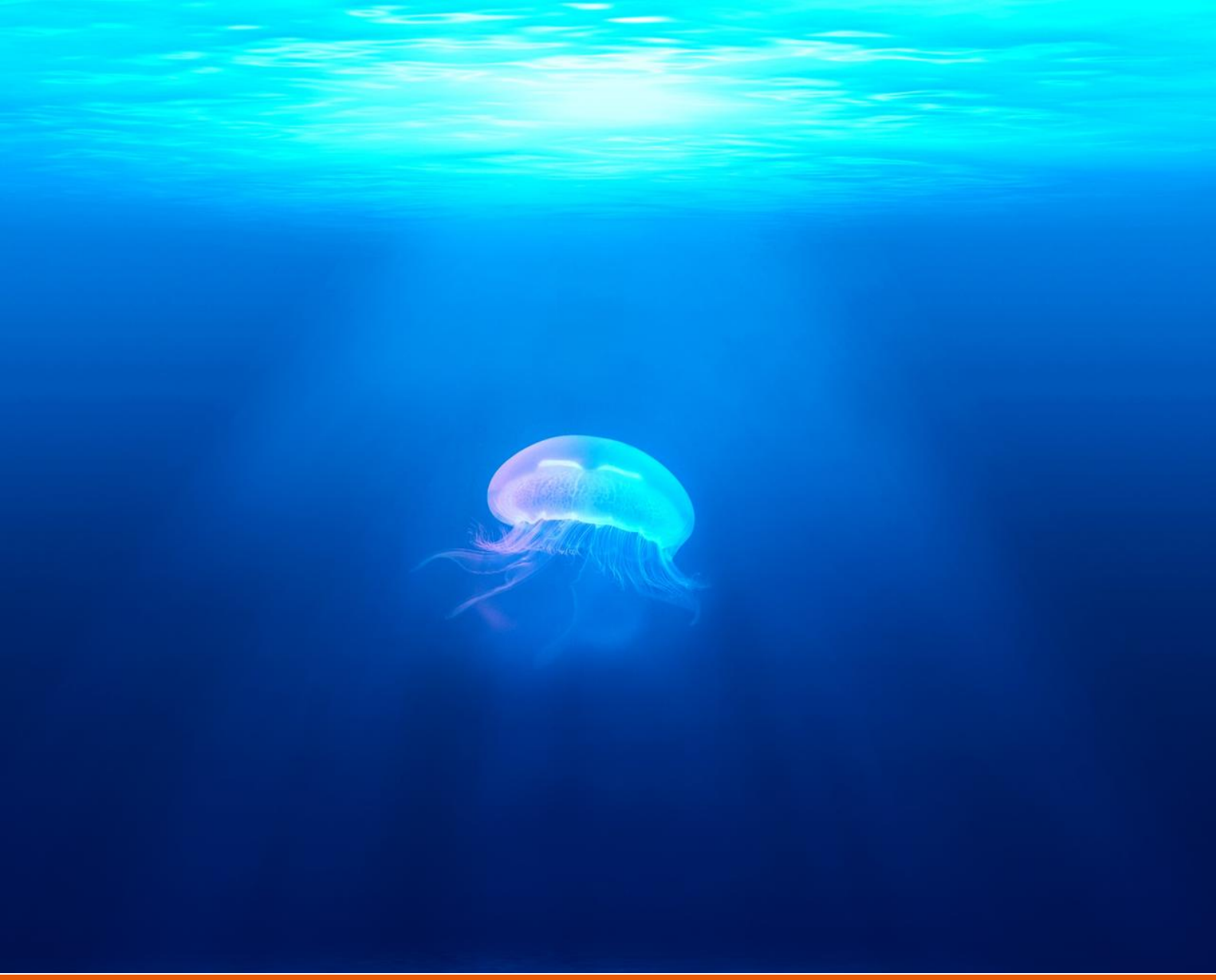
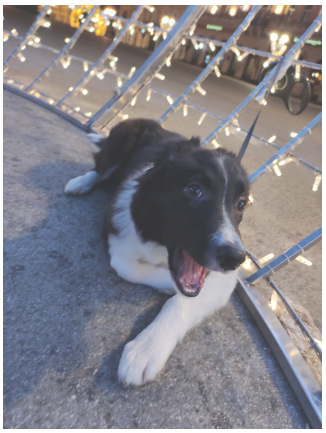


Pragmatic Node.js development

Primer in Nest.js



Author: Zlatibor Zed Veljkovic

Date: June 5, 2022

Version: 0.1

Email: zveljkovic@hotmail.com

I had more time so it is short

Contents

Chapter 1 Developer tools

Since ancient times, mankind has constantly spent effort to create new or to improve existing tools. Even now, after thousands and thousands of years we are doing the same. We are making new tools that will make us more efficient or at least to do our tasks easier. In software development there are so many tools available it is hard to choose which set should be used. Next sections will give simple overview of most prominent tools for each section.

1.1 Command line interfaces

Command line interfaces (CLI) are programs that use console/textual interface and allow you to interact with it. Every operating system comes with one or more of these, Windows has Command Prompt (aka cmd.exe) and PowerShell, Linux has sh and bash with many alternatives (commonly known as shell), and macOS has Terminal.app. One issue that novice developers struggle is that when someone tells you to "open the terminal", they mean one for your system.

Before mentioned apps are the ones that allow you to execute some commands or run different programs. There are also some CLI that is specifically built for one purpose. One example would be Nest.js CLI which allows you to quickly create new projects, update dependencies or start the Nest app.

1.1.1 Command prompt

Command prompt comes preinstalled on Windows systems. It supports batch scripts and usually the file is with .bat extension.

Pros	Cons
• Available on all Windows systems • Allows executing programs in current directory without `.\` prefix	• Batch scripting language is really outdated and hard to write more complex stuff • No command history search

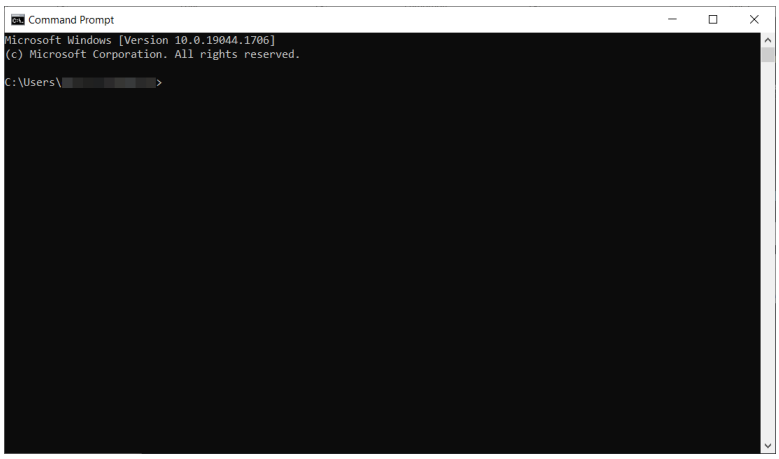


Figure 1.1: Command Prompt

Reference for available Command Prompt commands can be found at Windows Commands

1.1.2 PowerShell

Another shell for Windows systems is called PowerShell, and it is available from Windows 7 or later operating systems. Open source version PowerShell Core was released in 2016, and it is based on .Net Core which also made it cross-platform. It has better integration with various functionalities available in Windows, so it is preferred choice to Command Prompt when working with system administration. For developer work it might be an overkill.

Pros

- Available on all Windows systems
- Better integration with Windows functionalities
- Has command history search (with F8 key)

Cons

- Does not allow executing programs in current directory without `.\` prefix

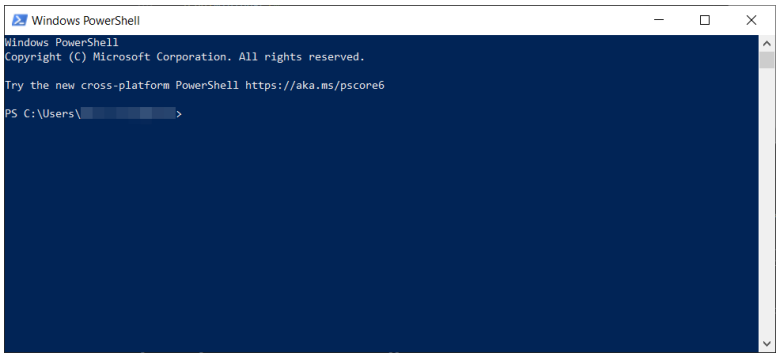


Figure 1.2: PowerShell

1.1.3 Linux/Mac terminals/shells

Linux and Mac have much bigger choice of terminals. Terminal refers to a program that allows you to run programs which are known as shells. Shells come in lots of varieties sh, bash, ksh, csh, zsh. . . . Linux has many command line programs that allow you to manipulate output of commands and offers a lot for power users.

1.2 Package managers

Package managers can be used to install additional software on your PC. They usually automate process of downloading, installing and configuring software. Later it also helps with keeping the installed software up to date or with removal.

- Windows: Chocolatey is the most prominent package manager. It can be downloaded from https://chocolatey.org/. Searching for packages is done with `choco search postgresql` and installation with `choco install postgresql` will install Postgres.
- Windows: Winget is the alternative package manager created by Microsoft. It can be downloaded from https://chocolatey.org/. Searching for packages is done with `winget search postgres` and installation with `winget install postgresql` will install Postgres.
- Mac OS X: Homebrew is a package manager written in Ruby. It can be downloaded from https://brew.sh. Command `brew search postgres` is used to search for package while `brew install postgresql` to install a package.
- Linux has many package managers and preferred way is to use package manager that comes with operating system.

Chapter 2 NestJS application recommendations

2.1 NestJS

I couldn't explain it better than the words of the author "Nest (NestJS) is a framework for building efficient, scalable Node.js server-side applications. It uses progressive JavaScript, is built with and fully supports TypeScript (yet still enables developers to code in pure JavaScript) and combines elements of OOP (Object-Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming)."

The documentation is found on https://docs.nestjs.com/ where you can read all about framework components. Some components are not up to task and in further sections we will focus on good and bad sides of available components and some alternatives.

2.2 Environments

Modern apps usually run in multiple environments. My recommendation for environments is with code name in parenthesis.

- Local development environment (***a***)
- Local testing environment (***b***)
- Public development environment (***d***)
- Public stable development environment (***s***)
- CI testing environment (***t***)
- Public quality assurance environment (***q***)
- Public production environment (***p***)

2.2.1 Local development environment

Local development environment is pretty obvious as each developer is running the application on development machine. Code name to be used for this environment is ***a*** which as a letter represents first letter in alphabet so this environment represents first step in the application development.

2.2.2 Local testing environment

Local testing environment (***b***) is where developer runs tests on development machine. This environment is necessary to isolate testing of the app completely from local development environment. Without this isolation data that has been entered during development could influence test running. We usually use same variables as ***a*** env, but we usually want to change external dependencies like database/queues/Kafka/storage so test runs are properly isolated.

2.2.3 Public development environment

Public development environment is the environment which contains the latest application code. It is configured to be automatically built from the ***develop*** branch. This environment is meant to be broken/reset at any time. It is also first public environment for use mostly by developers.

2.2.4 Public stable development environment

Public stable development environment ***s*** is the environment that is manually updated. When team is happy with the state of ***d*** environment they can promote it to this environment. This allows team leads to test the app without interruptions that would come if they used ***d*** environment.

2.2.5 CI testing environment

CI testing environment is used during the PR verifications in the build pipelines. This is necessary for same reasons as local testing environment, just on the public side.

2.2.6 Public quality assurance environment

Public quality assurance environment ***q*** is environment meant for external evaluation. When all issues identified in ***s*** environment are fixed and team lead is happy with its state, this build can be promoted to ***q*** environment. This environment should be same in specs as public production environment, so it can be also used as database migration verification or as performance test target.

2.2.7 Public production environment

Public production environment ***p*** is environment that is available to the users of the application. This makes it last step in build - release cycle. As real users are using it this environment should be under constant monitoring.

2.3 Environment zones

All of the environments can be grouped in production and non-production zones. The app code can then provide developers with:

- More information about errors for example by outputting stack traces, displaying error data, and various other info which should be hidden from the real users.
- Improve testability by including features to log in as certain type of users, commands that will precreate certain data sets, etc. . .
- Distinguish between seeding production data and developer data.

2.4 Application Configuration

Application configuration in modern development is inescapable. NestJS offers a component ***ConfigService*** which is using ***dotenv*** package internally. What we get is a way to load .env environment file and access the variables defined in environment.

What we are missing is:

- Checking that all required variables are supplied
- That variables conform to the expected type (configuration schema)
- Ability to use multiple .env files so that you can easily configure local development environment (***a***) and local testing environment (***b***) for example by having ***.env.b*** file overriding database name from ***.env*** used by local development environment.
- Type safety as we need to use string identifiers when getting the values

Package convict has almost all of the above but has a requirement that each setting has default value which will not fail the startup if required variable is not found in environment variables.

2.4.1 Zeddy Config

Zeddy Config is my library heavily inspired by convict but offers type safety without default values and is offering just enough functionality for any app. This package also provides variable loading from the ***.env*** and ***.env.env-name*** files with the ***dotenvize*** method. With this simple package we cover all missing elements for good config:

- Single method load the environment variables into JS variable.
- Validation of the configuration schema.
- Full type safety as schema type property is transferred to config object properties.

- It also has additional method to load multiple .env files (***dotenvize***).

```
import { configz, dotenvize } from "zeddy-config";

dotenvize();

export const config = configz({
  env: {
    description: "Running environment",
    envVar: "NODE_ENV",
    type: "string"
  },
  server: {
    description: "Server info",
    type: "object",
    properties: {
      port: {
        description: "Server port",
        envVar: "SERVER_PORT",
        type: "int",
        validator: (port) => {
          return port === 3000;
        }
      },
      host: {
        description: "Server host",
        envVar: "SERVER_HOST",
        type: "string"
      },
    }
  }
});

// resulting config interface is {env: string; server: {port: number; host: string}}
}
```

2.5 Exceptions and errors

Usually people use one or the other term to represent both errors and exceptions but these terms, while both having the same root of something unexpected occurred, have a difference that exceptions can be handled, while error shouldn't be handled.

For example, if we try to open a file which does not exist we should get an FileNotFoundException exception. Normal application will handle that exception to inform the user about missing file and continue working normally.

Errors on the other way prevent the program from running or even compiling. One of the frequent errors is SyntaxError which happens when we don't follow language constructs such as using undeclared variable or misspelling the keywords. This type of error is also called compile time error as it is thrown when compiling the application. Other type of errors are runtime errors which happen during the application run. Usually these errors would lead to app ending or restarting. For example if we do not have enough memory on the system we will get OutOfMemoryError, and best the app can do is to end/restart.

2.5.1 REST Api Errors/Exceptions and HTTP Status Codes

HTTP Status Codes are defined in RFC 9110 which groups the statuses codes (100-599) in five categories:

- 100–199 are informational codes (101 Switching Protocols)
- 200–299 are success codes (200 OK)
- 300–399 are redirect codes (301 Moved Permanently)
- 400–499 are client error code (404 Not Found)
- 500–599 are server error code (500 Internal Server Error)

REST APIs are able to return the result of the action or error/exception and most frameworks like NestJS are tying the errors with server codes. One example would be if we search the database by id and don't find the object we can throw NestJS NotFoundException which will use 404 http status code.

2.5.2 Recommendations for HTTP call errors

My recommendation is to use only two HTTP Status Codes:

- 400 for all exceptions where client retries would not change the outcome. This should represent that user has sent wrong data or not allowed to do something. If something doesn't change this request would never succeed.
- 500 for all errors where client retries could succeed after some timeout. This might be thrown when cache is refreshing, or there is loss of network connectivity to some other services.

When exception is being returned to the HTTP client, in addition to HTTP Status Code, we also need to return more data about the error in response body. These response bodies should be more or less detailed depending on the environment zone the app is running. In ***prod zone*** we should output only exception id and generic name of the error, while in ***dev zone*** we can additionally send more data or stack trace.

Also, JavaScript and NestJS error implementation provide short description of the exception/error, with ***message*** property - which is not recommended. Having many exceptions/errors with unique name is better for responding and providing proper error description with the translation tooling. Having errors translated on the client (frontend) is much preferable as the clients/frontends are translated more often than backends. We have frequently seen multilingual content on websites but rarely (if ever) translated versions of endpoint URLs or POST data names. Additional point is that if we have multiple clients (web and mobile app) each of those can define their own user presentation of that error which is not possible with fixed description returning from backend.

Fine-grained exception/errors are preferred and naming should reflect those, and build from something larger to something smaller. For example ***AuthorizationTokenBearerMissingException***, ***AuthorizationTokenExpiredException***, ... If possible we should group certain exceptions into one by providing additional parameters as we have entity field in ***EntityNotFoundException(*{*entity: User, id: '1234'*}*)*** and ***EntityNotFoundException(*{*entity: UserProfile, id: '1234'*}*)***.

Exceptions/errors should be easy to make, and they should be grouped together with relevant errors instead of all in one place or each for it self.

2.5.3 Zeddy Errors

Zeddy Errors is npm package https://www.npmjs.com/package/zeddy-errors that follows recommendations from REST Api Errors/Exceptions and HTTP Status Codes. It features most simple creation of new errors, allows groups of errors, ease of import, type safety and ease of use. It has 0 dependencies except TypeScript and integrates well into any existing frameworks. There is a sample NestJS project available at (https://github.com/zveljkovic/zeddy-errors-example).

Usage is simple, export error group with defined errors with optional generic argument about the data type, import it where needed and throw the new instance.

```
// file: shared.errors.ts
export const SharedErrors = {
  EntityNotFound: class extends ExceptionBase<{ type: string; id: number }> {},
  JwtTokenMissingBearer: class extends ExceptionBase{},
```

```
  IntentionalError: class extends ErrorBase<{ reason: string }> {},
};

// file: some.service.ts
import {SharedErrors} from './shared.errors';
//...
throw new SharedErrors.IntentionalError({reason: "Testing errors"});
throw new SharedErrors.JwtTokenMissingBearer();
```

2.6 Logging

Logging is one of the most basic but very important aspect of application development. Logs provide the insight into application processing, helps you reproduce the bugs, and can be used as visible comments.

Modern logging should be:

- Effortless. The easier the logs are to write the more developers will use them.
- Centrally configured. Logging configuration should live in one file, so it is easy to modify and extend.
- Optimized for different environments. For example on local environments we want logs to be in plain format printed to standard output, so developers can read them, but for remote environments logs can be in json format for shipping to ELK stack for example.
- Adaptable. Logger should be easy to adapt to specific services. For example integrating with Azure's App Insights.
- Versatile. It's not rare that logs need to be sent to multiple services. Sending logs to ELK stack for long-lasting storage and to Azure AppInsights is common practice.

Chapter 3 NestJS component guide

To better understand the available architectural guides in NestJS we need to shortly introduce ourselves with core building blocks of NestJS.

3.0.1 NestJS Modules

NestJS has is able to modularize the code with Module classes. Each application must have at least one module but can have many modules. These modules can contain multiple classes, and they have an interface with which we can specify what is available from the modules importing it. This allows us to have separation of concerns, by creating self-sufficient modules with public interfaces (by specifying public classes in exports field of module). For example, we can create UserModule that will provide UserService class to any module importing UserModule. Parent modules will be able to get the UserService object via dependency injection system, but they couldn't get the "private" classes used within module like UserRepository, UserEmailService, UserController and others unless explicitly exported from the module. This allows to hide internal implementation of the module and force usage of exported classes to interact with this module.

3.0.2 NestJS Providers

NestJS concept of provider is very large but can be sumarized as a class that can be injected through NestJS dependency injection system. Many of the NestJS basic classes are providers like services, repositories, helpers, controllers, etc. . . Class can be treated as a provider if it has @Injectable annotation and is defined in modules provider field.

3.0.3 NestJS Controllers

NestJS has built-in support for Controllers. They are objects that define endpoints that allow the user to interact with the application. For each endpoint controllers specify which URL, and HTTP method are required to trigger it. Also, controllers serve as a place to define expected request parameters, validation, authorization, and most importantly to trigger the specific code to handle that request.

3.0.4 NestJS Middleware, Exception Filters, Pipes, Guards, Interceptors

The rest of NestJS components are essentially a way to interact with request before it reaches controller. Short explanation of each follow but for more in-depth explanation please read official documentation.

- Middleware - code that runs before the routing.
- Exception filters - code that will allow you to react to unhandled exception from your code.
- Pipes - used for transformation and validation of data before execution of controller.
- Guards - checking if request is allowed to be executed by controller (authentication and authorization).
- Interceptors - wrap controller execution so you can write code that is executed before or after controller execution

Chapter 4 NestJS architectural guides

Depending on the expected size and complexity, application architecture should be designed to maximise productivity, and minimise complexity but without sacrificing the maintainability or development of possible future requirements. There are various online topics on subject of architecture and their applicability, so we will shortly focus on most common types of errors during architecture design.

Over-engineering: this happens when solution is unnecessarily complicated. There are many examples of this but most common ones are:

- designing for multiple databases in agnostic way when 99% of projects never switch the database type (i.e. going from Postgresql to Sql Server).
- using microservice architecture where hidden cost of included complexity on both developer and devops side is not properly compared to the benefits of this approach.
- designing for infinite customisation of the components.

Under-engineering (Hacky): this happens when solution is not using parts for intended purposes but is trying to repurpose existing ones in different way (which often fails). Some of these are:

- using database instead of in memory cache.
- using self-made code for distributed task scheduling (please use ZooKeeper).
- using cron jobs instead of proper background task processing.

This section will focus on most commonly used architectural practices for building NestJS based backend applications in increasing complexity.

4.1 As Simple As It Gets

The most basic setup we can have is to have one module that has one controller defined. All the code gets written in the controller method. This is only suitable for quick POC maybe when testing new libraries and later thrown away. But even then it is a questionable practice as it might be easier to test it in main project.

4.2 One module project based with services based on db entities

One module project will have no modularization but can have multiple controllers, services and other NestJS components. Modularization is emulated with the folder structure and this works really well for POC or small projects. Mainly for smaller projects there is 1:1 correlation between controllers and services. Downside of this approach is that availability of each app component may tempt usage of classes that should be private if there was proper modularization. One example would be to update user directly via UserRepository instead of going through the UserService as it requires all attributes to be specified. Later, for example if we wanted to send a confirmation email everytime user is updated we would have to find all places where UserRepository was used and add code there.

4.3 Multi module project with services based on db entities

An expansion on previous architecture is that now we are using multiple modules properly (by forbidding imports of classes that are not exported from modules), but we still have groupings around entities. This way we have multiple modules, well-defined public interfaces but if application grows in complexity we may start encountering cyclic dependencies. Cyclic dependencies mostly happen in services, and reason is that two service require each other to do the job. For simplistic example we can say that UserService and WorkService have a cyclic dependency when WorkService needs UserService to fetch the user, while UserService in deleteUser method needs WorkService to remove all the Work entities for current user.

4.4 Multi module project with services based on Domain Driven Design

To solve the previous issue we can try the DDD and model services around business domains, and that might help a bit, but when domain interactions become more complex we might still experience the cyclic dependency issue.

4.5 Multi module project with services based on Domain Driven Design plus events

To fight cyclic dependencies arising from large service classes, and make them independent of other modules we might start to utilize internal events. By raising events we may handle some cases in parts of other modules. For example, in UserService we can use exported class UserEvents to raise UserCreated event, and then the EmailService can respond to that event without needing a DI link to UserService. This will help greatly in decoupling the modules and making app more maintainable for larger projects.

Dependency Injection issues

It becomes obvious at this point that dependencies should be scoped to the point of the usage. Having large service classes that depend on the other big service classes, we will come to the point where we will have to fight with cyclic dependencies. But it's not only that, why do we need to instantiate a service if it is used in only one method. We need a new (old) approach.

4.6 Multi module project with CQRS

There is another approach to fight large dependencies - CQRS. CQRS stands for command query responsibility segregation and this pattern makes distinction (responsibility segregation) between commands and queries. Commands are used to mutate data (i.e. creates/updates) while queries are used where data is not updated (i.e. retrieving objects). This Nest.JS concept uses CommandBus/QueryBus pattern to run the commands/queries, but doesn't enforce any of the rules. Queries executed this way can still mutate data so framework still expect the developer to follow the guidelines.

This pattern helps with scoping dependencies to single actions (command or query) as they are separate from the rest of the actions even for the same resource. Sample calling of the command and query is listed below, please see whole example project at https://github.com/zveljkovic/book-cqrs-example.

```
import { Body, Controller, Get, Post } from '@nestjs/common';
import { CommandBus, QueryBus } from '@nestjs/cqrs';
import { GetAllUsersQuery } from '../commands/get-all-users-query';
import { User } from '../entity/user';
import { CreateUserCommand } from '../commands/create-user-command';

@Controller('users')
export class UserController {
  constructor(private commandBus: CommandBus, private queryBus: QueryBus) {}

  @Get()
  async findAll() {
    return await this.queryBus.execute<GetAllUsersQuery, User[]>(
      new GetAllUsersQuery(),
    );
  }
```

```
17
18    @Post()
19    async create(@Body('id') id: number, @Body('name') name: string) {
20      return await this.commandBus.execute<CreateUserCommand, User>(
21        new CreateUserCommand(id, name),
22      );
23    }
24  }
25
```

On line 3 and 5 we include empty POJO class, on line 9 we depend on the ***CommandBus*** and ***QueryBus*** but not on the actual app dependencies. This setup makes it possible that actions depend only on their own dependencies unlike services classes that have dependencies for all of their methods.

Action handler for CreateUserCommand is displayed below to showcase the actual ICommandHandler interface.

```
1   import { CommandHandler, EventBus, ICommandHandler } from '@nestjs/cqrs';
2   import { UserStore } from '../store/user-store';
3   import { User } from '../entity/user';
4   import { UserCreatedEvent } from '../events/user-created-event';
5   import { CreateUserCommand } from './create-user-command';
6
7   @CommandHandler(CreateUserCommand)
8   export class CreateUserCommandHandler
9   implements ICommandHandler<CreateUserCommand>
10    {
11    constructor(private userStore: UserStore, private eventBus: EventBus) {}
12
13    async execute(command: CreateUserCommand) {
14      const { id, name } = command;
15      const user = new User();
16      user.id = id;
17      user.name = name;
18      this.userStore.addUser(user);
19      this.eventBus.publish(new UserCreatedEvent(user));
20      return user;
21    }
22  }
```