




Aula 6

Inversão de dependência

Sumário

- 1 Recapitulando: Injeção de dependência
- 2 Recapitulando: Interfaces e Polimorfismo
- 3 Inversão de dependência
- 4 Decorators
- 5 typedi



Recapitulando: Injeção de dependência

Injeção de Dependência

Recapitulando

Injeção de dependência é um *Design Pattern* que visa **desacoplar** a implementação de uma classe das dependências que ela utiliza

Recapitulando

Injeção de dependência



```
import { Request, Response } from "express";
import { UserService } from "../services/UserService";

export class UserController {
  public static async listUsers(req: Request, res: Response) {
    const users = await UserService.listUsers()
    res.json(users)
  }
}
```

Recapitulando

Injeção de dependência

```
import { Request, Response } from "express";
import { UserService } from "../services/UserService";

export class UserController {
  public static async listUsers(req: Request, res: Response) {
    const users = await UserService.listUsers()
    res.json(users)
  }
}
```

```
import { Request, Response } from "express";
import { UserService } from "../services/UserService";

export class UserController {
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

Injeção de Dependência

Recapitulando

Pontos de atenção

1. Definir as dependências como **atributos** da classe

```
import { Request, Response } from "express";
import { UserService } from "../services/UserService";

export class UserController {
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```


Injeção de Dependência

Recapitulando

Pontos de atenção

2. Usar a *keyword* **this** para utilizar a dependência

```
import { Request, Response } from "express";
import { UserService } from "../services/UserService";

export class UserController {
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```


Injeção de Dependência

Recapitulando

Pontos de atenção


3. Usar **arrow function** para garantir que **this** terá a referência correta

```
import { Request, Response } from "express";
import { UserService } from "../services/UserService";

export class UserController {
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```



Recapitulando: Interfaces e Polimorfismo

Interfaces e Polimorfismo

Recapitulando

Interfaces são estruturas dentro da Programação Orientada a Objetos (POO ou OOP) que permitem definir **contratos** de comunicação entre classes.

Interfaces **não possuem implementações**, elas possuem apenas as definições dos atributos.

Em typescript, **classes** podem **implementar** mais de uma interface.

Interfaces e Polimorfismo

Recapitulando



```
import { User } from "../models/User";

export interface UserService {
  listUsers(): Promise<User[]>
}
```


Interfaces e Polimorfismo

Recapitulando



```
import { User } from "../models/User";
import { UserService } from "../contracts/UserService";

export class UserServiceHandler implements UserService {
  public async listUsers(): Promise<User[]> {
    return [{ id: '1', name: 'Ada' }, { id: '2', name: 'Pablo' }]
  }
}
```

Interfaces e Polimorfismo

Recapitulando

Pontos de atenção

1. A classe deve possuir **todos os atributos** da interface que implementa, seguindo as **mesmas assinaturas** de funções

```
import { User } from "../../models/User";

export interface UserService {
  listUsers(): Promise<User[]>
}
```

```
import { User } from "../models/User";
import { UserService } from "../contracts/UserService";

export class UserServiceHandler implements UserService {
  public async listUsers(): Promise<User[]> {
    return [{ id: '1', name: 'Ada' }, { id: '2', name: 'Pablo' }]
  }
}
```

Interfaces e Polimorfismo

Recapitulando

Pontos de atenção

2. Se uma função é **assíncrona**, a sua interface obrigatoriamente deve retornar uma **Promise**

```
import { User } from "../../models/User";

export interface UserService {
  listUsers(): Promise<User[]>
}
```

```
import { User } from "../../models/User";
import { UserService } from "../contracts/UserService";

export class UserServiceHandler implements UserService {
  public async listUsers(): Promise<User[]> {
    return [{ id: '1', name: 'Ada' }, { id: '2', name: 'Pablo' }]
  }
}
```

Interfaces e Polimorfismo

Recapitulando

Polimorfismo é uma característica da POO que diz que uma classe é do mesmo tipo:

- Das interfaces que implementa;
- Da classe que herda.

Dessa forma se uma classe espera **receber uma interface**, podemos **passar outra classe que a implementa**.

Interfaces e Polimorfismo

Recapitulando



```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";

export class UserController {
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

Interfaces e Polimorfismo

Recapitulando



```
const userService = new UserServiceHandler()  
const userController = new UserController(userService)
```

Inversão de dependência

Inversão de dependência

Inversão de dependência é uma técnica em que, em vez de definirmos uma classe como dependência de outra, definimos um contrato de comunicação.

Esse contrato é feito utilizando **uma interface** e fazendo com que **outras classes a implementem**.

Inversão de dependência



```
import { User } from "../../models/User";

export interface UserService {
  listUsers(): Promise<User[]>
}
```



```
import { Service } from "typedi";
import { User } from "../../models/User";
import { UserService } from "../contracts/UserService";

@Service()
export class UserServiceHandler implements UserService {
  public async listUsers(): Promise<User[]> {
    return [{ id: '1', name: 'Ada' }, { id: '2', name: 'Pablo' }]
  }
}
```

Inversão de dependência



```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";

export class UserController {
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

Inversão de dependência

Vantagens

1. **Diminuição de acoplamento:** Ao utilizarmos a inversão de dependência, introduzimos uma abstração entre as dependências. Dessa forma, se quisermos modificar a implementação, podemos criar uma nova classe com essa nova implementação, em vez de alterar a anterior.

Inversão de dependência

Vantagens

2. **Testabilidade**: Abstrações facilitam escrever testes unitários, já que permitem a criação de “mocks”, tornando os testes mais eficientes.

Inversão de dependência

Vantagens

3. **Manutenibilidade:** A inversão de dependência diminui o impacto de mudanças no código. Com essa estrutura modular e abstrata, é mais simples atualizar ou substituir componentes do código sem afetar todo o sistema.

Se houver a necessidade de mudar uma implementação, tendo a opção de criar uma nova classe faz com que não haja a necessidade de alterar códigos que já podem estar sendo usados em outras partes do sistema, o que poderia gerar problemas inesperados. Sem contar que extingue a necessidade de entender códigos antigos que não foi você quem criou.

Inversão de dependência

Vantagens

4. **Escalabilidade**: A abstração permite a adição de novas implementações sem alterar códigos existentes. Assim, é mais fácil estender a funcionalidade de um sistema, ainda preservando comportamento antigos.

Inversão de dependência



```
import { User } from "../models/User";
import { UserService } from "../contracts/UserService";

export class UserServiceHandlerFromDB implements UserService {
  private userRepository: UserRepository

  public async listUsers(): Promise<User[]> {
    return userRepository.listAll()
  }
}
```

Decorators

Decorators

Decorators fazem parte de uma estratégia de alterar classes ou seus atributos de uma forma fácil e unificada

O nome vem da ideia de que eles “**decoram**” classes e seus atributos para funcionarem da forma que gostaríamos

Classificação

1. **Decorators de classes:** Servem para alterarem classes

```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "typedi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

Classificação

2. **Decorators de parâmetros:** Servem para alterar parâmetros

```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "typedi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

Classificação

3. **Decorators de métodos:** Servem para alterar métodos ou funções

```
import 'reflect-metadata';
import { Controller, Param, Body, Get, Post, Put, Delete } from 'routing-controllers';

@Controller()
export class UserController {
  @Get('/users')
  getAll() {
    return 'This action returns all users';
  }

  @Get('/users/:id')
  getOne(@Param('id') id: number) {
    return 'This action returns user #' + id;
  }

  @Post('/users')
  post(@Body() user: any) {
    return 'Saving user...';
  }

  @Put('/users/:id')
  put(@Param('id') id: number, @Body() user: any) {
    return 'Updating a user...';
  }

  @Delete('/users/:id')
  remove(@Param('id') id: number) {
    return 'Removing user...';
  }
}
```


Decorators

Pontos importantes

1. **Utilizam o símbolo @**: Na maioria das linguagens de programação, decorators são precedidos por @

```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "typedi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

Decorators

Pontos importantes

2. **Utilizam ()**: Em typescript, decorators são sempre acompanhados de **()** e podem ou não receber atributos dentro deles

```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "typedi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

typedi

typedi

typedi é uma *lib* famosa do Typescript que permite realizar a **injeção de dependência** utilizando decorators

Ele também permite utilizar a estratégia de **inversão de dependência**



```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "typedi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

typedi

Configuração

0. **Documentação:** Utilizem sempre a documentação do typedi para saber como configurá-lo em seu projeto: <https://www.npmjs.com/package/typedi>

typedi

Configuração

1. **Instalação:** Além de instalá-lo, é necessário instalar junto a *lib reflect-metadata*



```
npm install typedi reflect-metadata
```

typedi

Configuração

2. **Importar o reflect-metadata:** A *lib* reflect-metadata deve ser a primeira linha de código a ser rodada no seu projeto



```
import 'reflect-metadata';
import { setupContainer } from './containers';
setupContainer()

import app from './server'
import { AddressInfo } from 'net'

const listener = app.listen(3000, () => {
  const address = listener.address() as AddressInfo
  console.log('Listening on port ' + address?.port);
});
```


tyepdi

Configuração

3. **Alterações no `tsconfig.json`:** A *lib* `reflect-metadata` deve ser a primeira linha de código a ser rodada no seu projeto



```
"emitDecoratorMetadata": true,  
"experimentalDecorators": true,
```

Funcionalidades do typedi

1. **@Service**: Indica que esta classe será utilizada na injeção de dependência e isso fará com que ela seja registrada no *container* do typedi



```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "typedi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

tyepdi

Funcionalidades do typedi

2. @Inject

```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "typedi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```

Funcionalidades do typedi

3. **Container.get**: Se formos utilizar uma dependência fora de uma classe, podemos utilizar a função `Container.get`



```
import { Router } from 'express'
import { UserController } from './controllers/UserController'
import { Container } from 'typedi'

const userController = Container.get(UserController)

const userRouter = Router()

userRouter.get('/', userController.listUsers)

export default userRouter
```


Pontos importantes

1. **Injetando Interfaces:** Como interfaces não existem quando o código é transpilado para Javascript, precisamos indicar pro typedi como injetá-las. Para isso:

- Criamos uma string constante que servirá como chave (usamos o Token do typedi para isso)
- No Container, devemos usar a função `Container.set`

tyepdi

Pontos importantes

1. Injetando Interfaces



```
import { Token } from "tyepdi";
import { User } from "../models/User";

export const UserService = new Token("UserService")
export interface UserService {
  listUsers(): Promise<User[]>
}
```

tyepdi

Pontos importantes

1. Injetando Interfaces



```
import { Container } from 'tyepdi';
import { UserServiceHandler } from '../services/UserServiceHandler';
import { UserService } from '../services/contracts/UserService';

export const setupContainer = () => {
  const userServiceHandler = Container.get(UserServiceHandler)
  Container.set(UserService, userServiceHandler)
}
```

tyepdi

Pontos importantes

1. Injetando Interfaces

```
import { Request, Response } from "express";
import { UserService } from "../services/contracts/UserService";
import { Inject, Service } from "tyepdi";

@Service()
export class UserController {
  @Inject(UserService)
  private userService: UserService

  constructor(userService: UserService) {
    this.userService = userService
  }

  public listUsers = async (req: Request, res: Response) => {
    const users = await this.userService.listUsers()
    res.json(users)
  }
}
```


Pontos importantes

2. **Configurar o Container primeiro:** Sempre devemos fazer a configuração do Container antes de qualquer trecho de código nosso (até mesmo antes de imports)

```
import 'reflect-metadata';
import { setupContainer } from './containers';
setupContainer()

import app from './server'
import { AddressInfo } from 'net'

const listener = app.listen(3000, () => {
  const address = listener.address() as AddressInfo
  console.log('Listening on port ' + address?.port);
});
```

Prática

Parte 1: Inversão de dependência

1. Comecem fazendo a inversão de dependência
2. Criem as interfaces para representar as comunicações entre classes
3. Façam com que classes implementem essas interfaces
4. Façam isso em um endpoint e depois apliquem aos demais

Parte 2: typedi

1. Depois de terem concluído a inversão de dependência, comecem a configuração do `typedi`
2. Instalem as *libs*, façam os devidos `imports` e alterem o `tsconfig.json`
3. Criem os `Tokens` para as interfaces
4. Configurem o `Container`
5. Utilizem os `Decorators`
6. Façam isso para um endpoint; garantam que está funcionando; e, então, apliquem aos demais

Materias de Aula

1. Repositório do Github com exemplos de código:
<https://github.com/joaogolias/ada-node-avancado-aula-6-inversao-de-dependencia>

Obrig.ada