NestJS Cheat Sheet

A comprehensive cheat sheet for NestJS, covering core concepts, modules, controllers, services, and common utilities.

Core Concepts

Modules

Definition: Modules organize application components.

Usage: Use `@Module()` decorator to define a module.

Example:

```
import { Module } from '@nestjs/common';
import { AppController } from
'./app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Imports: Used to import other modules.
Controllers: Defines controllers within the module.
Providers: Defines services, repositories, and other providers within the module.

Controllers

Definition: Controllers handle incoming requests and return responses.

Usage: Use `@Controller()` decorator to define a controller.

Example:

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

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

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

@Get(), @Post(), @Put(), @Delete(), @Patch(), @Options(), @Head(): Methods to handle specific HTTP requests.

Services

Definition: Services contain business logic and are used by controllers.

Usage: Use `@Injectable()` decorator to define a service.

Example:

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

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

Dependency Injection

Providers and Injection

@Injectable(): Marks a class as a provider.
@Inject(): Injects a provider into a class.

Constructor Injection Example:

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

@Injectable()
export class CatsService {
  constructor(@Inject('DATABASE_CONNECTION') private readonly connection:
Connection) {}
}
```

Custom Providers: Allows defining custom provider logic.

Value Provider Example:

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

const config = { api_key: 'yourkey' };

@Module({
  providers: [
    {
      provide: 'CONFIG',
      useValue: config,
    },
  ],
})
export class AppModule {}
```

Scopes

Default Scope (Singleton)	Instances are created once and shared across the application.
Request Scope	Instances are created per request. Use `REQUEST` from `@nestjs/core`.
Transient Scope	Instances are not shared. New instance on each injection.

Middleware & Interceptors

Middleware

Definition: Functions that are executed before the route handler.

Usage: Implement `NestMiddleware` interface.

Example:

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

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

Applying Middleware:

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './logger.middleware';

@Module({})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('*');
  }
}
```

Interceptors

Definition: Interceptors augment request/response flow.

Usage: Implement `NestInterceptor` interface.

Example:

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

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}
```

Applying Interceptors:

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

@UseInterceptors(LoggingInterceptor)
export class AppController {}
```

Pipes

Pipes Overview

Definition: Pipes transform or validate request data.

Types:

- **Transformation:** Transform input data to the desired output.
- **Validation:** Evaluate input data and throw an exception if invalid.

Built-in Pipes

`ValidationPipe`	Validates request body using class-validator.
`ParseIntPipe`	Parses a string to an integer.
`ParseBoolPipe`	Parses a string to a boolean.
`ParseArrayPipe`	Parses a string to an array.
`ParseUUIDPipe`	Parses a string to a UUID.

Custom Pipes

Creating a Custom Pipe:

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

@Injectable()
export class CustomValidationPipe implements
PipeTransform {
  transform(value: any, metadata:
ArgumentMetadata) {
    if (!this.isValid(value)) {
      throw new
BadRequestException('Validation failed');
    }
    return value;
  }

  private isValid(value: any): boolean {
    // Validation Logic Here
    return true; // Replace with actual
validation
  }
}
```

Applying a Pipe:

```
import { Controller, Get, Query, UsePipes }
from '@nestjs/common';
import { CustomValidationPipe } from
'./custom-validation.pipe';

@Controller()
export class AppController {
  @Get()
  @UsePipes(new CustomValidationPipe())
  getHello(@Query('param') param: string):
string {
    return `Hello ${param}`;
  }
}
```