Aula 2

Created	@October 28, 2023 11:22 AM
Data	@February 2, 2024
Status	Iniciado
Tipo	Aula

Implementando e testando sua primeira API (Layer Architecture)

Introdução a Arquitetura de Camadas (Layer Architecture)

A arquitetura de camadas é um paradigma de design de software que organiza um sistema em camadas distintas, cada uma com responsabilidades específicas e bem definidas. Cada camada oferece serviços para a camada acima dela e utiliza os serviços da camada abaixo. Essa abordagem promove a modularidade, reusabilidade e a facilidade de manutenção do código. Vamos explorar alguns conceitos fundamentais da arquitetura de camadas:

Princípios Básicos:

1. **Separação de Responsabilidades:**
    - Cada camada possui uma responsabilidade clara e específica no sistema, o que facilita a compreensão e a manutenção do código.
2. **Dependências Direcionadas:**
    - As camadas são organizadas hierarquicamente, com cada camada dependendo apenas das camadas imediatamente abaixo dela. Isso cria uma

direção unidirecional nas dependências.

3. **Encapsulamento:**
    - Cada camada é um componente independente, encapsulando suas funcionalidades internas. A interface externa da camada é usada pelas camadas superiores.
4. **Reusabilidade:**
    - Componentes em cada camada podem ser reutilizados em diferentes partes do sistema ou em sistemas diferentes, proporcionando flexibilidade e eficiência no desenvolvimento.

Principais Camadas em uma Arquitetura de Camadas:

1. **Camada de Apresentação (Presentation Layer):**
    - Responsável por apresentar informações ao usuário e coletar dados de entrada. Inclui interfaces gráficas, APIs e outros componentes de interação com o usuário.
2. **Camada de Lógica de Negócios (Business Logic Layer):**
    - Contém a lógica de negócios da aplicação. Processa dados, executa regras de negócios e coordena a interação entre diferentes partes do sistema.
3. **Camada de Acesso a Dados (Data Access Layer):**
    - Responsável pela comunicação com o banco de dados ou outras fontes de armazenamento de dados. Realiza operações de leitura e escrita, encapsulando as complexidades do acesso aos dados.
4. **Camada de Infraestrutura (Infrastructure Layer):**
    - Fornece suporte para as camadas acima, incluindo serviços como logging, autenticação, segurança e gerenciamento de configurações. Pode também conter componentes de acesso a serviços externos.

Benefícios da Arquitetura de Camadas:

1. **Manutenção Facilitada:**

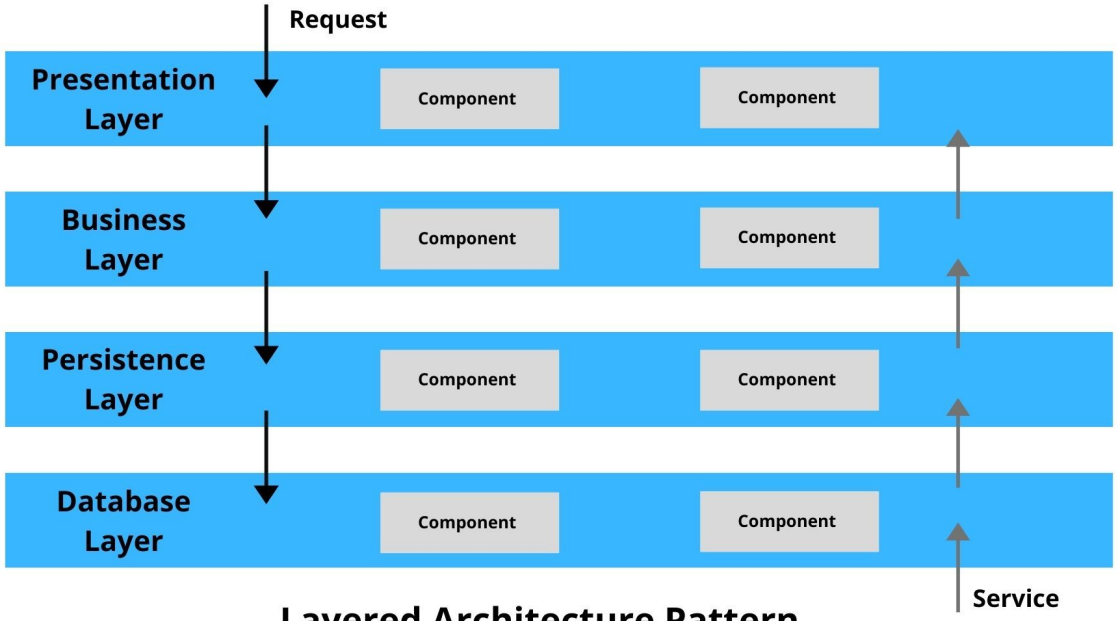
- As alterações em uma camada têm impacto mínimo nas outras, facilitando a manutenção e a evolução do sistema ao longo do tempo.

2. **Reusabilidade:**
   - Componentes em cada camada podem ser reutilizados em diferentes contextos, reduzindo a duplicação de código.
3. **Escalabilidade:**
   - A separação de responsabilidades permite escalar partes específicas do sistema conforme necessário.
4. **Testabilidade:**
   - Cada camada pode ser testada de forma isolada, facilitando a implementação de testes unitários e a detecção de falhas.
5. **Flexibilidade:**
   - A modularidade e a clareza nas responsabilidades facilitam a adaptação do sistema a novos requisitos e mudanças.

Desafios e Considerações:

1. **Overhead de Comunicação:**
   - A comunicação entre camadas pode introduzir um overhead. Portanto, é importante encontrar um equilíbrio entre a separação de responsabilidades e a eficiência da comunicação.
2. **Escolha Adequada de Camadas:**
   - É crucial identificar corretamente as responsabilidades de cada camada para evitar confusão e garantir uma arquitetura eficaz.
3. **Complexidade da Configuração:**
   - Em sistemas grandes, a configuração inicial pode ser complexa devido às dependências entre as camadas.

A arquitetura de camadas é amplamente utilizada em muitos tipos de sistemas, desde aplicações web até sistemas corporativos complexos. Ela proporciona uma abordagem estruturada para o design de software, facilitando a criação de sistemas robustos e flexíveis.



Layered Architecture Pattern

Criando uma API utilizando a arquitetura de camadas

Vamos criar um exemplo de uma aplicação Express.js em TypeScript, utilizando uma arquitetura de camadas (layers) e SQLite como banco de dados. Para isso, você precisará ter o Node.js, o TypeScript e o SQLite instalados. Execute os seguintes comandos para configurar o projeto:

1. **Crie um novo diretório para o projeto:**

```
bashCopy code
mkdir express-ts-sqlite
cd express-ts-sqlite
```

2. **Inicialize o projeto com TypeScript:**

```
bashCopy code
npm init -y
```

```
npm install express typescript ts-node @types/node @types/
express sqlite3
```

3. **Crie um arquivo de configuração TypeScript:**

```
bashCopy code
npx tsc --init
```

4. **Estrutura do Projeto:**

   - Organize o projeto da seguinte maneira:

```
plaintextCopy code
express-ts-sqlite/
├── src/
|   ├── controllers/
|   |   └── UserController.ts
|   ├── models/
|   |   └── User.ts
|   ├── routes/
|   |   └── userRoutes.ts
|   ├── services/
|   |   └── UserService.ts
|   ├── app.ts
|   ├── database.ts
|   └── server.ts
├── package.json
├── tsconfig.json
└── nodemon.json
```

5. **Arquivos de Código:**

   - `src/models/User.ts`:

```
typescriptCopy code
export interface User {
  id: number;
  name: string;
  email: string;
}
```

- `src/database.ts`:

```
typescriptCopy code
import sqlite3 from 'sqlite3';

const db = new sqlite3.Database('./database.db');

db.serialize(() => {
  db.run("CREATE TABLE IF NOT EXISTS users (id INTEGER
PRIMARY KEY, name TEXT, email TEXT)");
});

export default db;
```

- `src/services/UserService.ts`:

```
typescriptCopy code
import db from '../database';
import { User } from '../models/User';

export class UserService {
  static getAllUsers(): Promise<User[]> {
    return new Promise((resolve, reject) => {
      db.all('SELECT * FROM users', (err, rows) => {
        if (err) {
          reject(err);
```

```
        } else {
          resolve(rows);
        }
      });
    });
  }

  static getUserById(id: number): Promise<User | undefi
ned> {
    return new Promise((resolve, reject) => {
      db.get('SELECT * FROM users WHERE id = ?', [id],
(err, row) => {
        if (err) {
          reject(err);
        } else {
          resolve(row);
        }
      });
    });
  }

  static createUser(name: string, email: string): Promi
se<User> {
    return new Promise((resolve, reject) => {
      db.run('INSERT INTO users (name, email) VALUES
(?, ?)', [name, email], function (err) {
        if (err) {
          reject(err);
        } else {
          resolve({ id: this.lastID, name, email });
        }
      });
    });
  }

  static deleteUser(id: number): Promise<void> {
```

```
    return new Promise((resolve, reject) => {
      db.run('DELETE FROM users WHERE id = ?', [id], (e
rr) => {
        if (err) {
          reject(err);
        } else {
          resolve();
        }
      });
    });
  }
}
```

- `src/controllers/UserController.ts`:

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

export class UserController {
  static async getAllUsers(req: Request, res: Respons
e): Promise<void> {
    try {
      const users = await UserService.getAllUsers();
      res.json(users);
    } catch (error) {
      console.error(error);
      res.status(500).json({ error: 'Internal Server Er
ror' });
    }
  }

  static async getUserById(req: Request, res: Respons
e): Promise<void> {
```

```
    const userId = Number(req.params.id);
    try {
      const user = await UserService.getUserById(userI
d);
      if (user) {
        res.json(user);
      } else {
        res.status(404).json({ error: 'User not found'
});
      }
    } catch (error) {
      console.error(error);
      res.status(500).json({ error: 'Internal Server Er
ror' });
    }
  }

  static async createUser(req: Request, res: Response):
Promise<void> {
    const { name, email } = req.body;
    try {
      const newUser = await UserService.createUser(nam
e, email);
      res.status(201).json(newUser);
    } catch (error) {
      console.error(error);
      res.status(500).json({ error: 'Internal Server Er
ror' });
    }
  }

  static async deleteUser(req: Request, res: Response):
Promise<void> {
    const userId = Number(req.params.id);
    try {
      await UserService.deleteUser(userId);
```

```
      res.json({ message: 'User deleted successfully'
});
    } catch (error) {
      console.error(error);
      res.status(500).json({ error: 'Internal Server Er
ror' });
    }
  }
}
```

- `src/routes/userRoutes.ts`:

```
typescriptCopy code
import express from 'express';
import { UserController } from '../controllers/UserCont
roller';

const router = express.Router();

router.get('/', UserController.getAllUsers);
router.get('/:id', UserController.getUserById);
router.post('/', UserController.createUser);
router.delete('/:id', UserController.deleteUser);

export default router;
```

- `src/app.ts`:

```
typescriptCopy code
import express from 'express';
import userRoutes from './routes/userRoutes';

const app = express();
```

```
// Middleware para facilitar a leitura do corpo das req
uisições
app.use(express.json());

// Rotas
app.use('/users', userRoutes);

export default app;
```

- `src/server.ts`:

```
typescriptCopy code
import app from './app';

const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server is running on http://localhos
t:${PORT}`);
});
```

- `nodemon.json`:

```
jsonCopy code
{
  "watch": ["src"],
  "ext": "ts",
  "exec": "ts-node src/server.ts"
}
```

6. **Configuração do Script de Execução:**

- Adicione o seguinte script no seu arquivo `package.json`:

```
jsonCopy code
"scripts": {
  "start": "nodemon"
}
```

7. **Execução do Projeto:**
    - Execute o comando `npm start` para iniciar o servidor.

Esta estrutura organiza o código em camadas, facilitando a manutenção e escalabilidade. O TypeScript oferece a vantagem de adicionar tipagem estática ao JavaScript,

Prático:

- Implementar API (CRUD) Arquitetura de Camadas