Aula 5

Created	@February 9, 2024 12:21 PM
Data	@February 9, 2024
Status	Não iniciado
Tipo	Aula

Refatoração API para utilizar instâncias de classes ao invés de classes estáticas

Diferença entre classe estática e classe instanciada

A diferença fundamental entre classes estáticas e construtores em TypeScript (e em linguagens orientadas a objetos em geral) reside em como esses elementos são usados e em que contexto operam.

1. **Construtores:**
   - Um construtor é uma função especial dentro de uma classe que é chamada automaticamente quando você cria uma instância dessa classe usando a palavra-chave `new`. Ele é usado para inicializar propriedades e realizar outras tarefas de configuração na instância recém-criada.

   Exemplo:

```
typescriptCopy code
class Car {
  constructor(public make: string, public model: string) {
```

```
    // Código de inicialização
  }
}

const myCar = new Car('Toyota', 'Camry');
```

Neste exemplo, o construtor `Car` é chamado quando uma nova instância de `Car` é criada, e ele recebe os valores para as propriedades `make` e `model`.

2. **Classes Estáticas:**
   - As classes estáticas não são instanciadas e não podem ser instanciadas. Elas contêm membros que pertencem diretamente à classe, em vez de instâncias específicas da classe. Métodos e propriedades estáticos são acessados diretamente na classe, sem a necessidade de criar uma instância.

   Exemplo:

```
typescriptCopy code
class MathOperations {
  static add(x: number, y: number): number {
    return x + y;
  }
}

const result = MathOperations.add(5, 3);
```

Neste exemplo, `add` é um método estático da classe `MathOperations`. Não é necessário criar uma instância de `MathOperations` para chamar o método `add`.

Considerações:

- **Instância vs. Classe:**
  - Os construtores operam no contexto de uma instância específica da classe, enquanto membros estáticos operam no contexto da própria

classe.

- **Acesso a Membros:**
  - Para acessar membros de instância (não estáticos), você precisa criar uma instância da classe. Para acessar membros estáticos, você pode referenciar diretamente a classe.
- **Utilização:**
  - Use construtores quando precisar criar instâncias da classe e inicializar propriedades específicas dessa instância. Use membros estáticos quando a funcionalidade ou propriedade pertence à classe em si, independentemente de instâncias específicas.
- **Estado Compartilhado:**
  - Membros estáticos compartilham estado entre todas as instâncias da classe, enquanto cada instância possui seu próprio estado independente.

Em resumo, escolha entre construtores e membros estáticos com base na natureza da funcionalidade que você está implementando e se ela faz mais sentido no contexto de instâncias específicas ou da classe como um todo.

Exemplos de Uso de Classes Estáticas:

1. **Métodos Utilitários:**
   - Se você tem um conjunto de métodos utilitários que não precisam manter estado de instância, uma classe estática pode ser apropriada.

```
typescriptCopy code
class MathOperations {
  static add(x: number, y: number): number {
    return x + y;
  }

  static multiply(x: number, y: number): number {
```

```
    return x * y;
  }
}

const sumResult = MathOperations.add(5, 3);
const productResult = MathOperations.multiply(5, 3);
```

2. **Constantes:**
   - Se você precisa de constantes que são compartilhadas entre todas as instâncias da classe (ou não estão diretamente associadas a uma instância específica), você pode usá-las como propriedades estáticas.

```
typescriptCopy code
class Constants {
  static PI: number = 3.14;
}

console.log(Constants.PI);
```

3. **Singleton:**
   - Quando você deseja implementar um padrão Singleton, garantindo que apenas uma instância da classe exista.

```
typescriptCopy code
class Singleton {
  private static instance: Singleton;

  private constructor() {
    // Construtor privado para evitar instâncias adicionai
s
  }
```

```
  static getInstance(): Singleton {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
}

const singletonInstance1 = Singleton.getInstance();
const singletonInstance2 = Singleton.getInstance();

console.log(singletonInstance1 === singletonInstance2); //
true
```

Exemplos de Uso de Instâncias de Classe:

1. **Modelagem de Objetos:**
   - Ao modelar objetos do mundo real que têm estado específico de instância.

```
typescriptCopy code
class Car {
  private make: string;
  private model: string;

  constructor(make: string, model: string) {
    this.make = make;
    this.model = model;
  }

  getDetails(): string {
    return `${this.make} ${this.model}`;
  }
}
```

```
const myCar = new Car('Toyota', 'Camry');
console.log(myCar.getDetails());
```

2. **Interatividade:**
   - Quando você precisa manter o estado e interagir com instâncias específicas.

```
typescriptCopy code
class Counter {
  private count: number = 0;

  increment(): void {
    this.count++;
  }

  getCount(): number {
    return this.count;
  }
}

const counterInstance1 = new Counter();
const counterInstance2 = new Counter();

counterInstance1.increment();
console.log(counterInstance1.getCount()); // 1

counterInstance2.increment();
console.log(counterInstance2.getCount()); // 1 (cada instâ
ncia mantém seu próprio estado)
```

3. **Injeção de Dependência:**
   - Quando você deseja usar a injeção de dependência para fornecer dependências a uma classe.

```
typescriptCopy code
class Logger {
  log(message: string): void {
    console.log(message);
  }
}

class ProductService {
  private logger: Logger;

  constructor(logger: Logger) {
    this.logger = logger;
  }

  getProductDetails(productId: string): void {
    // Lógica do serviço
    this.logger.log(`Detalhes do produto ${productId}`);
  }
}

const consoleLogger = new Logger();
const productService = new ProductService(consoleLogger);
productService.getProductDetails('123');
```

Em resumo, você deve escolher entre classe estática e instância de classe com base na natureza da funcionalidade que você está implementando. Classes estáticas são adequadas para funcionalidades que não precisam de estado de instância, enquanto instâncias de classe são mais apropriadas quando você precisa manter estado específico de instância ou quando está modelando objetos do mundo real.

Exemplo de API utilizando o padrão construtor

1. Dentro de `src/controllers`, crie um arquivo chamado `UserController.ts`:

```
typescriptCopy code
import { Request, Response } from 'express';
import { UserService } from '../services/UserService';

export class UserController {
  private userService: UserService;

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

  public getUser(req: Request, res: Response): void {
    const user = this.userService.getUser();
    res.json(user);
  }
}
```

1. Dentro de `src/services`, crie um arquivo chamado `UserService.ts`:

```
typescriptCopy code
import { UserRepository } from '../repositories/UserRepositor
y';

export class UserService {
  private userRepository: UserRepository;

  constructor(userRepository: UserRepository) {
    this.userRepository = userRepository;
  }
```

```
  public getUser(): string {
    return this.userRepository.getUser();
  }
}
```

1. Dentro de `src/repositories`, crie um arquivo chamado `UserRepository.ts`:

```
typescriptCopy code
export class UserRepository {
  public getUser(): string {
    return 'Usuário do Repositório';
  }
}
```

1. No seu arquivo de entrada, por exemplo, `src/index.ts`, configure o Express e utilize a injeção de dependência manualmente:

```
typescriptCopy code
import express from 'express';
import { UserController } from './controllers/UserControlle
r';
import { UserService } from './services/UserService';
import { UserRepository } from './repositories/UserRepositor
y';

const userRepository = new UserRepository();
const userService = new UserService(userRepository);
const userController = new UserController(userService);

const app = express();
app.use(express.json());

app.get('/user', (req, res) => userController.getUser(req, re
```

```
s));

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Servidor ouvindo na porta ${PORT}`);
});
```

Este exemplo ainda segue a arquitetura de camadas, mas a injeção de dependência é feita manualmente, passando as instâncias necessárias diretamente para os construtores. Lembre-se de que o uso de bibliotecas específicas de injeção de dependência, como o Inversify, pode simplificar a gestão de dependências em projetos maiores.

Prático:

- Refatorar API para utilizar o padrão de injeções de dependências
- Testar através da pagina do swagger