NestJs Framework

A Progressive NodeJs Framework

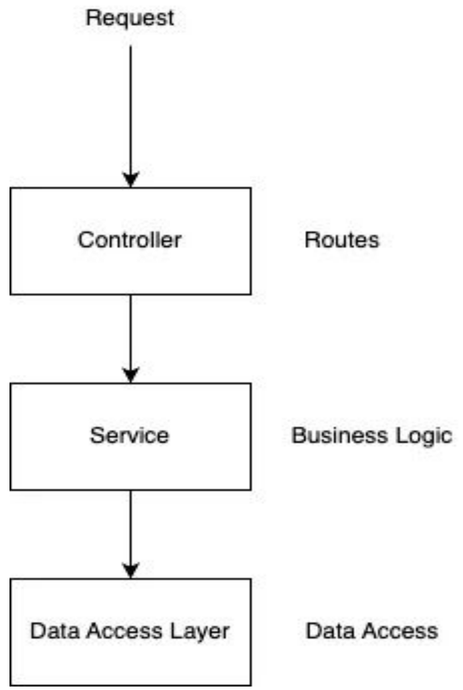
Introduction

- **Nest. js** is one of the fastest-growing Node. js frameworks for building efficient, scalable, and enterprise-grade backend applications using **Node. js**.

- It is known for producing highly testable, maintainable, and scalable applications using modern **JavaScript** and **TypeScript**.

- It's simple to learn and master, with a powerful CLI to boost productivity and ease of development.

Architecture



Installation

- Firstly, you need to install npm package of nestjs/cli to run nest using cli.

>>>> **npm install @nestjs/cli**

- To create a new project, run

>>>> **nest new <project_name>**

This command will install all the necessary dependencies and creates the file structure for running nestjs application.

File Usage

1. **Main.ts** – Starting point of application. Includes the appModule file to initialize all the dependencies required to run the application.
2. **App.module.ts** – Includes all the dependencies, handles the imports and exports of the application, controls the middlewares and guards.
3. **App.controller.ts** – Includes the route handlers for the application and mostly performs the business logic.
4. **App.service.ts** – Deals with the Database to provide the required data to the controllers to handle the business logic.
5. **App._.spec.ts** – Used to test the logic written in the file.

Controllers

- Controllers are used to handle request for the specific routes in the application.
- Controllers need to be registered as controllers in app.module file.
- **@Controller(<route>)** decorator is used to create a controller.
- Types of request:

  @Get(<sub-route>)

  @Post(<sub-route>)

  @Put(<sub-route>)

  @Delete(<sub-route>)

Request Objects in Controller

- Different request objects in route handlers are:

Body - @Body('id') id:<number> or @Body() bodyObejct: <Paramtype: bodyObejct>

Query - @Query('id') id:<number> or @Query() queryObject: <Paramtype: queryObject>

Parameters - @Param('id') id:<number> or @Param() paramObject: <Paramtype: paramObject>

Headers - @Headers('id') id:<number> or @Headers() headerObject: <Paramtype: headerObject>

Request - @Req() reqObject: Request

Response - @Res() resObject: Response

Syntax

```
@Controller(<route-name>)
export class <Class-Name>{
    <Request Type>(<sub-route>)
    <handler-name>(<Request-Object>): <Return-type> {
        // function logic
    }
}
```

Syntax

Eg:

```
@Controller(‘users’)
export class AppController{
        @Get(‘/getUser/:id’)
        getUser(@Param(‘id’) id: number){
                console.log(id)
        }
        @Post(‘/getUser/’)
        createUser(@Body(‘id’) id: number){
                console.log(id)
        }
}
```

Response Handling in Controllers

- **By default, nestjs returns 200 statusCode in response.**
- To make a custom responseCode for specific route-handler, we can use decorator :

  > @HttpCode(400) or @HttpCode(HttpStatus.NOT_FOUND)

- Other way is to access a response object using @Res() decorator and change the response status:

  > res.status(400).json({ message: “Not found” });

- Similarly, we can set response headers using **@Header(<header-field-name>,<value>)** decorator for a specific route-handler.

Response Handling in Controllers

- We can also redirect to another route using decorator:

  > @Redirect(<url-path>, <status-code>)

- We can also redirect from within the function using dynamic routing.

  > return {

  ‘url’: <url-path>/<dynamic-value>,

  ‘statusCode’: <status-code>

  }

Services

- Services are used to deal with database and handle the business logic.
- No decorator is required to create a services.
- Services will be used by other modules, so, they should be included as providers in the app.module file.
- Other classes can inject it in their constructor, and use its functions.
- It imports the model or schema, and deals with them to perform CRUD operations.

Schema

- **@Schema()** decorator is used to create the schema object in NestJs.
- **@Prop()** decorator is used to define the property of the schema fields.
- Type of the field must be defined while defining the schema.
- Schema class should extend the **Document** class used for creating the schema.
- Syntax:

```
@Schema()
export class User extends Document {
        @Prop({required: true})
        Name: string;
}
```

- **SchemaFactory** is used to create the schema from the class.
  > **SchemaFactory.createForClass(User)**

Dependency Injection

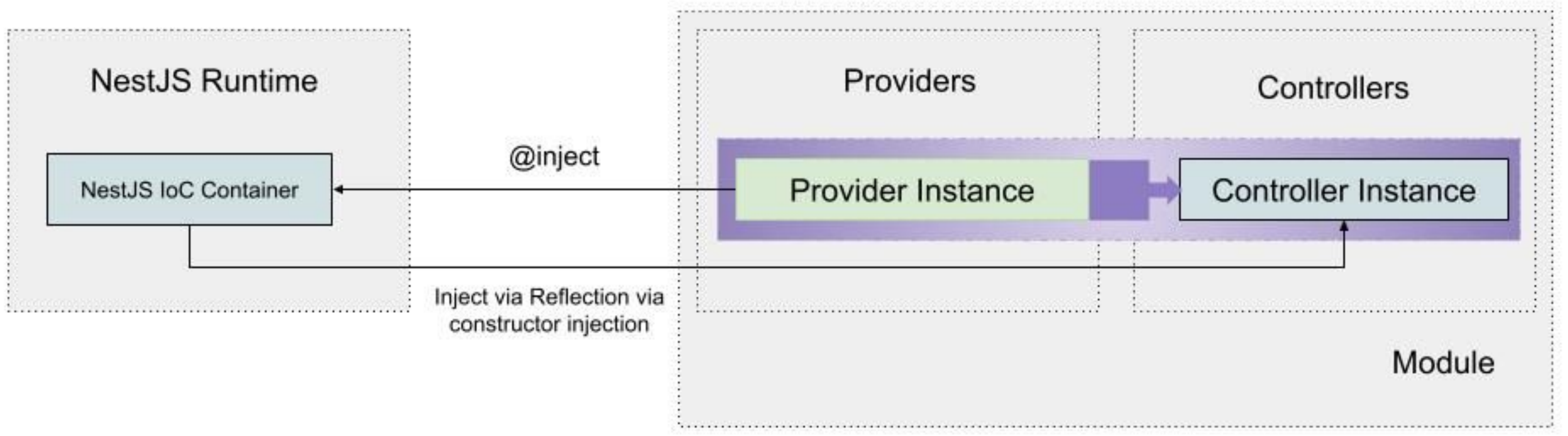
- Nestjs resolves the dependency hierarchy, using the dependency injection and it uses **IOC(Inversion of Control) Approach** for that.
- IOC Approach gives the module or dependency control to the framework and NestJs handles all the dependency hierarchies and dependencies injection in different modules.
- IOC container holds the modules that can be used as a dependency in other modules. This container is called provider.
- Using IOC Approach, we delegate the control to NestJs runtime for instantiation and managing dependencies in our application.

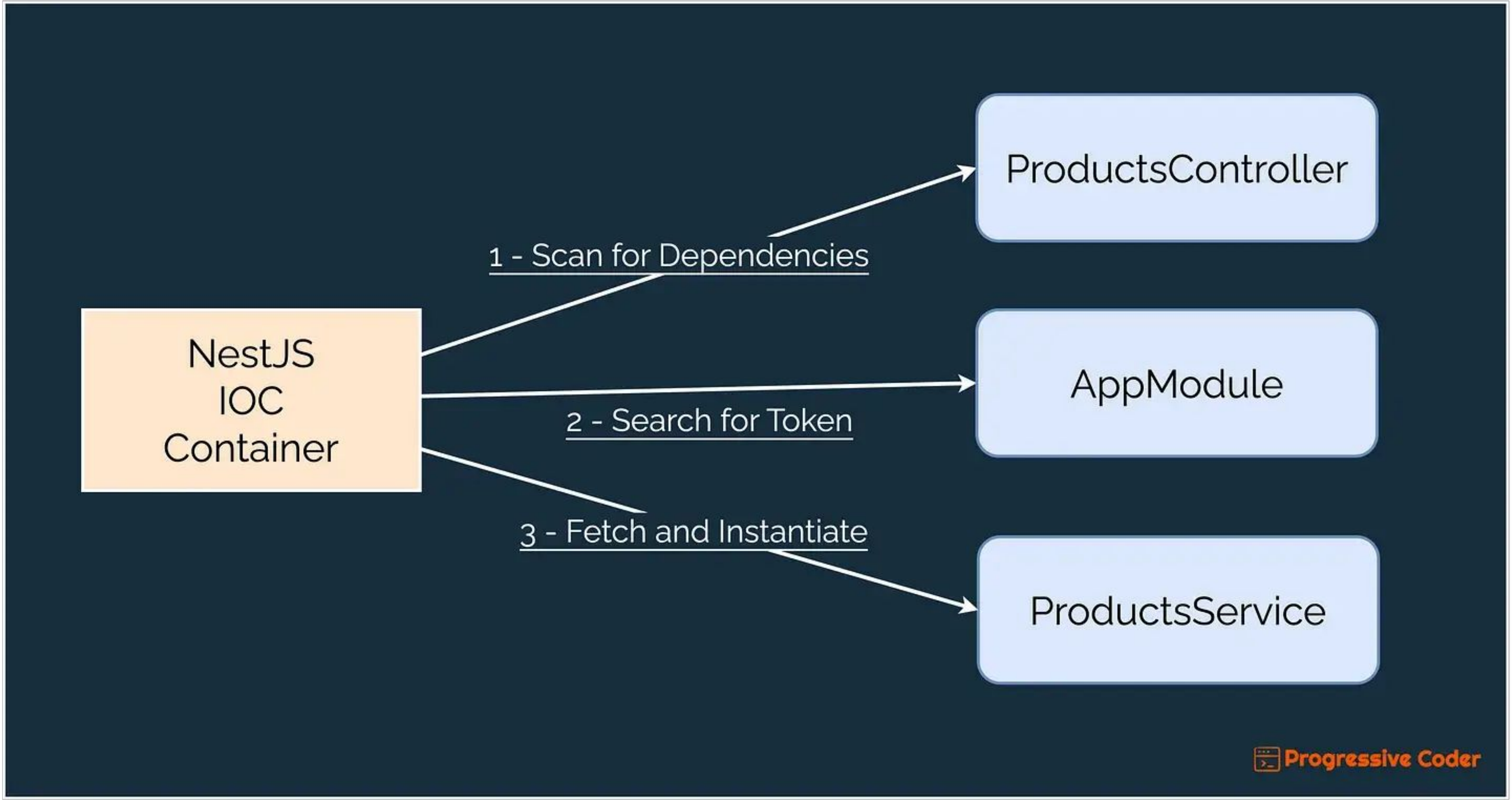
Dependency Injection

- To add the modules to the IOC Container, we just have to use the decorator **@Injectable()** for the class, we need to use as a dependency.
- Injection Token is assigned and used to identify each dependency uniquely in the application. These token can be string or className.
- When any module needs to use this, NestJs takes an injection token and creates an instance of the corresponding dependency and make it available to use in that module.
- To use such dependency inside the module, we need to instantiate it in the constructor using :

  > @Inject(<token-name>) <instance-name> : <Dependency>.

NestJs Depedency Injection




NestJS IOC Container
1 - Scan for Dependencies
2 - Search for Token
3 - Fetch and Instantiate
ProductsController
AppModule
ProductsService
Progressive Coder

Provider Scope

- To use module in different classes, register that module in the injector container of that module and the injector will create an instance for that module in that scope.
- Every dependency or module has a specific scope associated with it.
- Two modules in the same scope can inject that dependency but the modules outside the scope cannot directly inject that dependency.
- The dependency in other modules (out of the scope) can also be used by injecting the module that injects that dependency.

Provider Types

1. **Class Based - Standard Provider**

   - Class is used as a provider in the module.

   - Use Case: Services

2. **Non-Class based**

   - Number, Boolean, String, Object, Array, Function can be used as provider.

   - Use Case: DatabaseName, ConnectionObject, Url, Configs

3. **Factory**

   - Factory Function, Async Factory Function

   - Use Case: Dynamic or Conditional values, class instance

Injection Types

- We can inject the dependency using injection token either in the constructor or the property definition.
  a. **Constructor Injection:**
     - **Class Provider** >> constructor(@Inject(Account) account: Account) {.....}
     - **Non-Class Provider** >> constructor(@Inject(DB_NAME) dbName: String) {.....}

  b. **Property Injection:**
     - **Class Provider** >>
       ```
       class User { @Inject(Account)
                    account: Account
                    .....
       }
       ```
     - **Non-Class Provider** >>
       ```
       class User { @Inject(DB_NAME)
                    dbName: String
                    .....
       }
       ```

Standard Providers Registration

- To register the module as a dependency, you have to add @Injectable() decorator as well as you have to provide the module as provider in app.module file

  Syntax: Register > { provide: <Injection-token>, **useClass**: <module-name> }

  Use > constructor(<variable-name> : <injection-token>)

- When you provide the module name to providers in app.module, nestjs creates an instance of the class and provides it when asked.

  Note : By default, Same instance is shared to all the modules.

Standard Providers Registration

1. Providers: [{ provide: userStore, **useClass**: userStore }] or Providers: [userStore].

   Used as : constructor(@Inject(userStore) private store: userStore)

2. Providers: [{ provide: ‘STORE’, **useClass**: userStore }]

   Used as : constructor(@Inject(‘STORE’) private store: userStore)

3. Providers: [userStore, { provide: ‘STORE’, **useExisting**: userStore }]

   Used as : constructor(@Inject(‘STORE’) private store: userStore)

Value Providers Registration

- Syntax: Register > { provide: <Injection-token>, **useValue**: <value> }

  Use > constructor(<variable-name> : <injection-token>)

- For value as string:

  Providers: [{ provide: ‘DB_NAME’, **useValue**: ‘User’ }]

  Used as : constructor(@Inject(‘DB_NAME’) private dbname: string)

- Same syntax can be used for arrays, objects, boolean, function, etc

Factory Provider Registration

- Syntax: Register > { provide: <Injection-token>, **useFactory** : () => {} }

  Use > constructor(<variable-name> : <injection-token>)

- Eg:

  Providers: [{ provide: ‘Config’, **useFactory**: () => { return 1; } }]

  Used as : constructor(@Inject(‘Config’) private x: number)

- If async factory provider is used, till it is not resolved, the module using it will not get initialized.

Factory Provider Registration

- We can provide dependency to a factory provider, we just have to inject it.

```
Eg: providers: [
{ provide: ‘Config’,
          useFactory: (limit) => { return limit>0 ? limit : 0; },
          inject:[‘LIMIT’]  // ( inject: [{token:’LIMIT’, optional: true}] to make it optional injection)
},
{provide: ‘LIMIT’, useValue: 2},
]
Used as : constructor(@Inject(‘Config’) private x: number)
```

Injection Scopes

1. <u>**DEFAULT**</u>: Only single instance is created and it its shared between all the modules.

   Syntax: @Injectable()

2. <u>**REQUEST**</u>: New instance is created for every incoming request.

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

3. <u>**TRANSIENT**</u>: New instance is created every time, when dependency is injected.

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

Modules

- **App.module.ts** file is the main module which is initialized when the application is bootstrapped. All the dependencies, we need to include, should be there in app.module.ts directly or indirectly.
- Modules can be created using **@Module()** decorator in nestjs. This decorator takes following parameters:
  - **Imports**: All the modules that should be imported from another module should be imported here.
  - **Providers**: All the dependencies that are used by the other classes should be included here.
  - **Controllers**: All the controllers classes should be included here.
  - **Exports**: All the modules that should be used by another module should be exported from here.

Note: You can re-export the same module(i.e. You can export the imported module)

Types of Modules

1. **Feature Module**: If one feature named ‘User’ is created in application, then it’s module will be there and that module should be included in main app module.
2. **Shared Module**: If more than one feature uses that module, then it is shared module.
3. **Global Module**: If all the features uses that module, then instead of including that module in the other modules, we can create the shared module as global. This global module can be used by other feature modules without importing it.

   Note: Global Module may not get imported in other features modules, but it should still be present in main app module to get initialized.

Types of Modules - Dynamic Modules

4. **Dynamic Modules**: We can also register the module dynamically using syntax:

```
@Module({
        imports: [],
        Providers:[],
        controllers:[],
})
export class dynamicModule{
        register(dbModule):DynamicModules{
                return {
                        module: [<Module-Name>],
                        Providers:[dbModule],
                        controllers:[],
                }
        }
}
```

- To import DynamicModule, you have to use dynamicModule.register(dbModule) as a dependency while importing in app.module.

Dynamic Modules

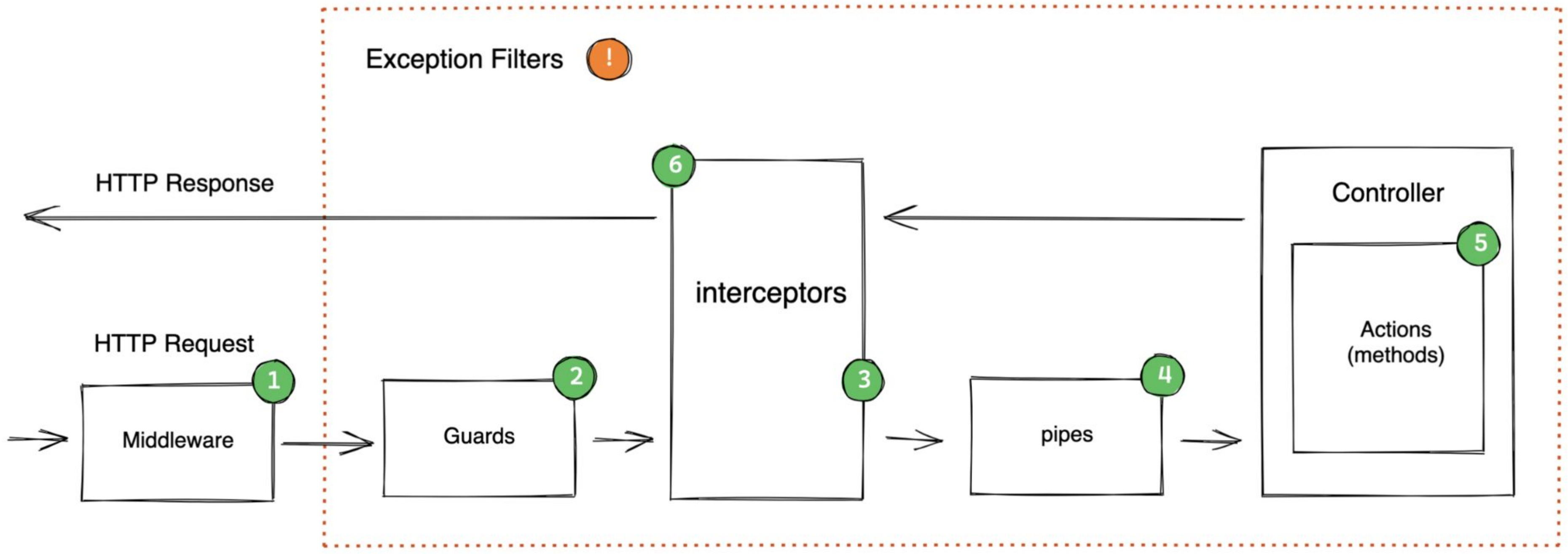
- We can create dynamic modules for the root(whole application) and for the feature(for specific service).
- For root: We can create it using syntax:

  Module.forRoot()

- For specific feature, we can create it using syntax:

  Module.forFeature()

Request and Response Lifecylce



Middlewares

- Middleware functions are called before the route handler.
- Middleware functions have access to request and response objects and can execute any code to change the request objects.
- We can implement custom middleware in either a function or a class, using **@Injectable()** decorator.
- The class should implement NestMiddleware Module and have to implement the **use()** function in which the logic for middleware is written.
- Use function has request and response objects which can be changed and along-with that it also has next object which is used to call the next function.

Applying Middleware

- Middlewares can’t be implemented in **@Module()** decorator.
- We set them up using **configure()** method of module class.
- Modules that include middleware has to implement **NestModule** interface.
- We can implement the middleware only for certain routes or certain controllers.
- We can also apply multiple middlewares for single route separated by commas.

Applying Middleware

Eg:

```
@Module({})
export class AppModule implements NestModule{
        configure(consumer:MiddlewareConsumer) {
                consumer.apply(<Middlewareclass>).forRoutes(<path>)
                consumer.apply(<Middlewareclass>).exclude (<path>)
        }
}
```

Guards

- Guards have a single responsibility.
- They determine whether a given request will be handled by the route handler or not, depending on the certain conditions like permissions, roles, etc at the run time
- They work similar to middlewares.
- But middlewares are used to handle request and response objects while guards are used for access to routes or resources based on specific conditions like authorization or authentication.
- Also, middleware doesn’t know which handler will be called after calling the next function but guards have access to execution context, so they know what’s going to be executed next.

Applying Guards

- Guards can be implemented on the route handler as well as on the controller level using **@UseGuards()** decorator. We can also use guards at the global level using **app.useGlobalGuards(<GuardClass>)** in main.ts file.
- Every guard must implement the **canActivate()** function which takes the execution context as an argument.
- These function returns the **boolean or Promise<Bollean> or Observable<Boolean>**.
- If the guard function returns true, then the user will be allowed to access the route.

Applying Guards

```
@Injectable()
export class AuthGuard implements CanActivate {
        canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observable<boolean> {
                const request = context.switchToHttp().getrequest();
                return validateRequest(request);
        }
}
// Guard logic
validateRequest(request: Request) : Boolean {
        return true;
}
```

Interceptors

- Interceptors are inspired by the **Aspect-Oriented Programming**.
- AOP refers to the separation of the business logic and the cross-cutting point like security, logging, error handling, performance monitoring.
- Interceptors can be used for
  a. Bind extra logic before/after execution
  b. Transform the result returned from the function.
  c. Transform the exception thrown
- They can be used similar to middlewares but the interceptors allow us to manipulate the response objects by continuously observing the response coming from the implementation.

Applying Interceptors

- Each interceptor should implement **intercept()** method.
- This method take two arguments: **ExecutionContext** similar to middlewares and **call handler** which implements the handle function which can be used to invoke any function at any point of time in your interceptor.
- As, this method returns as **Observable object**, you can access the response or request at any point of time in your interceptor.
- When an interceptor is applied, **.pipe()** method intercepts the incoming request and modifies the request object or response object or both when needed.
- Interceptors can be implemented using @UseInterceptors() decorator at controller level or global level using app.useGlobalInterceptors(<Interceptor>).

Applying Interceptors

Eg:

```
@Module({})
export class InterceptorClass implements NestInterceptor{
        intercept(context, next: callHandler) : Observable<any> {
                console.log(‘Before.........’);
                return next.handle().pipe(
                        tap(()=> console.log(‘After........’)),
                        )
        }
}
```

Pipes

- Pipes are used to transform and validate the parameters of the request object.
- NestJs provides us some of the built-in pipes to directly use it within the request object.
- We can also create our custom pipes using NestJs
- To apply pipes to each requestObject, we can use :

  @Param('id', ParseIntPipe) id: number
- To apply pipes to the controller or route level, we can use @UsePipes() decorator:

  @UsePipes(ParseIntPipe)

Transformation Pipes

- Some of the built-in transformation pipes are:
    a. ParseIntPipe
    b. PaseFloatPipe
    c. ParseBoolPipe
- These pipes converts the incoming value to the desired datatype.
- To create a custom error message types for built-in transformation pipes, we can do it using:

  new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE })
- If we create custom Pipes, then we need to instantiate it using **new** keyword.

Validation Pipes

- Some of the built-in validation pipes are:
  a. **ParseUUIDPipe** - Checks whether the passed id is valid uuid or not. We can also add custom checks if we need the uuid of only specific version.
  b. **PaseEnumPipe** - Checks whether the passed id value is supported by the custom enum or not.
  c. **ParseArrayPipe** - To perform this pipe, class-transformer and class-validator is required to be installed.
     - **Class-Transformer** - Transforms the array of data into specific type.
     - **Class-Validator** - Validates the array of data.

Eg: **new ParseArrayPipe({ items: Number, separator: ‘,’ })**

Validation Pipes Using Decorators

- We can also create validation using decorators from **‘class-validator’.**
  a. @IsString()
  b. @IsOptional()
  c. @IsEmail()
  d. @IsBoolean()
  e. @IsNumber()
  f. @ArrayMinSize()
  g. @IsIn()
  h. @IsNotEmpty
  i. @IsNumber
  j. @ValidateNested, etc
- To use this validation in request objects, we have to create a **DTO object** with such decorators and use **ValidationPipe** in the request object.

DTO - Data Transfer Objects

- DTO is the proper structure of how the input data will be.
  ```
  Eg: export class UserDTO {
          Name: string,
          Email: string
  }
  ```
- We can use decorators with this dto object like this:
  ```
  Eg: export class UserDTO {
          @IsString()
          @IsNotEmpty()
          Name: string,

          @IsEmail())
          Email: string
  }
  ```
- We can apply this validation pipe as @Param(ValidationPipe) body: UserDTO.

Custom Pipes

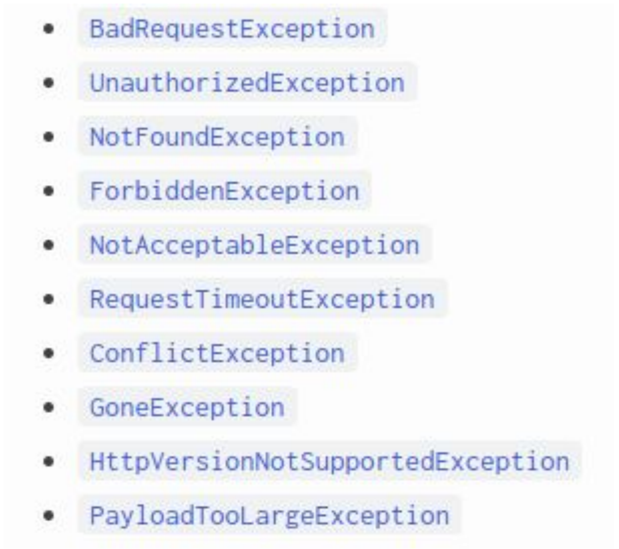
- We can create our custom pipes by implementing class **PipeTransform** and calling its method named - **transform()**
- Transform method takes two arguments: value and metadata.
- Value gives the input on which this pipe will be used and metadata provides the property of data.
- We can apply business logic on the input values and return the output values from the transform method.

Pipes on Module and Global Level

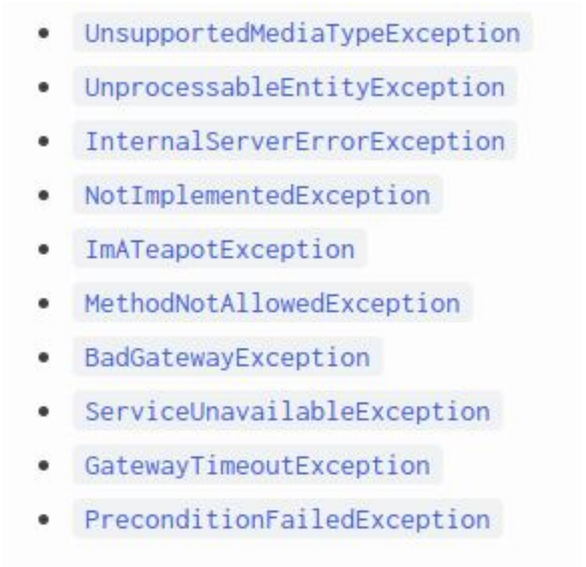
- We can pass pipes on the module-level as well as on the global level.
- To pass the pipe on the controller-level, we need to provide it in the **@UsePipes()** decorator.
- To include the pipes on the global level, we can add **app.useGlobalPipes(<Pipe_class_name>)** in the main.ts file and the pipe will apply to all the modules in the application.

Exception Filters

- NestJs provides a set of standard exceptions that inherit from the base HttpException.



- `BadRequestException`
- `UnauthorizedException`
- `NotFoundException`
- `ForbiddenException`
- `NotAcceptableException`
- `RequestTimeoutException`
- `ConflictException`
- `GoneException`
- `HttpVersionNotSupportedException`
- `PayloadTooLargeException`



- `UnsupportedMediaTypeException`
- `UnprocessableEntityException`
- `InternalServerErrorException`
- `NotImplementedException`
- `ImATeapotException`
- `MethodNotAllowedException`
- `BadGatewayException`
- `ServiceUnavailableException`
- `GatewayTimeoutException`
- `PreconditionFailedException`

Custom Exception Filters

- @UseFilters(<Filter-Name>) is used to apply the custom filter to the controller or route level. At global level, it can be applied by app.useglobalFilters(<filter>).
- Custom filter class should implement the ExceptionFilter module.
- catch() method should be implemented by the exception handler class.
- Catch method takes two arguments: exception type and ArgumentsHost.
- @Catch(<Exception-type>) is used as a decorator for the exception class to catch specific exception.

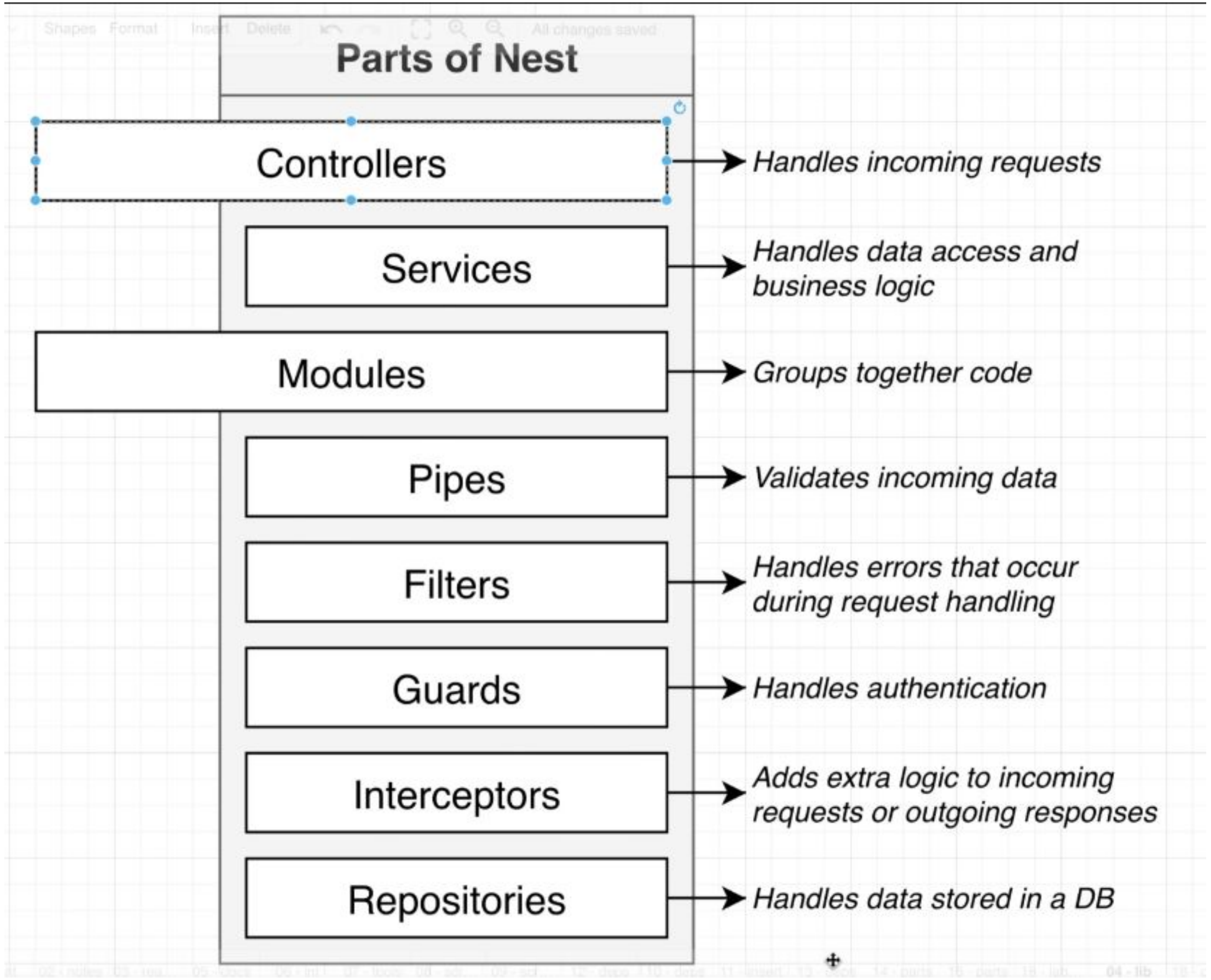
Applying Exception Filter

Eg:

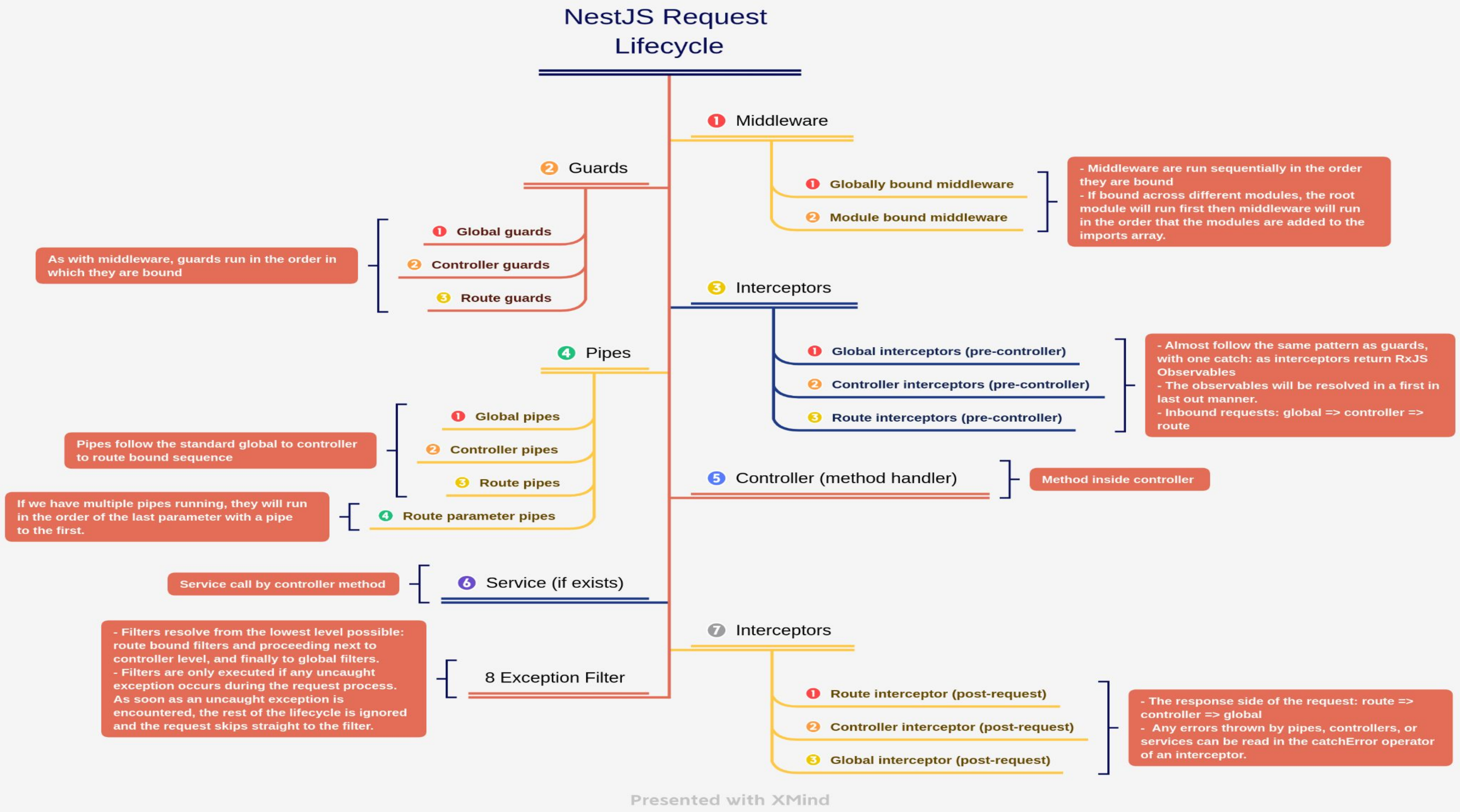
```
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
        catch(exception: HttpException, host: ArgumentsHost) {
                // return exception using response object.
        }
}
```

This exception filter can be applied on controller using @UseFilters(HttpExceptionFilter) decorator.

Now, Let's have a quick recap of what we covered in the session.


Parts of Nest
Controllers
Handles incoming requests
Services
Handles data access and business logic
Modules
Groups together code
Pipes
Validates incoming data
Filters
Handles errors that occur during request handling
Guards
Handles authentication
Interceptors
Adds extra logic to incoming requests or outgoing responses
Repositories
Handles data stored in a DB

NestJS Request Lifecycle



1. Middleware
   1. Globally bound middleware
   2. Module bound middleware

   - Middleware are run sequentially in the order they are bound
   - If bound across different modules, the root module will run first then middleware will run in the order that the modules are added to the imports array.

2. Guards
   1. Global guards
   2. Controller guards
   3. Route guards

   As with middleware, guards run in the order in which they are bound

3. Interceptors
   1. Global interceptors (pre-controller)
   2. Controller interceptors (pre-controller)
   3. Route interceptors (pre-controller)

   - Almost follow the same pattern as guards, with one catch: as interceptors return RxJS Observables
   - The observables will be resolved in a first in last out manner.
   - Inbound requests: global => controller => route

4. Pipes
   1. Global pipes
   2. Controller pipes
   3. Route pipes

   Pipes follow the standard global to controller to route bound sequence

   4. Route parameter pipes

   If we have multiple pipes running, they will run in the order of the last parameter with a pipe to the first.

5. Controller (method handler)

   Method inside controller

6. Service (if exists)

   Service call by controller method

7. Interceptors
   1. Route interceptor (post-request)
   2. Controller interceptor (post-request)
   3. Global interceptor (post-request)

   - The response side of the request: route => controller => global
   - Any errors thrown by pipes, controllers, or services can be read in the catchError operator of an interceptor.

8 Exception Filter

- Filters resolve from the lowest level possible: route bound filters and proceeding next to controller level, and finally to global filters.
- Filters are only executed if any uncaught exception occurs during the request process. As soon as an uncaught exception is encountered, the rest of the lifecycle is ignored and the request skips straight to the filter.

NestJs With GraphQL

- We can use NestJs with graphql, we just have to create two new files and register them as a dependency in the app.module.

1. App.input.ts : It shows how the data coming from the graphql api should be.

   @InputType() decorator should be used for its class.

2. App.Resolver.ts : It works similar to controller. It uses the services module to perform the query and mutation on the database.

   @Resolver(()=><return-type> decorator should be used for its class.

- Query: Get request by the graphql api.
- Mutation: Create, Update or Delete request by the graphql api.

GraphQL InputType

```
@InputType()
export class OrderInput {

  @Field()
  deliveryDone: boolean;

  @Field()
  orderCancelled: boolean;

  @Field()
  deliveryDate: string;

  @Field()
  deliveryAddress: string;

  @Field()
  receiverPhone: string;

  @Field({ defaultValue: 'COD' })
  paymentMethod: string;
}
```

GraphQL Resolver

```
@Resolver(() => Order)
export class AppResolve  {
  constructor(
    private readonly appService: AppService,
  ) {}
  @Query(() => [Order])
  async orders(): Promise<Order[]> {
    return await this.appService.findAll();
  }
  @Mutation(() => Order)
  async delete(
    @Args('id', { type: () => ID }) id: number,
  ): Promise<Order> {
       return await this.appService.delete(id);
  }
}
```

GraphQL Object type

- To change the database fields with graphql api, we need to create an object of the data that is to be inserted into the database.
- We can create this using **@ObjectType()** decorator and using **@Field()** decorator for each database field.
- This ObjectType can also be defined in schema file.

Syntax:

```
@ObjectType()
@Schema()
export class Customer extends Document {
  @Field({ nullable: false })
  @Prop({ required: true })
  name: string;
}
```

Thank You
Your logo