ensuring a secure and seamless user experience.

```
nest g resolver auth
```

Step 2: Resolver with Signup Mutation and Login Query

In a NestJS GraphQL service, the resolver acts as a bridge between the GraphQL API and the underlying data models, facilitating the mutation and query operations. A 'Signup' mutation within the resolver typically handles the creation of new user accounts, managing the input data, user validation, and persistence, while a 'Login' query is responsible for authenticating users, issuing tokens, and ensuring secure access control.

For best practices, the resolver functions must be designed with security as a priority, using practices such as password hashing and validation checks. Additionally, leveraging features such as guards and interceptors can optimize the authentication flow and enhance the application's security posture. It is also recommended to modularize the resolver code for maintenance and readability, separating different functionalities into distinct, manageable units.

```
import { Args, Mutation, Resolver, Query } from "@nestjs/graphql";
import {
  SignupResponse,
  SignupInput,
```

```typescript
  LoginInput,
  LoginResponse,
} from "src/graphql";
import { UsersService } from "src/users/users.service";
import { AuthService } from "./auth.service";

@Resolver()
export class AuthResolver {
  constructor(
    private userService: UsersService,
    private authService: AuthService
  ) {}

  @Mutation("signup")
  async singupUser(
    @Args("signupInput")
    signupInput: SignupInput
  ): Promise<SignupResponse> {
    return this.userService.create(signupInput);
  }

  @Query("login")
  async loginUser(
    @Args("loginInput")
    loginInput: LoginInput
  ): Promise<LoginResponse> {
    return this.authService.login(loginInput);
  }
}
```

Now run the application and test the signup mutation and login request.

Apply Authentication using AuthGuard

Step 1: Define a Profile Query

```graphql
type Query {
  profile: Profile!
}

type Profile {
  email: String!
```

```
    userId: String!
}
```

Defining new TypeScript typings via a dedicated script command enhances the robustness of the codebase by ensuring that data structures are predictable and type checks are enforceable at compile time. It is a sound practice to include this step in continuous integration pipelines for automated checks.

Securing a resolver in NestJS, such as a user profile resolver, is crucial to safeguard sensitive data and ensure that only authenticated users can access their profiles. Implementing a custom AuthGuard is a strategic approach to enforcing authentication and authorization policies on specific application routes or handlers, maintaining the integrity of user data. Integrating guards into the application's lifecycle allows for scalable and manageable security mechanisms, which can be tailored to the evolving needs of the application.

Step 2: Create a Resolver for the Profile object

```
@Query('profile')
@UseGuards(GraphQLAuthGaurd)
getProfile(parent, args, contextValue, info): Profile {
    console.log(parent);
    console.log(args);
    console.log(contextValue);
    console.log(info);
    return contextValue.req.user;
}
```

The `getProfile` function showcased is a query resolver in a GraphQL API built with NestJS, guarded by a custom `GraphQLAuthGaurd`. This resolver retrieves the user profile from the context of an authenticated request, where `contextValue.req.user` is typically populated by the authentication middleware after a user successfully logs in.

Best practices for such resolvers include implementing robust logging strategies to facilitate debugging and introspection, while also ensuring sensitive information is not logged in production environments. Additionally, thorough checks and validation on `args` and `contextValue` should be in place to maintain the integrity of the data and the security of the API.

Step 3: Create a GraphQLAuth authentication guard

In NestJS, guards are a type of component that determine whether a given request will be handled by the route handler or not, typically used for authentication and authorization. Creating a new guard in the `auth` folder encapsulates authentication logic, ensuring that only valid requests with proper credentials can access certain routes, contributing to a secure application architecture.

As a robust practice, this guard should be thoroughly tested to guarantee it behaves as expected under various scenarios, thereby reinforcing the security posture. It is also advisable to keep the

guard's logic decoupled and modular, allowing for reuse across different parts of the application as security requirements evolve.

```typescript
import { AuthenticationError } from "@nestjs/apollo";
import { ExecutionContext, Injectable } from "@nestjs/common";
import { ExecutionContextHost } from
"@nestjs/core/helpers/execution-context-host";
import { GqlExecutionContext } from "@nestjs/graphql";
import { AuthGuard } from "@nestjs/passport";
import { Observable } from "rxjs";

@Injectable()
export class GraphQLAuthGaurd extends AuthGuard("jwt") {
  canActivate(
    context: ExecutionContext
  ): boolean | Promise<boolean> | Observable<boolean> {
    const ctx = GqlExecutionContext.create(context);
    const { req } = ctx.getContext();
    return super.canActivate(new ExecutionContextHost([req]));
  }

  handleRequest<TUser = any>(err: any, user: any): TUser {
    if (err || !user) {
      throw new AuthenticationError("GqlAuthguard");
    }
    return user;
  }
}
```

This custom guard overrides the canActivate and handleRequest methods, facilitating the integration of the authentication guard with the GraphQL execution context. By ensuring the authentication flows seamlessly with GraphQL's unique context handling, the application upholds security across GraphQL resolvers. As a best practice, it is critical to handle errors gracefully and provide informative messages, as demonstrated in the handleRequest method. Additionally, utilizing Observables for asynchronous operations offers a robust approach to handle streams of authentication events, ensuring a reactive system that can scale with the application's needs.

Step 4: Test the Application

In the context of authentication within a NestJS application, sending a login request typically involves submitting user credentials to receive an access token. This token is then used in subsequent requests to access user-specific resources, such as a profile, serving as proof of authentication and authorization.

As a best practice, the access token should be handled securely, often sent in the HTTP Authorization header, and validated on the server for each request to protect against unauthorized

access. Utilizing middleware or guards in NestJS to automate the token validation process can streamline security and ensure consistency across the application.

Subscriptions in GraphQL

Step 1: Installing graphql subscriptions

The `graphql-subscriptions` package is a key element for integrating subscription-based real-time functionality into a Nest.js application, leveraging GraphQL. By implementing subscriptions, the application can push updates to clients in real-time, which is essential for features that require immediate data reflection, such as live chats or real-time feeds.

Incorporating this package into a project follows the best practice of enhancing user experience with dynamic content updates, without the need for manual refreshes. It's advisable to ensure that the use of subscriptions is optimized and does not strain the server with unnecessary loads, and proper authorization mechanisms are implemented to secure real-time data channels.

```
npm install graphql-subscriptions
```

```
GraphQLModule.forRoot<ApolloDriverConfig>({
      installSubscriptionHandlers: true,
    }),
```

Enabling `installSubscriptionHandlers` to `true` within the `AppModule` activates WebSocket support in a NestJS application, which is essential for handling real-time bi-directional communication between the server and clients. This configuration step is critical when the application needs to utilize WebSockets for features such as live chat, real-time feeds, or push notifications.

It is considered good practice to encapsulate the WebSocket configuration within dedicated modules or providers, thus keeping the `AppModule` clean and maintainable. Additionally, thorough testing of WebSocket handlers is recommended to ensure that they handle client communication effectively and securely.

Step 2: Create Subscription

```
type Subscription {
  songCreated: Song!
}
```

A new subscription, `songCreated`, is established, which dispatches notifications with comprehensive details to subscribers whenever a new song is created. In practical applications, akin to an artist

releasing a song on Spotify, the system triggers notifications to inform users about the new release, enhancing user engagement and experience.

Step 3: Resolve the Subscription

A `PubSub` instance facilitates event publishing to specific labels and event subscription for those labels within server code. Creation of a `PubSub` instance is initiated at the top of the `SongResolver`.

Instantiation of a `PubSub` instance at the `SongResolver` commencement allows for event-driven interactions throughout the resolver's scope. This pattern is central to implementing subscription operations effectively in GraphQL, allowing for real-time updates and communication within the application.

```
const pubSub = new PubSub();

@Subscription('songCreated')
  songCreated() {
    return pubSub.asyncIterator('songCreated'); //1
  }
```

The `songCreated` field has been established within the subscription. Upon running the application and initiating the subscription, it actively listens and awaits the event to be published. The `AsyncIterator` specifically listens for events associated with a designated label, facilitating event-driven workflows in the application.
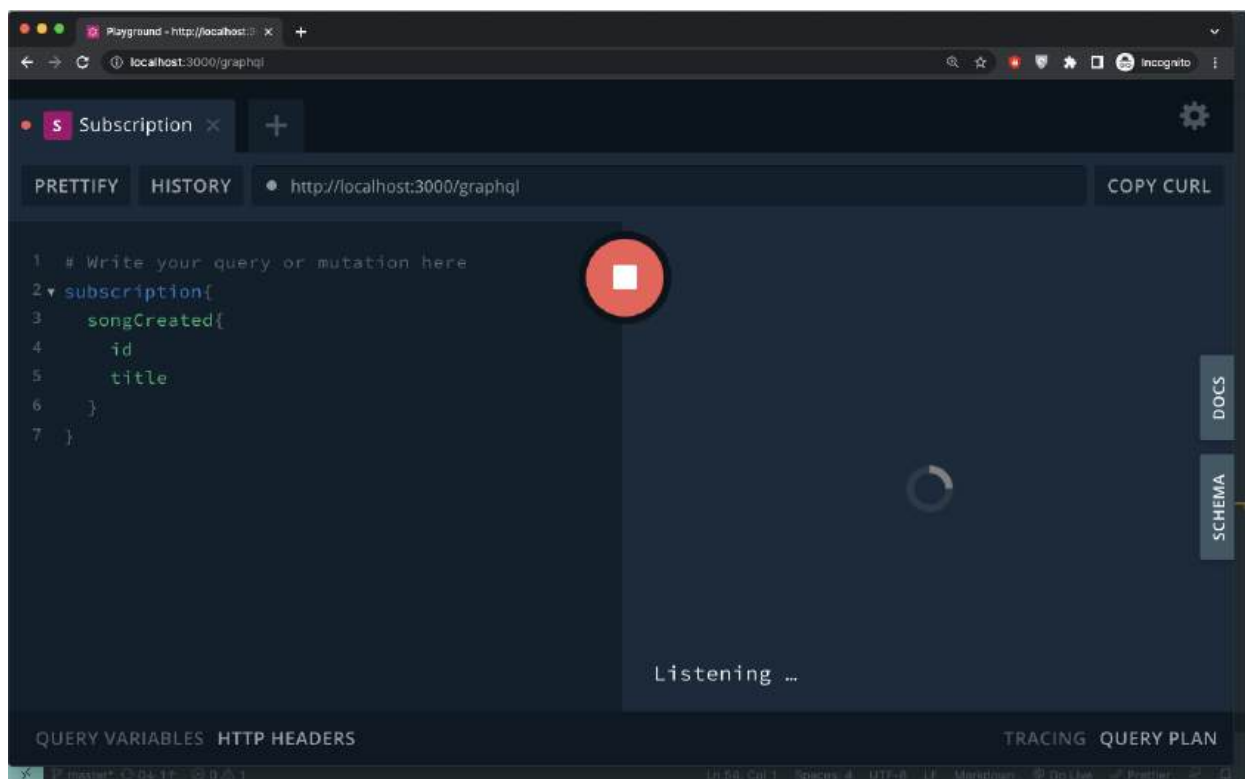
Step 4: Publish the event

```
const newSong = this.songService.createSong(args);
pubSub.publish("songCreated", { songCreated: newSong });
return newSong;
```

The createSong resolver is configured to publish an event, with songCreated being the designated payload. Observers of this event can expect to receive the songCreated payload as demonstrated in the response provided below.
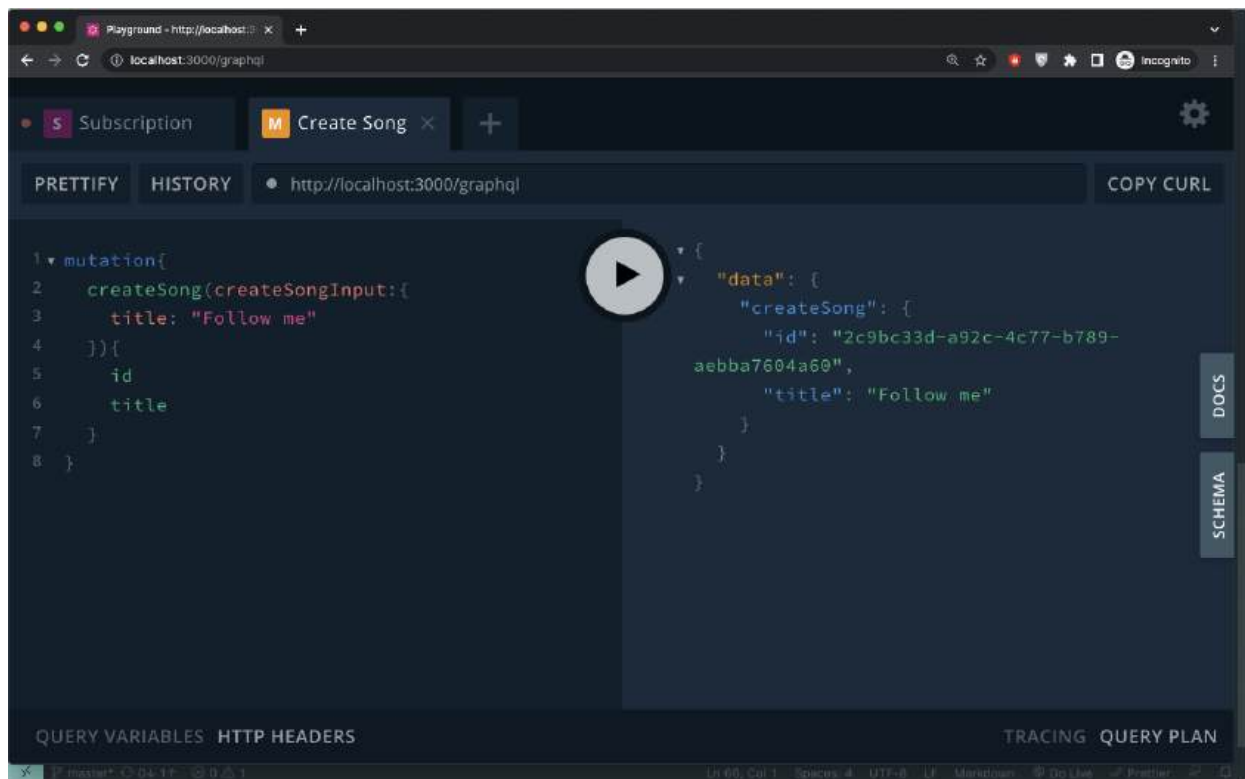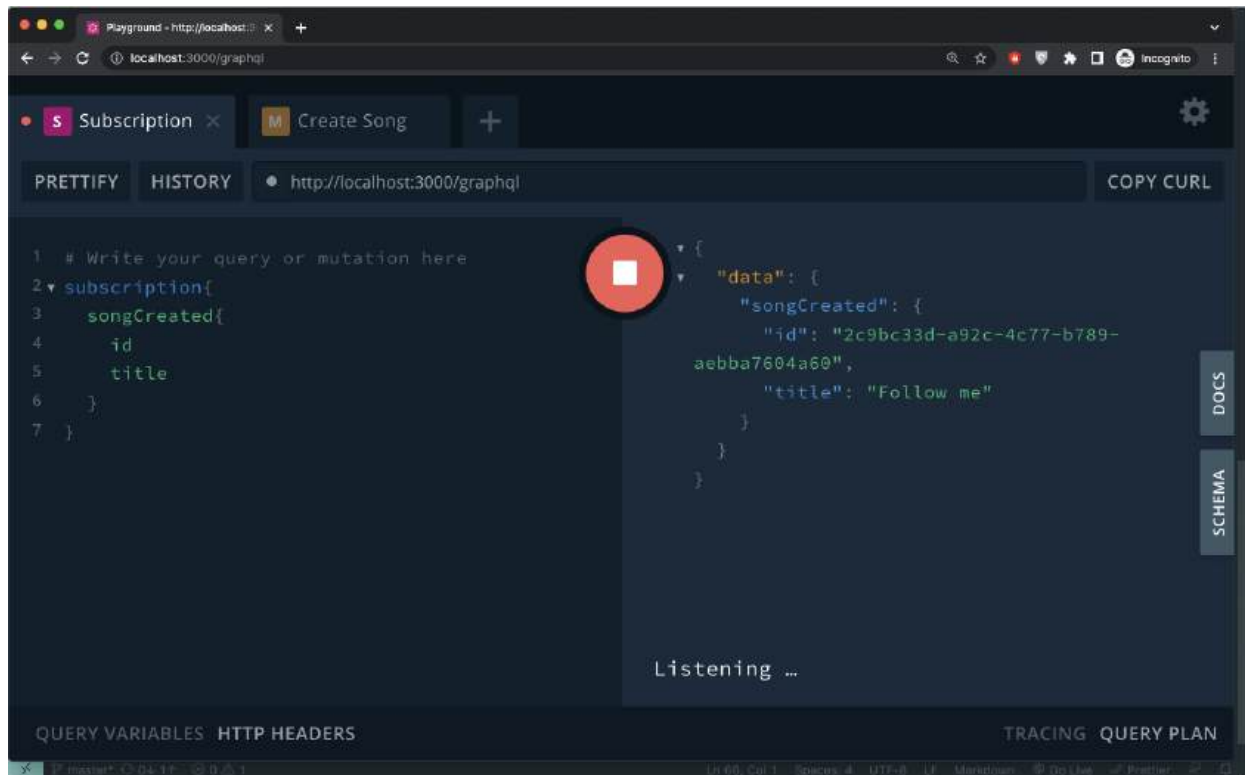
Chapter 16 Testing GraphQL APIs

Unit Test Resolver

Step 1: Run the Test

Running tests in watch mode using `npm run test:watch` allows for continuous feedback during development by monitoring changes to the codebase and executing related tests. If an error arises concerning `SongService` dependencies during the test of `song.resolver.spec.ts`, it suggests that the testing environment may not be correctly simulating the application's service layer, or the service itself may not be properly mocked or instantiated. Regularly updating the test configurations to reflect new dependencies prevents such issues from disrupting the development workflow.

Here is your default `song.resolver` code structure

```
import { Test, TestingModule } from "@nestjs/testing";
import { SongResolver } from "./song.resolver";
```

```
describe("SongResolver", () => {
  let resolver: SongResolver;

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

    resolver = module.get<SongResolver>(SongResolver);
  });

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

Running the test file in question will result in an error. This indicates a potential issue with the configuration or the environment setup for the tests, which necessitates investigation and correction to proceed with successful test execution.

```
SongResolver
    X  should be defined (34 ms)

  ● SongResolver › should be defined

    Nest can't resolve dependencies of the SongResolver (?). Please make sure
that the argument SongService at index [0] is available in the RootTestModule
context.

    Potential solutions:
    - Is RootTestModule a valid NestJS module?
    - If SongService is a provider, is it part of the current RootTestModule?
    - If SongService is exported from a separate @Module, is that module imported
within RootTestModule?
      @Module({
        imports: [ /* the Module containing SongService */ ]
      })
```

Step 2: Mock the SongService

In the context of NestJS and testing, mocking services is essential for isolating units of work and ensuring tests run reliably and efficiently. The implementation of a mock `SongService` would involve creating a simplified version of the service, simulating its behavior and responses, which allows for comprehensive testing of components that depend on it without the overhead of actual database operations.

Mock implementations should closely mimic the actual service logic, providing predictable and controlled outputs for test scenarios. Additionally, it's advisable to keep the mock implementations updated in tandem with their respective services to ensure consistency in test behavior. This approach enhances test reliability and reflects the application's behavior in a controlled environment.

```
providers: [
      SongResolver,
      {
        provide: SongService,
        useValue: {
          getSongs: jest
            .fn()
            .mockResolvedValue([{ id: 'a uuid', title: 'Dancing Feat' }]),
          getSong: jest.fn().mockImplementation((id: string) => {
            return Promise.resolve({ id: id, title: 'Dancing' });
          }),
          createSong: jest
            .fn()
            .mockImplementation((createSongInput: CreateSongInput) => {
              return Promise.resolve({ id: 'a uuid', ...createSongInput });
          }),
          updateSong: jest
            .fn()
            .mockImplementation((updateSongInput: UpdateSongInput) => {
              return Promise.resolve({ affected: 1 });
          }),

          deleteSong: jest.fn().mockImplementation((id: string) => {
            return Promise.resolve({ affected: 1 });
          }),
        },
      },
    ],
```

The practice of crafting mock implementations for services within controllers is a strategic step for testing and development purposes. By simulating the behavior of the `SongService` in the `SongController`, one can validate the interaction and logic without the need for the actual service to be implemented or available.

The update from `CreateSongDTO` to `CreateSongInput` and `UpdateSongDTO` to `UpdateSongInput` reflects a typical progression towards a more structured and type-safe codebase. This change is often indicative of optimizing the code to align with advanced patterns like GraphQL, where `Input` types are commonly used to define the shape of data for mutations, enhancing the code maintainability and type checking.

When the test is run, it will pass now.

Step 3: Test getSongs from the Song Resolver

Testing of resolvers is a strategic approach to verify that the GraphQL query handling is performing as expected. For the `getSongs` resolver test, the goal is to confirm that the resolver accurately fetches and returns the correct set of song data, which is a crucial part of ensuring the GraphQL API's reliability.

A well-constructed test for a resolver, like `getSongs`, not only assesses the returned value against expected results but also encapsulates the performance of underlying services and components. It's a proactive measure to guard against regressions in the API's behavior, especially after updates or refactoring, ensuring that the end-user's queries receive consistent and accurate responses.

```
it("should fetch the songs", async () => {
  const songs = await resolver.getSongs();
  expect(songs).toEqual([{ id: "a uuid", title: "Dancing Feat" }]);
  expect(songs.length).toBe(1);
});
```

Step 4: Unit Test createSong from the Song Resolver

This is similar to getSongs above.

```
it("should create new song", async () => {
  const song = await resolver.createSong({ title: "Animals" });
  expect(song).toEqual({ id: "a uuid", title: "Animals" });
});
```

Step 5: Unit Test updateSong from the Song Resolver

The updateSong method within the Song Resolver would have a dedicated test that mocks dependencies, inputs a known entity, and asserts that the method returns the updated song object correctly.

```
it("should update the song", async () => {
  const song = await resolver.updateSong("a uuid", { title: "DANCING FEAT" });
  expect(song.affected).toBe(1);
});
```

Step 6: Test deleteSong from the Song Resolver

Similarly, testing the deleteSong method requires setting up a scenario where a song is marked for deletion, invoking the method, and subsequently verifying that the method effectively removes the target entity from the system. This would typically involve checking for the absence of the song's identifier in the dataset post-operation. This is important for maintaining data integrity and confirming that the application logic conforms to requirements. Adding mock data that resembles real-world

cases and asserting on both the returned value and the final state of the dataset are additional layers that enhance test effectiveness.

```
it("should delete the song", async () => {
  const song = await resolver.deleteSong("a uuid");
  expect(song.affected).toBe(1);
});
```

End to End Test

Step 1: Adding End to End watch script to package.json file

Adding an End-to-End (E2E) watch script to the `package.json` file allows for the continuous observation and testing of the application as changes are made, facilitating immediate feedback and efficient issue detection. This script is configured to run the E2E testing suite in a watch mode, which monitors for file changes and automatically reruns tests, proving essential for iterative development and enhancing productivity.

The integration of a watch mode for E2E tests into the development workflow can significantly improve code quality by enabling developers to identify and resolve integration issues as they code. It acts as a real-time safeguard against regressions and defects, which is a reflection of high standards in software development and maintenance.

```
"test:e2e:watch": "jest --watch --detectOpenHandles --config
./test/jest-e2e.json"
```

Step 2: Create a new E2E Testing File

You can have to create a new testing file in the `test/song/song.e2e-spec.ts`

Step 3: Setup a E2E Testing File

```
import { Test, TestingModule } from "@nestjs/testing";
import { INestApplication } from "@nestjs/common";
import * as request from "supertest";
import { AppModule } from "../song/../../src/app.module";
import { TypeOrmModule } from "@nestjs/typeorm";
import { Song } from "../song/../../src/song/song.entity";
import { SongModule } from "../song/../../src/song/song.module";
import { CreateSongDTO } from "../song/../../src/song/dto/create-song-dto";
```

```javascript
describe("Song Resolver (e2e)", () => {
  let app: INestApplication;

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

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

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [
        TypeOrmModule.forRoot({
          type: "postgres",
          url: "postgres://postgres:root@localhost:5432/test-dev",
          synchronize: true,
          entities: [Song],
          dropSchema: true,
        }),
        SongModule,
      ],
    }).compile();

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

  afterEach(async () => {
    const songRepository = app.get("SongRepository");
    await songRepository.clear();
  });

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

  const createSong = (createSongDTO: CreateSongDTO): Promise<Song> => {
    const song = new Song();
    song.title = createSongDTO.title;
    const songRepo = app.get("SongRepository");
    return songRepo.save(song);
  };
```

```
  it("/ (GET)", () => {
    return request(app.getHttpServer())
      .get("/")
      .expect(200)
      .expect("Hello World!");
  });
});
```

To ensure test accuracy and environment isolation, the Songs table in the database must be cleared after each test is executed. This practice prevents state leakage between tests and maintains consistency.

The file to initiate End-to-End testing in watch mode is executed with the command `npm run test:e2e:watch`. This initiates a process that continually monitors for any changes in the codebase and reruns tests accordingly, allowing for immediate feedback and efficient debugging.

Step 4: Test the Songs Query

```
it("(Query) it should get all songs with songs query", async () => {
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `query {
        songs {
          id
          title
        }
      }`,
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData);

  expect(results.statusCode).toBe(200);
  expect(results.body).toEqual({ data: { songs: [newSong] } });
});
```

1. Testing queries in Nest.js, especially when involving a database, begins by ensuring that the relevant records exist to be queried against. The setup phase of a test typically includes invoking methods like `createSong` to insert necessary data into the Songs table, resulting in a controlled and predictable testing environment.
2. The formation of GraphQL queries is facilitated through the use of structured variables, such as `queryData`, which hold the GraphQL query string. These queries are then utilized to interact with the GraphQL API, which, by convention, operates under a single endpoint, typically `/graphql`.

3. To execute these queries within a test suite, the query data must be provided to the `send` method of the testing tool. This simulates a client request to the server, allowing for the testing of query handling and response accuracy.
4. In high-quality software engineering, a tip is to ensure the testing environment closely mimics production conditions without affecting real data, maintaining isolation between tests for reliability. Additionally, structuring test cases to reflect a variety of realistic scenarios can uncover edge cases and potential bugs before deployment.

Step 5: Test A Single Song Query

```
it("(Query) it should get a song by id", async () => {
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `query GetSong($id: ID!){
        song(id: $id){
          title
          id
        }
      }`,
    variables: {
      id: newSong.id,
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData)
    .expect(200);

  expect(results.body).toEqual({ data: { song: newSong } });
});
```

1. Defining a Query in GraphQL with specific operation names and required variables, such as `GetSong($id: ID!)`, clarifies the expected input and makes the query reusable. Variables in GraphQL, like `$id`, are used to parameterize queries, allowing for dynamic input that can be varied with each query execution.
2. Supplying the `$id` variable within a `song` query allows for the specification of which particular song is being requested. This is a part of query composition in GraphQL, where the query variables are declared and then used within the query body to fetch data.
3. The `variables` section in a GraphQL operation is where the actual values for the declared variables are specified. This is where one sets the specific `ID` value for `$id` when executing the query, thereby allowing for the retrieval of data corresponding to that particular identifier.

Step 6: Test The Creation of a aong mutation

```
it("(Mutation) it should create a new song", async () => {
  const queryData = {
```

```
    query: `mutation CreateSong($createSongInput: CreateSongInput!){
        createSong(createSongInput: $createSongInput){
          title
          id
        }
      }`,
    variables: {
      createSongInput: {
        title: "Animals",
      },
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData)
    .expect(200);

  expect(results.body.data.createSong.title).toBe("Animals");
});
```

Step 7: Testing The Update Song Mutation

The testing of the 'Create Song Mutation' in Nest.js involves invoking the mutation responsible for creating a new song entry and verifying that the operation successfully adds the song to the application state or database. The test ensures that the mutation receives the correct input and produces the expected song record, consistent with defined data schemas and business logic.

```
it("(Mutation) it should update existing song", async () => {
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `mutation UpdateSong($id: ID!, $updateSongInput: UpdateSongInput!){
        updateSong(id: $id, updateSongInput: $updateSongInput){
          affected
        }
      }`,
    variables: {
      id: newSong.id,
      updateSongInput: {
        title: "Lover",
      },
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData)
```

```
      .expect(200);

  expect(results.body.data.updateSong.affected).toBe(1);
});
```

Step 8: Test The Delete Song Mutation

Testing the 'Update Song Mutation' entails executing the mutation that modifies an existing song entry and confirming that the changes are accurately persisted. This test asserts that the mutation not only accepts valid input but also integrates seamlessly with the underlying data handling layers, effectively maintaining data integrity throughout the operation.

```
it("(Mutation) it should delete existing song", async () => {
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `mutation DeleteSong($id: ID!){
        deleteSong(id: $id){
          affected
        }
      }`,
    variables: {
      id: newSong.id,
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData)
    .expect(200);

  expect(results.body.data.deleteSong.affected).toBe(1);
});
```

For optimal results, these tests would typically include checks for handling of invalid inputs and edge cases to ensure resilience. Leveraging a suite of such mutation tests establishes a safeguard against regressions and aids in maintaining a stable and reliable codebase as new features are developed.

Chapter 17 GraphQL Advanced Concepts

Server Side Caching

Step 1: Adding End to End watch script to package.json file

Adding an End-to-End (E2E) watch script to the `package.json` file allows for the continuous observation and testing of the application as changes are made, facilitating immediate feedback and efficient issue detection. This script is configured to run the E2E testing suite in a watch mode, which monitors for file changes and automatically reruns tests, proving essential for iterative development and enhancing productivity.

The integration of a watch mode for E2E tests into the development workflow can significantly improve code quality by enabling developers to identify and resolve integration issues as they code. It acts as a real-time safeguard against regressions and defects, which is a reflection of high standards in software development and maintenance.

```
"test:e2e:watch": "jest --watch --detectOpenHandles --config
./test/jest-e2e.json"
```

Step 2: Create a new E2E Testing File

You can have to create a new testing file in the `test/song/song.e2e-spec.ts`

Step 3: Setup a E2E Testing File

```ts
import { Test, TestingModule } from "@nestjs/testing";
import { INestApplication } from "@nestjs/common";
import * as request from "supertest";
import { AppModule } from "../song/../../src/app.module";
import { TypeOrmModule } from "@nestjs/typeorm";
import { Song } from "../song/../../src/song/song.entity";
import { SongModule } from "../song/../../src/song/song.module";
import { CreateSongDTO } from "../song/../../src/song/dto/create-song-dto";

describe("Song Resolver (e2e)", () => {
  let app: INestApplication;

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

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

```
  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [
        TypeOrmModule.forRoot({
          type: "postgres",
          url: "postgres://postgres:root@localhost:5432/test-dev",
          synchronize: true,
          entities: [Song],
          dropSchema: true,
        }),
        SongModule,
      ],
    }).compile();

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

  afterEach(async () => {
    const songRepository = app.get("SongRepository");
    await songRepository.clear();
  });

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

  const createSong = (createSongDTO: CreateSongDTO): Promise<Song> => {
    const song = new Song();
    song.title = createSongDTO.title;
    const songRepo = app.get("SongRepository");
    return songRepo.save(song);
  };

  it("/ (GET)", () => {
    return request(app.getHttpServer())
      .get("/")
      .expect(200)
      .expect("Hello World!");
  });
});
```

To ensure test accuracy and environment isolation, the Songs table in the database must be cleared after each test is executed. This practice prevents state leakage between tests and maintains consistency.

The file to initiate End-to-End testing in watch mode is executed with the command `npm run test:e2e:watch`. This initiates a process that continually monitors for any changes in the codebase and reruns tests accordingly, allowing for immediate feedback and efficient debugging.

Step 4: Test the Songs Query

```
it("(Query) it should get all songs with songs query", async () => {
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `query {
        songs {
          id
          title
        }
      }`,
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData);

  expect(results.statusCode).toBe(200);
  expect(results.body).toEqual({ data: { songs: [newSong] } });
});
```

1. Testing queries in Nest.js, especially when involving a database, begins by ensuring that the relevant records exist to be queried against. The setup phase of a test typically includes invoking methods like `createSong` to insert necessary data into the Songs table, resulting in a controlled and predictable testing environment.
2. The formation of GraphQL queries is facilitated through the use of structured variables, such as `queryData`, which hold the GraphQL query string. These queries are then utilized to interact with the GraphQL API, which, by convention, operates under a single endpoint, typically `/graphql`.
3. To execute these queries within a test suite, the query data must be provided to the `send` method of the testing tool. This simulates a client request to the server, allowing for the testing of query handling and response accuracy.
4. In high-quality software engineering, a tip is to ensure the testing environment closely mimics production conditions without affecting real data, maintaining isolation between tests for reliability. Additionally, structuring test cases to reflect a variety of realistic scenarios can uncover edge cases and potential bugs before deployment.

Step 5: Test A Single Song Query

```
it("(Query) it should get a song by id", async () => {
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `query GetSong($id: ID!){
```

```
        song(id: $id){
          title
          id
        }
      }`,
    variables: {
      id: newSong.id,
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData)
    .expect(200);

  expect(results.body).toEqual({ data: { song: newSong } });
});
```

1. Defining a Query in GraphQL with specific operation names and required variables, such as GetSong($id: ID!), clarifies the expected input and makes the query reusable. Variables in GraphQL, like $id, are used to parameterize queries, allowing for dynamic input that can be varied with each query execution.
2. Supplying the $id variable within a song query allows for the specification of which particular song is being requested. This is a part of query composition in GraphQL, where the query variables are declared and then used within the query body to fetch data.
3. The variables section in a GraphQL operation is where the actual values for the declared variables are specified. This is where one sets the specific ID value for $id when executing the query, thereby allowing for the retrieval of data corresponding to that particular identifier.

Step 6: Test The Creation of a aong mutation

```
it("(Mutation) it should create a new song", async () => {
  const queryData = {
    query: `mutation CreateSong($createSongInput: CreateSongInput!){
        createSong(createSongInput: $createSongInput){
          title
          id
        }
      }`,
    variables: {
      createSongInput: {
        title: "Animals",
      },
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
```

```
      .send(queryData)
      .expect(200);

    expect(results.body.data.createSong.title).toBe("Animals");
});
```

Step 7: Testing The Update Song Mutation

The testing of the 'Create Song Mutation' in Nest.js involves invoking the mutation responsible for creating a new song entry and verifying that the operation successfully adds the song to the application state or database. The test ensures that the mutation receives the correct input and produces the expected song record, consistent with defined data schemas and business logic.

```
it("(Mutation) it should update existing song", async () => {
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `mutation UpdateSong($id: ID!, $updateSongInput: UpdateSongInput!){
        updateSong(id: $id, updateSongInput: $updateSongInput){
          affected
        }
    }`,
    variables: {
      id: newSong.id,
      updateSongInput: {
        title: "Lover",
      },
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData)
    .expect(200);

    expect(results.body.data.updateSong.affected).toBe(1);
});
```

Step 8: Test The Delete Song Mutation

Testing the 'Update Song Mutation' entails executing the mutation that modifies an existing song entry and confirming that the changes are accurately persisted. This test asserts that the mutation not only accepts valid input but also integrates seamlessly with the underlying data handling layers, effectively maintaining data integrity throughout the operation.

```
it("(Mutation) it should delete existing song", async () => {
```

```
  const newSong = await createSong({ title: "Animals" });
  const queryData = {
    query: `mutation DeleteSong($id: ID!){
        deleteSong(id: $id){
          affected
        }
      }`,
    variables: {
      id: newSong.id,
    },
  };
  const results = await request(app.getHttpServer())
    .post("/graphql")
    .send(queryData)
    .expect(200);

  expect(results.body.data.deleteSong.affected).toBe(1);
});
```

For optimal results, these tests would typically include checks for handling of invalid inputs and edge cases to ensure resilience. Leveraging a suite of such mutation tests establishes a safeguard against regressions and aids in maintaining a stable and reliable codebase as new features are developed.

Optimize Query Performance using DataLoader

What is DataLoader?

In GraphQL, a dataloader is a mechanism used to efficiently batch and cache requests made to a backend data source. It is particularly useful in scenarios where multiple GraphQL queries or mutations request the same or overlapping data, leading to redundant or redundant data-fetching operations. The dataloader helps to optimize these scenarios by reducing the number of database or API calls, resulting in improved performance and reduced latency.

The idea behind a data loader is to collect multiple individual data requests and then fetch the data in batches, minimizing the number of trips to the data source. It acts as a middleware or utility between the GraphQL resolver functions and the data source, sitting at the data access layer.

Here's how a data loader typically works:

1. Request Batching: When a resolver needs to fetch data from the data source, it registers the request with the dataloader instead of making the direct call immediately. The dataloader collects these requests during the GraphQL query execution phase.

2. Batch Execution: Once the execution phase is complete, the dataloader identifies identical or overlapping requests and groups them into batches. It then executes these batches in one or more optimized calls to the underlying data source.
3. Caching: The dataloader often incorporates a caching mechanism to store and reuse fetched data for future requests, further reducing the need for redundant data fetches.

By using a dataloader, GraphQL servers can avoid the "N+1" problem, where resolving N objects in a GraphQL query requires N+1 database or API calls. Instead, the dataloader can efficiently resolve multiple objects with a smaller number of data source calls.

Dataloader implementations are available for various programming languages and frameworks that support GraphQL, making it easier for developers to optimize data fetching in their GraphQL APIs. For example, in JavaScript, the `dataloader` library is commonly used for this purpose.

Run the Application without DataLoader

Running an application without DataLoader in a Nest.js GraphQL context can result in increased numbers of data retrieval operations, potentially leading to what is known as the "N+1" problem. Run the application to see what will happen.

```
type User {
  id: Int!
  name: String!
}

type Post {
  id: String!
  title: String!
  body: String!
  createdBy: User!
}

type Query {
  posts: [Post!]!
  users: [User!]!
}


{
    posts{
        id
        title
        body
        createdBy{
            id
            name
        }
```

```
    }
}
```

Upon sending the specified request to the GraphQL server, the system initiates a database query for the `createdBy` field within the `users` table. DataLoader addresses the inefficiency by batching user IDs and caching them.

Execution of the aforementioned request results in console outputs such as `FETCHING DATA FROM DB: Getting user with id 1...` multiple times for the same user ID. This indicates repetitive fetching of user records for each post by the `createdBy` resolver. DataLoader mitigates this issue by batching requests and utilizing caching mechanisms.

Step 1: Create UsersLoader in the users folder

Ensure that the `dataloader` package is installed.

The creation of `users.loader` should take place within the user's directory.

```typescript
import * as DataLoader from "dataloader";

import { mapFromArray } from "../util";
import { User } from "./user.entity";
import { UsersService } from "./users.service";

export function createUsersLoader(usersService: UsersService) {
  return new DataLoader<number, User>(async (ids) => {
    const users = await usersService.getUsersByIds(ids);

    const usersMap = mapFromArray(users, (user) => user.id);

    console.log("usersMap", usersMap);
    const results = ids.map((id) => usersMap[id]);
    console.log("results", results);
    return results;
  });
}
```

This specific setup creates a DataLoader that batches user retrieval by their identifiers, leveraging a service method that fetches users in bulk and a utility function to map the resulting array to an object for constant-time lookups. This pattern minimizes unnecessary database queries and is a resource-efficient strategy for fetching associated data, such as user information, by aggregating multiple queries into a single batch. This approach is reflected in the asynchronous function passed to the DataLoader constructor, which organizes the users into a map and resolves the batch of user requests in an optimized sequence that aligns with the initial ordering of IDs.

Step 2: Register Loader in Context

```
@Module({
  imports: [
    PostsModule,
    GraphQLModule.forRootAsync({
      imports: [UsersModule],
      useFactory: (usersService: UsersService) => ({
        autoSchemaFile: join(process.cwd(), 'src/schema.gql'),
        context: () => ({
          randomValue: Math.random(),
          usersLoader: createUsersLoader(usersService),
        }),
      }),
      inject: [UsersService],
    }),
  ],
})
```

The above configuration within a `@Module` decorator is setting up a GraphQL module with asynchronous importation. This approach ensures that dependencies such as `UsersModule` are loaded correctly and their services, like `UsersService`, are available for use within the GraphQL context.

For efficient data fetching and batching, the `usersLoader` is instantiated through the `createUsersLoader` function, providing a DataLoader instance specifically for user data. This setup is vital for optimizing GraphQL query performance by reducing the number of data access requests, particularly beneficial in scenarios with complex data loading requirements such as nested queries.

Step 3: Load data from a loader in PostResolver

Loading data from a loader in the context of a Nest.js resolver, such as `PostResolver`, involves a key strategy for optimizing data retrieval when dealing with GraphQL queries. It leverages tools like DataLoader to batch and cache data requests, which significantly enhances performance by reducing the number of database queries. It's a recommended practice to implement this pattern when dealing with GraphQL resolvers to ensure efficient and scalable data fetching operations.

```
@ResolveField('createdBy', () => User)
getCreatedBy(
  @Parent() post: Post,
  @Context('usersLoader') usersLoader: DataLoader<number, User>,
) {
  const { userId } = post;
  return usersLoader.load(userId);
}
```

Step 4: Run the Application

In the console, you will observe the following output: `Getting users with IDs (1, 2, 4)`. This output signifies that the DataLoader has efficiently consolidated multiple database requests into a single query, fetching all the users associated with each post, optimizing data retrieval and reducing database load.

Fetching Data from External API

What is RESTDatasource in Apollo

In Apollo Server, a RESTDataSource is a built-in class provided by the apollo-datasource-rest package that facilitates fetching data from RESTful APIs and integrating it with your GraphQL server. It acts as a data source for your GraphQL resolvers, allowing you to easily make HTTP requests to external REST APIs and transform the responses into GraphQL-compatible data.

The RESTDataSource class provides several benefits:

1. HTTP Requests: It abstracts away the details of making HTTP requests, including handling GET, POST, PUT, DELETE, etc., so you can focus on fetching data from the API.
2. Caching: It has built-in caching support, which can improve the performance of your GraphQL server by reducing redundant requests to the same REST API endpoints.
3. Error Handling: It automatically handles errors and HTTP status codes, making it easier to manage error responses from the REST API.
4. GraphQL Integration: It maps the fetched data to your GraphQL schema, allowing you to define resolvers that directly use the methods provided by RESTDataSource to fetch data.

We are going to fetch data from the `jsonplanceholder` api. We will this endpoint `https://jsonplaceholder.typicode.com/todos` to fetch all the todos.

Step 1: Create Schema file

```
type Todo {
  id: ID!
  userId: Int!
  title: String!
  completed: Boolean
}

type Query {
  todos: [Todo!]!
}
```

Step 2: Implement RESTDatasource in TodoService

```typescript
import { RESTDataSource } from "@apollo/datasource-rest";
import { Injectable } from "@nestjs/common";

@Injectable()
export class TodoService extends RESTDataSource {
  constructor() {
    super();
    this.baseURL = "https://jsonplaceholder.typicode.com";
  }
  async getTodos() {
    return this.get("/todos");
  }
}
```

Step 3: Use DataSource in TodosResolver

```typescript
@Resolver()
export class TodoResolver {
  @Query("todos")
  async getTodods(
    @Context("dataSources")
    dataSources
  ) {
    return dataSources().todoAPI.getTodos();
  }
}
```

Step 4: Add Datasource in context

```typescript
GraphQLModule.forRoot<ApolloDriverConfig>({
    driver: ApolloDriver,
    typePaths: ['./**/*.graphql'],
    definitions: {
      path: join(process.cwd(), 'src/graphql.ts'),
      outputAs: 'class',
    },
    context: async () => ({
      dataSources,
    }),
  }),
```

Chapter 18 Prisma Integration with Nest.js

Setup Prisma

What is Prisma?

Prisma is different than traditional Object-Relational Mapping (ORM). Unlike traditional ORMs, Prisma prioritizes the generation of a strongly-typed query API that closely mirrors the database schema. This approach simplifies database operations, eliminates the need for manual SQL queries, and enforces robust type safety throughout development. Prisma enjoys popularity in the modern JavaScript and TypeScript ecosystem and finds extensive use in Node.js applications and frameworks like Nest.js.

The upcoming endeavor involves constructing REST APIs utilizing Prisma, harnessing its capabilities to streamline database-related tasks within the API development process.

Prisma encompasses three primary components:

1. **Prisma Client:** It functions as a potent query builder and database client, enabling type-safe interaction with the database using TypeScript or JavaScript. By automatically generating a query API based on the data model and schema, it delivers typing and autocompletion support during database queries.
2. **Prisma Migrate:** This tool empowers you to manage database schema changes in a versioned and controlled manner. Prisma Migrate simplifies the application of migrations, ensuring your database schema remains synchronized with your data model.
3. **Prisma Studio:** This tool provides a visual interface for database management, offering an intuitive way to inspect and manipulate database data directly from the browser. It facilitates tasks such as viewing, editing, and exploring data stored in your database.

Prisma supports multiple database connectors, including PostgreSQL, MySQL, SQLite, and SQL Server, offering compatibility with various database systems.

Step 1: Setup Prisma

To begin, install the `prisma` package. The starter kit code is provided, accessible on the starter branch, where running `npm install` will handle the installation of all required dependencies. This project serves as a fundamental Nest.js application, necessitating the installation of dependencies through the `npm install` command.

Step 2: Create Database

You have to create a database for the prisma project. You can choose any name for the database. Please create the database by using `PGAdmin`

Step 3: Install Prisma

Now you need to install prisma as a dev dependency

```
npm install -D prisma
```

Step 4: Initialize Prisma Project

```
npx prisma init
```

This will create the prisma folder and `schema.prisma` file. It will also create the `.env` file.

```
# Environment variables declared in this file are automatically made available to
Prisma.
# See the documentation for more detail:
https://pris.ly/d/prisma-schema#accessing-environment-variables-from-the-schema

# Prisma supports the native connection string format for PostgreSQL, MySQL,
SQLite, SQL Server, MongoDB and CockroachDB.
# See the documentation for all the connection string options:
https://pris.ly/d/connection-strings

DATABASE_URL="postgresql://postgres:root@localhost:5432/nest-prisma-db?schema=public"
```

You will see the `prisma.schema` file.

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

Models and Migrations

What is Prisma Model

In Prisma, a model is a fundamental building block that represents a structured data entity in your application. A model defines the structure of a database table and maps it to a corresponding TypeScript/JavaScript class. Models are used to interact with the database and provide a type-safe API for querying and manipulating data.

When using Prisma, you define models in a declarative manner using the Prisma schema language. The Prisma schema language allows you to define the data model of your application, including entities, fields, data types, relationships, and constraints.

Create Prisma Model

We are going to create our first Model in this lesson

```
model Song {
  id    Int    @id @default(autoincrement())
  title String
}
```

We have created a song model with id and title. I have set the `id` to autoincrement automatically

Run Migrations

Migrations in Prisma are a way to manage and apply changes to your database schema in a version-controlled and systematic manner. They allow you to make changes to your Prisma schema and then apply those changes to the actual database without losing data or compromising its integrity.

When you modify your Prisma schema to add, remove, or alter models or fields, you create a new migration to represent those changes. Migrations are recorded as code files and are stored in the `prisma/migrations` directory of your project. Each migration file contains the steps required to apply or revert a specific change to the database schema.

Key aspects of Prisma migrations include:

1. **Version Control**: Migrations are managed by Prisma and stored as code files. These files are stored in your version control system (such as Git), enabling collaboration and tracking changes over time.
2. **Idempotent Operations**: Migrations are designed to be idempotent, meaning that applying the same migration multiple times results in the same outcome. This ensures that migrating a database to a specific state is a repeatable process.
3. **Data Preservation**: Prisma migrations are designed to be data-safe, meaning that data in the database is preserved during the migration process. When you alter a table or add new fields, the existing data is migrated or transformed accordingly.
4. **Rollback Support**: Migrations also support rollback, which allows you to revert a previously applied migration and restore the database to its previous state. This feature is useful when dealing with errors or when undoing specific changes.

To use migrations in Prisma, you need to have Prisma Migrate installed. Prisma Migrate is a separate component of the Prisma toolkit that provides migration functionality

Let's run the migration using `npx prisma migrate dev --name=init`

I have provided the name of the migration init. You can provide any name for your migrations

This migration command will generate the database tables for you. You can check the tables from your PGAdmin. It will also generate the Typescript typings for you with the help of `PrismaClient`

Creating Prisma Service

Step 1: Setup Prisma Client

`PrismaClient` will generate the typescript typings for your models. `PrismaClient` will also provide the `CRUD` functionality. Let's install the Prisma client

npm install @prisma/client

Step 2: Create PrismaService

```
import { Injectable, OnModuleInit } from "@nestjs/common";
import { PrismaClient } from "@prisma/client";

@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
  onModuleInit() {
    this.$connect();
  }
}
```

You have to create a new Prisma service inside the `src` folder. This PrismaService will make the connect with PrismaClient and interact with DB

Create, FindOne and Find

Step 1: Create a Resource

We are going to use the nestjs crud generator to generate api endpoints

```
nest g resource songs
```

You have to select the REST API and Endpoint to yes. This command will generate the songs module, controller, and service for you

Step 2: Update SongService

```
export class SongsService {
  constructor(private prisma: PrismaService) {} //inject the prismaService as a
dependency

  create(createSongDto: Prisma.SongCreateInput) {
    return this.prisma.song.create({
      data: createSongDto,
    });
  }

  findAll() {
    return this.prisma.song.findMany(); //find all the records
  }

  findOne(id: number) {
    return this.prisma.song.findUnique({ where: { id } }); // find unique single
record based on id
  }
}
```

We are going to use the types from the Prisma Client package. PrismaClient has generated many types of us you can find all of the types in `node_modules/.prisma/client/index.d.ts`

I have used the `SongCreateInput` for createSongDTO. You can find this type in `index.d.ts` file

```
export type SongCreateInput = {
  title: string;
  artist?: ArtistCreateNestedOneWithoutSongsInput;
};
```

Step 3: Update SongController

You can access the `song` object from the Prisma object. This `song` object is similar to `SongRepository` in TypeORM

We also have to update the type in the create method in `SongController`

```
@Post()
  create(@Body() createSongDto: Prisma.SongCreateInput) {
```

```
        return this.songsService.create(createSongDto);
    }
```

Step 4: Register PrismaService

```
providers: [SongsService, PrismaService];
```

You have to register the `PrismaService` in `SongModule`

Step 5: Test the Application

You have to run the Application and test create, find, and findone endpoints

```
### Create Song
POST http://localhost:3000/songs
Content-Type: application/json

{
    "title" : "FOLLOW ME"
}

### FETCH ALL SONGS
GET http://localhost:3000/songs


### FETCH SONG BY ID
GET http://localhost:3000/songs/2
```

Update and Delete

Step 1: Implement the Update method

```
//songs.service.ts

 update(
    where: Prisma.SongWhereUniqueInput,
    updateSongDto: Prisma.SongUpdateInput,
  ) {
    return this.prisma.song.update({
      where,
      data: updateSongDto,
    });
```

```
    }
```

I have used the `SongWhereUniqueInput`. You can find this type from `PrismaClient`. I have also used the `SongUpdateInput` from `PrismaClient`

```typescript
export type SongWhereUniqueInput = Prisma.AtLeast<
  {
    id?: number;
    AND?: SongWhereInput | SongWhereInput[];
    OR?: SongWhereInput[];
    NOT?: SongWhereInput | SongWhereInput[];
    title?: StringFilter<"Song"> | string;
    artistId?: IntNullableFilter<"Song"> | number | null;
    artist?: XOR<ArtistNullableRelationFilter, ArtistWhereInput> | null;
  },
  "id"
>;

export type SongUpdateInput = {
  title?: StringFieldUpdateOperationsInput | string;
  artist?: ArtistUpdateOneWithoutSongsNestedInput;
};
```

Step 2: Update Controller

```typescript
update(
    @Param('id') id: string,
    @Body() updateSongDto: Prisma.SongUpdateInput,
  ) {
    return this.songsService.update({ id: +id }, updateSongDto);
  }
```

Step 3: Delete Song

```typescript
//songs.service.ts
remove(where: Prisma.SongWhereUniqueInput) {
    return this.prisma.song.delete({ where });
  }

//songs.controller.ts
  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.songsService.remove({ id: +id });
  }
```

One to Many Relation

Step 1: Define Relation

- An Artist can publish many songs
- Many Songs can belong to a single Artist

```
model Song {
  id       Int      @id @default(autoincrement())
  title    String
  artist   Artist? @relation(fields: [artistId], references: [id])
  artistId Int?
}

model Artist {
  id     Int     @id @default(autoincrement())
  title String
  name   String
  songs Song[]
}
```

- It uses the `@relation` attribute to define the relationship. The `@relation` attribute has two arguments: `fields` and `references`. It specifies that the `artist` field in the `Song` model references the `id` field in the `Artist` model.
- `artistId`: This field is of type `Int` and represents the foreign key that links a `Song` to its corresponding `Artist`. It is optional (`Int?`) to allow for songs that may not have an associated artist.

Step 2: Generate Artist Resource

First of all, you have to run the migrations for your one-to-many relation

```
npx prisma migrate dev --name=one-to-many-relation

nest g resource artists
```

Step 3: Create Artist

```
//artists.service.ts
export class ArtistsService {
  constructor(private prisma: PrismaService) {}
```

```
  create(createArtistDto: Prisma.ArtistCreateInput) {
    return this.prisma.artist.create({
      data: createArtistDto,
    });
  }
}
```

//artists.controller.ts
```
  @Post()
  create(@Body() createArtistDto: Prisma.ArtistCreateInput) {
    return this.artistsService.create(createArtistDto);
  }
```

//artists.module.ts
```
  providers: [ArtistsService, PrismaService],
```

You have to register the `PrismaService` into ArtistModule

Now we need to send the HTTP request to create the artist

Create Artist
POST http://localhost:3000/artists
Content-Type: application/json

```
{
    "name" : "Avicci"
}
```

Step 4: Add Artist's relation during Song Creation
```
export type SongUncheckedCreateInput = {
  id?: number;
  title: string;
  artistId?: number | null;
};
```

We have used that type in the create method which means I can add `artistId` in the request body while creating a new song

Let's test the application by providing `artistId`

Create Song
POST http://localhost:3000/songs
Content-Type: application/json

```
{
    "title" : "Animals",
    "artistId":1
}
```

You can populate the record of the artist while fetching all the songs. You can include the artist in the query like this

```
 findAll() {
    return this.prisma.song.findMany({ include: { artist: true } });
  }
```

Now you will send the request to fetch all songs and you will get the `artist` record with a song

One to One Relation

Step 1: Add Relation

A user will have a unique profile. Each unique profile belongs to a single user. We have added the relationship between the user and the profile

```
model User {
  id   Int   @id @default(autoincrement())
  name String

  profile Profile?
}

model Profile {
  id      Int   @id @default(autoincrement())
  user    User  @relation(fields: [userId], references: [id])
  userId Int    @unique
  photo  String
  phone  String
}
```

- `profile`: This field represents the relationship between the `User` model and the `Profile` model. It uses the `@relation` attribute to define the relationship.

- The `@relation` attribute has two arguments: `fields` and `references`. It specifies that the `profile` field in the `User` model references the `id` field in the `Profile` model.
- The `?` indicates that the `profile` field is optional, meaning a user may or may not have an associated profile
- It uses the `@unique` attribute to ensure that each `Profile` is associated with a unique `User`

Step 2: Run Prisma Migration

```
npx prisma migrate dev --name=one-to-one
```

Step 3: Generate User Resource

```
nest g resource users
```

Step 4: Save User with Profile

```typescript
//users.service.ts
create(createUserDto: CreateUserDto) {
    return this.prisma.user.create({
      data: {
        name: createUserDto.name,
        profile: {
          create: {
            phone: createUserDto.phone,
            photo: createUserDto.photo,
          },
        },
      },
    });
  }
```

Step 5: Find a user with Profile

```typescript
findAll() {
    return this.prisma.user.findMany({ include: { profile: true } });
  }
```

Many to Many Relation

Use Case

- One blog post can belong to many categories
- Many categories can belong to a single blog post

Step 1: Add Many to Many Relation

```
model Post {
  id         Int                  @id @default(autoincrement())
  title      String
  categories CategoriesOnPosts[]
}
```

- categories: This field represents the relationship between the Post model and the CategoriesOnPosts model. It is an array of CategoriesOnPosts, indicating that a post can have multiple categories associated with it.

```
model Category {
  id    Int                  @id @default(autoincrement())
  name  String
  posts CategoriesOnPosts[]
}
```

- posts: This field represents the relationship between the Category model and the CategoriesOnPosts model. It is an array of CategoriesOnPosts, indicating that a category can be associated with multiple posts.

```
model CategoriesOnPosts {
  post       Post     @relation(fields: [postId], references: [id])
  postId     Int
  category   Category @relation(fields: [categoryId], references: [id])
  categoryId Int

  asignedAt  DateTime @default(now())
  assignedBy String

  @@id([postId, categoryId])
}
```

- post: This field represents the relationship between the CategoriesOnPosts model and the Post model. It uses the @relation attribute to define the relationship. The @relation attribute has two arguments: fields and references. It specifies that the post field in the CategoriesOnPosts model references the id field in the Post model.
- postId: This field is of type Int and represents the foreign key that links a CategoriesOnPosts to its corresponding Post.
- category: This field represents the relationship between the CategoriesOnPosts model and the Category model. It uses the @relation attribute to define the relationship. The

@relation attribute has two arguments: `fields` and `references`. It specifies that the `category` field in the `CategoriesOnPosts` model references the `id` field in the `Category` model.

- `categoryId`: This field is of type `Int` and represents the foreign key that links a `CategoriesOnPosts` to its corresponding `Category`.
- `assignedAt`: This field represents the timestamp when the category was assigned to the post. It uses the `DateTime` type and has a default value of `now()`, meaning it will be set to the current date and time when a new record is created.
- `assignedBy`: This field represents the name of the user who assigned the category to the post and is of type `String`.
- `@@id([postId, categoryId])`: This attribute defines a composite primary key using both `postId` and `categoryId` as the primary key fields. This ensures that a combination of post and category uniquely identifies each record in the `CategoriesOnPosts` table, allowing for a many-to-many relationship between posts and categories.

Step 2: Run Migrations

```
npx prisma migrate dev --name=many-to-many
```

Step 3: Create Posts Resource

```
nest g resource posts
```

Step 4: Create Post

```ts
//posts.service.ts
create(createPostDto: Prisma.PostCreateInput) {
    return this.prisma.post.create({ data: createPostDto })
}
```

```ts
//posts.service.ts
create(createPostDto: Prisma.PostCreateInput) {
    return this.prisma.post.create({ data: createPostDto })
}
```

Now we need to send the request to create a post with 2 new categories in the DB

```
### Create POST
POST http://localhost:3000/posts
Content-Type: application/json

{
  "title": "One to Many Relation",
  "categories": {
```

```
        "create": [
          {
            "assignedBy": "Jane",
            "asignedAt": "2023-08-01T10:03:38.016Z",
            "category": {
              "create": {
                "name": "Prisma"
              }
            }
          },
          {
            "assignedBy": "Jane",
            "asignedAt": "2023-08-01T10:03:38.016Z",
            "category": {
              "create": {
                "name": "Nest.js"
              }
            }
          }
        ]
      }
}
```

Step 5: Create a Post by making a relationship with Existing Categories

If you want to create a new post but don't want to create new categories you can make a relation with categories

```
POST http://localhost:3000/posts
Content-Type: application/json

{
  "title": "Transactions in Prisma",
  "categories": {
    "create": [
      {
        "assignedBy": "Bob",
        "assignedAt": "2023-08-01T10:07:00.918Z",
        "category": {
          "connect": {
            "id": 1
          }
        }
      },
```

```json
        {
          "assignedBy": "Bob",
          "asignedAt": "2023-08-01T10:07:00.918Z",
          "category": {
            "connect": {
              "id": 2
            }
          }
        }
      }
    ]
  }
}
```

Step 6: Relation Queries

```typescript
//posts.service.ts
findAll(where: Prisma.PostWhereUniqueInput) {
    return this.prisma.post.findMany({
        where,
    });
}
```

```typescript
//posts.controller.ts
@Get()
 findAll(
    @Body()
    where: Prisma.PostWhereUniqueInput,
 ){
    return this.postsService.findAll(where);
 }
```

Let's run the query to fetch all the posts with the Nest.js category

```
### FETCH ALL THE POSTS WITH NEST.JS CATEGORY
GET http://localhost:3000/posts
Content-Type: application/json

{
  "categories": {
    "some": {
      "category": {
        "name": "Nest.js"
      }
    }
```

```
  }
}
```

Nested Queries

What is Transaction

In Prisma, a transaction is a way to group multiple database operations into a single logical unit of work that must either be fully completed or fully rolled back. Transactions ensure data consistency and integrity by making sure that if any part of the transaction fails, all changes made within that transaction are reverted, and the database remains in a consistent state.

When to use nested writes

Consider using nested writes if:

- You want to create two or more records related by ID at the same time (for example, create a blog post and a user)
- You want to update and create records related by ID at the same time (for example, change a user's name and create a new blog post)

Use Case

- There is a One to one relationship between `Customer` and `Address`
- There is a One to many relationship between `Customer` and `Application`

```
model Customer {
  id           Int           @id @default(autoincrement())
  name         String
  email        String        @unique
  address      Address?      @relation(fields: [addressId], references: [id])
  applications Application[]
  addressId    Int?          @unique
}

model Address {
  id      Int       @id @default(autoincrement())
  zip     String?
  city    String
  country String
  Customer Customer?
```

```
}

enum APPLICATION_TYPE {
  LOAN
  CAR_FINANCING
  BUSINESS_FINANCING
}

model Application {
  id      Int              @id @default(autoincrement())
  type    APPLICATION_TYPE @default(LOAN)
  tenure String
  amount Int

  Customer    Customer? @relation(fields: [customerId], references: [id])
  customerId Int?
}
```

Step 2: Run migrations

```
npx prisma migrate dev --name=nested-queries
```

Step 3: Generate Application resource

```
nest generate resource applications
```

Step 4: Inject Prisma dependency

```
providers: [ApplicationsService, PrismaService],
```

You have to register the PrismaService as a provider in the Application Module

```
constructor(private prisma: PrismaService) {}
```

Step 5: Create the Application

```
create(createApplicationDto: Prisma.CustomerCreateInput) {
    return this.prisma.customer.create({ data: createApplicationDto });
}

@Post()
  create(@Body() createApplicationDto: Prisma.CustomerCreateInput) {
    return this.applicationsService.create(createApplicationDto);
  }
```

Let's create a new application:

```
### CREATE NEW APPLICATION
POST http://localhost:3000/applications
Content-Type: application/json

{
  "email": "jane1@gmail.com",
  "name": "Jone Doe",
  "address": {
    "create": {
      "city": "New York",
      "country": "USA",
      "zip": "34443"
    }
  },
  "applications": {
    "create": [
      {
        "amount": 32224,
        "tenure": "5 Years",
        "type": "BUSINESS_FINANCING"
      }
    ]
  }
}
```

Application is dependent on the `customerId` and `addressId` If any of the records fails to be created then the application will not be created or if any error occurs while creating the application it will roll back the transaction for the customer and address. The records will be deleted in the `customer` and `address` tables

Batch Bulk Operations

If you have to perform batch and bulk operations like `deleteMany`, `createMany`

and `updateMany` then you can use the `$transaction` API

We are going to run multiple queries at the same time, if any of query fails you will not get any results back. It helps when you are creating multiple records at the same time. If one record fails then the complete operation will be discarded

```
return this.prisma.$transaction([
  this.prisma.post.findMany(),
  this.prisma.artist.findMany(),
```

```
    this.prisma.song.findMany(),
    this.prisma.application.findMany(),
]);
```

Interactive Transactions

Use Case:
- You are going to build the account transfer feature in the banking application
- John wants to transfer the amount of $100 to Sam Account

Process / Flow
1. Update the John Account or deduct the $100 from John
2. Check the balance of John account
3. If balance is less than $0 you have to rollback the transaction. The operation should not be continue
4. If any of the step fails the complete operation will be discarded

Step 1: Create Model
```
model Account {
  id      Int     @id @default(autoincrement())
  balance Float
  title   String
}
```

Step 2: Run Migrations and generate resource
```
npx prisma migrate dev --name=add-account

nest g resource accounts
```

Step 3: Create two new accounts
Inject `PrismaService` in AccountsModule

```
providers: [AccountsService, PrismaService],
```

Inject PrismaService as a dependency

```
constructor(private prisma: PrismaService) {}

create(createAccountDto: Prisma.AccountCreateInput) {
```

```
        return this.prisma.account.create({ data: createAccountDto });
    }

    create(@Body() createAccountDto: Prisma.AccountCreateInput) {
        return this.accountsService.create(createAccountDto);
    }
```

Create Account
POST http://localhost:3000/accounts
Content-Type: application/json

```
{
  "title" : "John",
  "balance" : 100
}
```

Create Account
POST http://localhost:3000/accounts
Content-Type: application/json

```
{
  "title" : "Sam",
  "balance" : 100
}
```

You have to create two new accounts to do the testing

Step 4: Create route for transfer account

```
@Post('transfer')
  transfer(@Body() transferAccountDTO: TransferAccountDTO) {
    return this.accountsService.transfer(transferAccountDTO);
  }

export class TransferAccountDTO {
  sender: number;
  receiver: number;
  amount: number;
}

transfer(transferAccountDTO: TransferAccountDTO) {
    throw new Error('Method not implemented.');
  }
```

Step 5: Implement Account Transfer Logic

```
transfer(transferAccountDTO: TransferAccountDTO) {
    const { sender: from, receiver: to, amount } = transferAccountDTO;
    return this.prisma.$transaction(async (tx) => {
      // John Account
      // 1. Decrement amount from the sender.
      const sender = await tx.account.update({
        data: {
          balance: {
            decrement: amount,
          },
        },
        where: {
          id: from,
        },
      });

      // 2. Verify that the sender's balance didn't go below zero.
      if (sender.balance < 0) {
        throw new Error(`${from} doesn't have enough to send ${amount}`);
      }

      // 3. Increment the recipient's balance by amount
      // SAM Account
      const recipient = await tx.account.update({
        data: {
          balance: {
            increment: amount,
          },
        },
        where: {
          id: to,
        },
      });

      return recipient;
    });
  }
```

Test Transfer Process

```
### Trasnfer Amount
POST http://localhost:3000/accounts/transfer
Content-Type: application/json
```

```
{
  "sender" :1,
  "receiver" :2,
  "amount": 50
}
```

Trasnfer Amount
POST http://localhost:3000/accounts/transfer
Content-Type: application/json

```
{
  "sender" :1,
  "receiver" :2,
  "amount": 40
}
```

Trasnfer Amount
POST http://localhost:3000/accounts/transfer
Content-Type: application/json

```
{
  "sender" :1,
  "receiver" :2,
  "amount": 20
}
```

You will get an error at the last HTTP operation because John will not have a balance more than $20 so the transaction will be rollback

Chapter 19 Additional Nestjs Concepts

File Upload

Step 1: Install Multer Types

`npm install @types/multer`

This is a package that provides TypeScript type definitions for the `multer` package, which is a middleware for handling `multipart/form-data` (typically used for uploading files) in Node.js.

When working with TypeScript, it's important to have accurate type definitions for third-party libraries like `multer`. These type definitions help you write more reliable and type-safe code by providing information about the functions, objects, and properties that the library exposes.

Step 2 Create upload route handler

```typescript
//app.controller.ts
import { FileInterceptor } from '@nestjs/platform-express';
import { diskStorage } from 'multer';

import {
  HttpStatus,
  ParseFilePipeBuilder,
  UploadedFile,
  UseInterceptors,
} from '@nestjs/common';

@Post('upload')
@UseInterceptors(FileInterceptor('file'))
uploadFile(@UploadedFile() file: Express.Multer.File) {
  console.log(file);
}
```

- `@UseInterceptors(FileInterceptor('file'))`: This decorator indicates that the method should use the `FileInterceptor` interceptor to handle file uploads.
- The string `'file'` inside the `FileInterceptor` decorator refers to the field name in the request payload that contains the uploaded file
- This interceptor automatically processes the uploaded file and makes it accessible in the `@UploadedFile()` decorator.
- `uploadFile(@UploadedFile() file: Express.Multer.File)`: This parameter decorator retrieves the uploaded file that was processed by the `FileInterceptor`
- The `@UploadedFile()` decorator extracts the uploaded file from the request and assigns it to the file parameter.

Step 3: Upload file with Validations

```typescript
@Post('upload-png')
@UseInterceptors(
  FileInterceptor('file', {
    storage: diskStorage({
      destination: './upload/files',
      filename: (req, file, cb) => {
        cb(null, file.originalname);
      },
    }),
```

```
    }),
  )
  uploadFileWithValidation(
    @UploadedFile(
      new ParseFilePipeBuilder()
        .addFileTypeValidator({
          fileType: 'png',
        })
        // .addMaxSizeValidator({
        //   maxSize: 70706,
        // })
        .build({
          errorHttpStatusCode: HttpStatus.UNPROCESSABLE_ENTITY,
        }),
    )
    file: Express.Multer.File,
  ) {
    console.log(file);
    return {
      messge: 'file uploaded successfully!',
    };
  }
```

- The interceptor is configured with a `diskStorage` option, specifying that the uploaded files will be stored on the local disk. The `destination` option specifies the directory where the uploaded files will be saved, and the `filename` option determines the name of the saved file (in this case, it retains the original name)
- `ParseFilePipeBuilder`: This is a custom utility that creates a pipe for validating uploaded files. In this code, it's used to add a validation check to ensure that the uploaded file is of type PNG

Custom Decorator

What are Custom Dectorators?

Custom decorators in Nest.js are user-defined annotations that allow you to add metadata to classes, methods, properties, or parameters in your application. These decorators are a powerful feature of TypeScript and enable you to extend the functionality of Nest.js by creating reusable and structured code constructs.

Here's why you might want to use custom decorators in Nest.js:

1. **Modularity and Reusability**: Custom decorators enable you to encapsulate specific functionality or behavior into reusable modules. By creating decorators, you can define a set of actions that can be easily applied to different parts of your application, promoting modularity and reducing code duplication.
2. **Code Organization**: Nest.js applications can become complex as they grow. Custom decorators allow you to keep related code together and organize it in a structured manner. This enhances code readability and maintainability.
3. **Cross-Cutting Concerns**: Cross-cutting concerns, such as logging, authentication, authorization, and validation, often need to be applied to multiple parts of your application. Custom decorators provide a clean way to inject these concerns into your codebase without cluttering your business logic.
4. **Aspect-Oriented Programming**: Decorators enable a programming paradigm called aspect-oriented programming (AOP), where you can separate concerns that cross multiple parts of your application. This separation makes your codebase more modular and easier to understand.
5. **Metadata and Reflection**: Decorators provide a way to add metadata to your classes, methods, or properties. This metadata can be introspected and used for various purposes, such as generating documentation, enforcing policies, or performing transformations.
6. **Extend Nest.js Functionality**: Nest.js itself uses decorators extensively to add features like routing, dependency injection, middleware, and more. By creating custom decorators, you can extend the core functionality of Nest.js to suit your specific needs.
7. **Domain-Specific Language**: Decorators allow you to create a domain-specific language (DSL) that expresses the intent of certain operations in a clear and concise way, making your codebase more expressive and easier to understand.
8. **Third-Party Libraries**: You can use custom decorators to integrate with third-party libraries, frameworks, or tools. For example, you could create a decorator that integrates with a logging library or an authentication module.

In summary, custom decorators in Nest.js provide a mechanism to add behavior, metadata, or cross-cutting concerns to your codebase in a modular and reusable way. They enhance code organization, promote best practices, and allow you to extend and customize the behavior of your application beyond what the framework provides out of the box.

Step 1: Create Custom Decorator

```typescript
//user.decorator.ts
import { ExecutionContext, createParamDecorator } from "@nestjs/common";

export const User = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    request.user = { id: 1, name: "Haider Ali" };
    return request.user;
  }
);
```

- `(data: unknown, ctx: ExecutionContext) => { ... }`: This is the decorator factory function. It takes two parameters:
  - `data`: This parameter allows you to pass additional data to the decorator. In your code, you're not using this parameter, so it's of type `unknown`.
  - `ctx: ExecutionContext`: This parameter provides context information about the current request. It's used to access the `request` object.
- `const request = ctx.switchToHttp().getRequest();`: This line retrieves the `request` object from the `ExecutionContext`. The `switchToHttp()` method returns an `HttpArgumentsHost` instance that provides access to the `request` and `response` objects.

Step 2: Create User Entity

```
export class UserEntity {
  id: number;
  name: string;
}
```

Step 3: Apply User Decorator

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

 @Get('/user/:id')
  findOne(
    @User()
    user: UserEntity,
  ) {
    console.log(user);
    return user;
  }
```

Running CRON Task

Step 1: Install Packages

Many times, we need to run an arbitrary piece of code based on some schedule. The schedule can be a fixed time, a recurring interval or after a specific timeout.

Let's install dependencies

```
npm install --save @nestjs/schedule
```

```
npm install --save-dev @types/cron
```

Step 2: Register ScheduleModule in AppModule

```
import { ScheduleModule } from '@nestjs/schedule';


imports: [ScheduleModule.forRoot()],
```

Step 3: Create TaskService and define CRON Job

```
nest g service Task

import { Injectable, Logger } from "@nestjs/common";
import { Cron } from "@nestjs/schedule";

@Injectable()
export class TaskService {
  private readonly logger = new Logger(TaskService.name);

  @Cron("0 * * * * *")
  myCronTask() {
    this.logger.debug("Cron Task Called");
  }
}

* * * * * *
| | | | | |
| | | | | day of week
| | | | months
| | | day of month
| | hours
| minutes
seconds (optional)
```

Examples

```
* * * * * *  every second
45 * * * * * every minute, on the 45th second
0 10 * * * * every hour, at the start of the 10th minute
0 */30 9-17 * * *  every 30 minutes between 9am and 5pm
0 30 11 * * 1-5   Monday to Friday at 11:30am
```

Cookies

What are Cookies?

Cookies are small pieces of data that a server sends to a user's web browser while the user is browsing a website. These cookies are stored on the user's device and are used to store information about the user's interactions with the website. Cookies are an essential part of web technology and play a crucial role in providing a personalized and seamless browsing experience. Here's why cookies are important:

1. **Session Management**: Cookies are often used to manage user sessions. When a user logs in to a website, a session cookie is created, which allows the server to recognize the user and maintain their authenticated state as they navigate different pages on the site.
2. **Personalization**: Cookies can store user preferences and settings, allowing websites to customize the user experience based on their previous interactions. This could include remembering language preferences, display settings, and more.
3. **Shopping Carts and E-Commerce**: E-commerce websites use cookies to maintain shopping carts. Items added to a cart are stored in a cookie, allowing the user to continue shopping and complete their purchase later.
4. **Tracking and Analytics**: Cookies are often used to gather data about user behavior, such as which pages they visit, how long they stay on a page, and what actions they take. This data is then used for analytics and to improve the website's performance and user experience.
5. **Authentication and Security**: Cookies can help with authentication and security by storing tokens or other information that verifies a user's identity. For example, a website might use cookies to remember that a user is logged in, so they don't have to log in again on every page.
6. **Remember Me Functionality**: Many websites offer a "Remember Me" option when logging in. This sets a persistent cookie that allows the user to stay logged in even after closing and reopening their browser.
7. **Tracking User Activity**: Cookies can be used to track a user's activity across different websites. This is often used for targeted advertising and marketing.
8. **Load Balancing**: In some cases, cookies are used to help distribute website traffic across multiple servers, ensuring a more even load distribution.
9. **State Management**: Cookies are often used to manage state information in web applications. For example, they can be used to remember where a user was in a multi-step process.
10. **Caching**: Cookies can be used to cache certain information on the user's device, reducing the need to repeatedly request the same data from the server.

While cookies offer many benefits, it's important to note that there are also concerns related to privacy and security. Some users may disable or delete cookies to protect their privacy, and there

are regulations in place, such as the General Data Protection Regulation (GDPR), that govern how cookies and user data can be used.

In recent years, there has been an increased focus on alternative solutions like server-side sessions, local storage, and newer web technologies like Web Storage API and IndexedDB. These options provide alternatives to traditional cookies for storing data on the user's device.

Step 1: Install Packages

```
npm install cookie-parser
npm install -D @types/cookie-parser
```

Step 2: Register Cookie Parser

```typescript
//main.ts
import * as cookieParser from "cookie-parser";


app.use(cookieParser());
```

Step 3: Set Cookies

```typescript
//app.controller.ts
import {
 Req,
  Res,
} from '@nestjs/common';
import { Request, Response } from 'express';


@Get('set-cookie')
  setCookie(
    @Res({ passthrough: true })
    response: Response,
  ) {
    response.cookie('Cookie token Name', 'encrypted cookie string');
    response.send('Cookie Saved Successfully');
  }

@Get('get-cookie')
  finndAll(@Req() req: Request) {
    console.log(req.cookies);
    return req.cookies;
  }
```

Step 4: Test

When you run the application you have to send the setCookies request first from your browser. The cookie will be saved in your Google Chrome Browser. You have to open chrome dev tools and find the Application tab. You can find your cookies there.

Queues

What are Queues in Nest.js?

In Nest.js, queues refer to the concept of handling tasks asynchronously using a queuing system. Queues are a way to manage and process tasks in the background without blocking the main application's execution. This is particularly useful for tasks that are time-consuming, resource-intensive, or don't need to be executed immediately as part of a request-response cycle.

A common use case for queues is to process tasks like sending emails, generating reports, processing images, or any other task that can be offloaded from the main application flow.

Here's a high-level overview of how queues work in Nest.js:

1. **Task Generation:** In your application, you identify tasks that can be processed asynchronously. For example, let's consider sending an email after a user registers. Instead of sending the email directly within the registration endpoint, you can push the task of sending the email to a queue.
2. **Queue Library Integration:** You integrate a queue library of your choice (e.g., Bull) into your Nest.js application. This involves installing the library, configuring it, and creating a queue instance.
3. **Enqueuing Tasks:** When you want to perform a task asynchronously, you enqueue it in the queue. For our example, after a user registers, you enqueue a task to send the registration email.
4. **Worker Process:** You create one or more worker processes that continuously monitor the queue for tasks. When a task is available in the queue, a worker picks it up and processes it. In our example, the worker would send the registration email.
5. **Background Execution:** The task is executed in the background, separate from the main application thread. This ensures that the main application remains responsive and doesn't get blocked by time-consuming tasks.

Step 1: Install Dependencies

```
npm install @nestjs/bull
npm install bull
```

`@nestjs/bull` is a Nest.js module that provides integration with the Bull queue library, which is built on top of Redis. In other words, when you use `@nestjs/bull` for implementing queues in your Nest.js application, you are actually using Redis under the hood.

Step 2: Creating Audio Module

```
nest g module audio && nest g controller audio
```

Step 3: Setup Redis with docker-compose

You have to create a new file with `docker-compose.yml` in the root directory. Make sure you have installed docker on your machine

```yaml
version: "3"
services:
  redis:
    image: redis:alpine
    ports:
      - 6379:6379
```

Let's start the redis service by opening the terminal and run `docker-compose up`

Step 4: Register BullModule

```typescript
//app.module.ts
imports: [
  BullModule.forRoot({
    redis: {
      host: "localhost",
      port: 6379,
    },
  }),
];
```

You have to register the `BullModule` in the `AppModule`

Step 5: Register BullModule Queue

```typescript
//audio.module.ts
imports: [
    BullModule.registerQueue({
      name: 'audio-queue',
    }),
  ],
```

1. `registerQueue({...})`: This method is used to register a queue within your application. It takes an options object as an argument to configure the queue.
2. `name: 'audio-queue'`: This is the `name` property within the options object. It specifies the name of the queue you want to create. In this case, the queue will be named "audio-queue".

Step 6: Creating audio convertor endpoint

```typescript
import { InjectQueue } from "@nestjs/bull";
import { Controller, Post } from "@nestjs/common";
import { Queue } from "bull";

@Controller("audio")
export class AudioController {
  constructor(
    @InjectQueue("audio-queue")
    private readonly audioQueue: Queue
  ) {}

  /**
   * Let's imagine we would like to convert .wav file into .mp3
   */
  @Post("convert")
  async convert() {
    await this.audioQueue.add("convert", {
      file: "sample.wav",
    });
  }
}
```

Step 7: Implement Audio Processor

```typescript
import { Process, Processor } from "@nestjs/bull";
import { Logger } from "@nestjs/common";
import { Job } from "bull";

@Processor("audio-queue")
export class AudioProcessor {
  private logger = new Logger(AudioProcessor.name);

  @Process("convert")
  handleConvert(job: Job) {
    this.logger.debug("start converting wav file to mp3");
    this.logger.debug(job.data);
    this.logger.debug("file converted successfully");
  }
```

```
}
```

- `@Processor('audio-queue')`: This decorator specifies that this class is a processor for the "audio-queue". This means that it will handle tasks enqueued in the "audio-queue".
- `@Process('convert')`: This decorator specifies that the `handleConvert` method should handle tasks with the name "convert". When a task with the name "convert" is enqueued in the "audio-queue", the `handleConvert` method will be triggered to process it.
- `handleConvert(job: Job)`: This is the method that processes tasks with the name "convert". It takes a `Job` object as its parameter, which contains information about the task and its data

Event Emitter

What is an Event Emitter?

In Nest.js, an event emitter is a mechanism that allows different parts of your application to communicate with each other using an event-driven approach. It's a way to facilitate communication and coordination between different modules, services, components, or classes within your application.

An event emitter works on the principle of publishers and subscribers. The entity that generates an event is called the "publisher," and the entity that listens and responds to the event is called the "subscriber."

Why do we need it?

Practical Use Cases of Event Emitters in Nest.js:

1. **Module Communication**: Modules in a Nest.js application are often designed to be independent and self-contained. However, there are scenarios where modules need to communicate with each other. Event emitters provide a way for one module to emit an event and for other modules to react to that event.
2. **Service Interaction**: Different services within your application might need to communicate or coordinate their actions. For example, when a user performs an action in one service, it might trigger actions in another service. Event emitters can facilitate this communication without creating direct dependencies between services.
3. **Notification Systems**: You can use event emitters to implement a notification system. When an important event occurs, you can emit an event, and subscribers (such as notification services) can react by sending notifications to users or other parts of the system.
4. **Real-time Updates**: In real-time applications, you can use event emitters to send updates to connected clients. For example, in a chat application, when a new message is received, the server can emit an event, and all connected clients receive the update instantly.

5. **Plugin System**: If your application supports plugins or extensions, event emitters can allow plugins to listen for specific events and modify the behavior of the core application accordingly.
6. **Logging and Monitoring**: You can use event emitters to notify a logging or monitoring system about significant events within your application. This can help track and analyze the application's behavior.
7. **Workflow Orchestration**: For complex workflows involving multiple steps or processes, event emitters can be used to trigger the next step once the previous step is completed.
8. **Error Handling and Reporting**: When an error occurs, you can emit an event that notifies an error handling service. This service can then log the error, notify administrators, or take other appropriate actions.
9. **User Authentication and Authorization**: Event emitters can be used to handle user authentication and authorization events. For example, an authentication service could emit an event when a user successfully logs in, and other parts of the application can respond accordingly.
10. **Caching and Data Management**: In a cache management system, event emitters can be used to notify the cache to update or clear cached data when relevant changes occur in the application.

Overall, event emitters in Nest.js facilitate loose coupling between different parts of your application, enabling better modularity, scalability, and maintainability. They help achieve separation of concerns and allow different parts of the application to interact without needing to know the details of each other's implementation.

Use Case

- We are going to take an example from our previous example let's say we want to send the notification to the user when the `.wav` file converts successfully into `.mp3` format.
- You can use EventEmitter to send the notification to the user

Step 1: Install Dependencies

```
npm install @nestjs/event-emitter
```

Step 2: Register EventEmitter Module

```
//app.module.ts
  EventEmitterModule.forRoot(),
```

Step 3: Create AudioConvertedListener

```typescript
import { Injectable } from '@nestjs/common';
import { OnEvent } from '@nestjs/event-emitter';
import { AudioConvertedEvent } from './events/audio-converted-event';


@Injectable()
```

```
export class AudioConvertedListener {
  @OnEvent('audio.converted') // We have registered a new event lister with
audio.converted name
    handleAudioConvertedEvent(event: AudioConvertedEvent) { //We have to create the
type for the AudioConvertedEvent
      console.log(event);
      // Here you can have your EmailService method you can call here
      console.log(
        'Notification has been sent to user that file is converted successfully,
      );
    }
}
```

Step 4: Create AudioConvertedEvent Type

```
export class AudioConvertedEvent {
  file: string;
  id: number;
}
```

You have to created this class in your `audio` folder.

Make sure you have registered it in your provider

```
providers: [AudioProcessor, AudioConvertedListener],
```

Step 5: Emit the Event in AudioProcessor

```
constructor(private eventEmitter: EventEmitter2) {}
```

Make sure you have injected the EventEmitter dependency in your `AudioProcessor` class

```
handleConvert(job: Job){
 //...
 this.eventEmitter.emit('audio.converted', job.data);
}
```

We have to emit the event in the `handleConvert` method.

Step 6: Run the Application

Now you have to run the application and send the audio convert request from `http-client.http`

You will see the message `Notification has been sent to the user that file is converted successfully`. It will also log the event details

Streaming

What is Streaming

Streaming in Nest.js refers to the process of sending or receiving data in small chunks, called "streams," rather than sending or receiving the entire data at once. This concept is based on the Stream API in Node.js and is utilized for more efficient data handling, especially when dealing with large amounts of data, such as files, network requests, or real-time data transmission.

Practical Use Cases of Streaming in Nest.js:

1. **File Uploads and Downloads**: When uploading or downloading large files, streaming allows you to process the data in chunks, reducing memory consumption and improving performance. This is particularly useful for handling large media files, backups, or logs.
2. **Real-time Communication**: Streaming is essential for real-time communication technologies like WebSockets. Streaming data in real-time ensures that clients receive updates as they happen, which is critical for applications like chat applications, live feeds, or online gaming.
3. **Media Streaming**: Applications that involve streaming audio or video content, such as music or video platforms, benefit from the ability to stream data to users' devices progressively. This allows users to start consuming the media before the entire file is downloaded.
4. **API Responses**: Streaming can be used to send large responses from APIs, like lists of items, without waiting for the entire response to be generated. This can improve the API's responsiveness and user experience.
5. **Data Transformation**: Streaming is used when processing data transformations or conversions, such as reading data from one format, transforming it, and writing it into another format. This can be used in ETL (Extract, Transform, Load) processes or data pipelines.
6. **Reading from Streams**: Reading from a stream is useful when processing data from sources that generate data incrementally, such as reading log files or parsing large XML or JSON documents.
7. **Data Aggregation**: When dealing with data aggregation or analytics, streaming can be used to process data in smaller chunks, allowing for real-time analysis or reducing memory usage.
8. **Server-Sent Events (SSE)**: SSE is a technology that enables a server to push real-time updates to a web browser over a single HTTP connection. It uses streaming to send a continuous stream of events to the client.
9. **Batch Processing**: In scenarios where data is collected in batches, streaming can be used to process and handle each batch efficiently.
10. **Database Operations**: When reading or writing large volumes of data to databases, streaming can help optimize data insertion or extraction processes.

11. **Proxy Servers**: Streaming can be used in proxy servers to forward data from one server to another without holding the entire data in memory.
12. **Data Transmission Optimization**: Streaming can help optimize data transmission in scenarios with limited bandwidth, ensuring that data is transferred in manageable chunks.

In summary, streaming in Nest.js is a versatile technique used for efficient data handling, especially when dealing with large volumes of data or real-time communication. It enhances application performance, reduces memory consumption, and improves user experience by allowing data to be processed incrementally as it's received or sent.

Step 1: Create a FileController

```
nest g controller file
```

Step 2: Download the file

```typescript
import { Controller, Get, Header, Res, StreamableFile } from '@nestjs/common';
import { Response } from 'express';
import { createReadStream } from 'fs';
import { join } from 'path';

//Download the file
@Get('stream-file')
  getFile1(): StreamableFile {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    return new StreamableFile(file);
  }

@Get('stream-file-customize')
  getFileCustomizedResponse(@Res({ passthrough: true }) res): StreamableFile {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    res.set({
      'Content-Type': 'application/json',
      'Content-Disposition': 'attachment; filename="package.json',
    });
    return new StreamableFile(file);
  }
```

Session

What is a session?

In Nest.js, sessions refer to the concept of maintaining stateful data between consecutive requests from the same client. A session allows you to store and retrieve user-specific information on the server across multiple HTTP requests, typically using cookies to identify the session.

Practical Uses of Session

1. **User Authentication**: Sessions are commonly used for maintaining the authentication state of a user. When a user logs in, a session can be created with their authentication data, and subsequent requests can be authenticated based on the session.
2. **Authorization**: Sessions can store authorization-related information, such as user roles and permissions, to determine what actions the user is allowed to perform.
3. **User Preferences**: You can use sessions to store user-specific preferences or settings, such as language preference or display settings.
4. **Shopping Carts**: E-commerce applications often use sessions to store the contents of a user's shopping cart as they navigate through the site.
5. **User Tracking**: Sessions can be used to track user behavior and interactions on a website, helping in analyzing user engagement and improving user experience.
6. **Caching**: Sessions can store frequently accessed data, reducing the need to fetch the same data from the database on every request.
7. **Personalization**: Websites can customize content based on user behavior stored in sessions, enhancing user experience.
8. **Form Data Persistence**: Sessions can temporarily store form data between requests, which is useful in multi-step processes.

Step 1: Install Dependencies

```
npm install express-session
npm install -D @types/express-session
```

Step 2 Register Middleware

```
app.use(
  session({
    secret: "my-secret",
    resave: false,
    saveUninitialized: false,
  })
);
```

Step 3: Create Login Route Handler

```
@Get('login')
  loginUser(@Session() session: Record<string, any>) {
```

```
    session.user = { id: 1, username: 'Jane' };
    return 'LoggedIn';
  }
```

Step 4: Profile Route

```
@Get('profile')
  profile(@Session() session: Record<string, any>) {
    const user = session.user;
    if (user) {
      return `Hello, ${user.username}`;
    } else {
      return 'Not logged in';
    }
  }
```

Take Your NestJS Skills to the Next Level!

Thank you for reading through this ebook! I hope you found it helpful in understanding the fundamentals of NestJS. If you're ready to dive deeper and master **NestJS Microservices**, I invite you to join my comprehensive course that covers everything you need to know to build scalable and efficient microservices using NestJS.

Click here to learn more about it