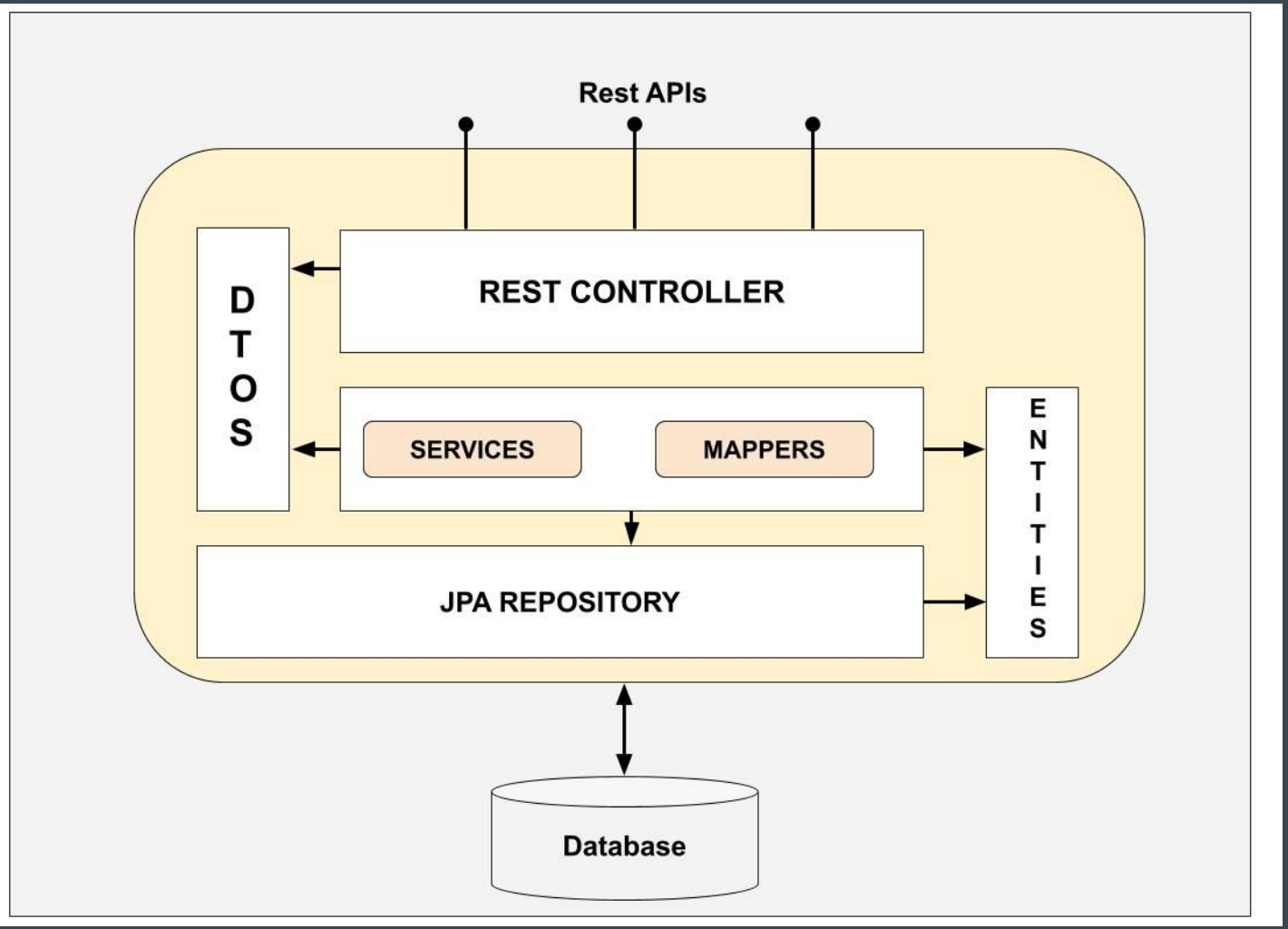
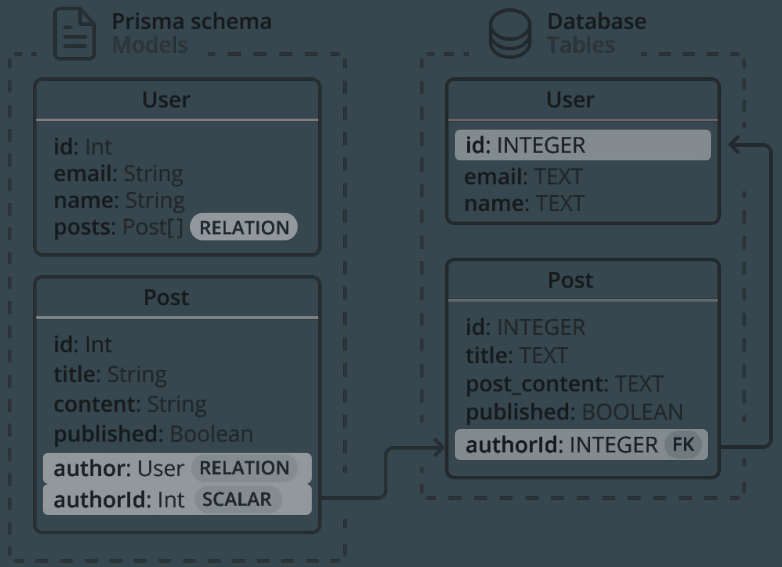
Aula 6 - Arquitetura de Software

José Glauber, UFCG 2024.2


Rest APIs
REST CONTROLLER
DTOS
SERVICES
MAPPERS
ENTITIES
JPA REPOSITORY
Database

Database

- Responsabilidade:
  - Armazenar e recuperar dados.
  - Garantir a integridade e consistência dos dados.
- Ferramentas comuns:
  - Bancos relacionais (MySQL e PostgreSQL)
  - Bancos NoSQL (MongoDB, Redis.)
- Boas práticas:
  - Usar migrations para gerenciar o esquema de banco de dados.
  - Evitar lógica de negócio no banco de dados.



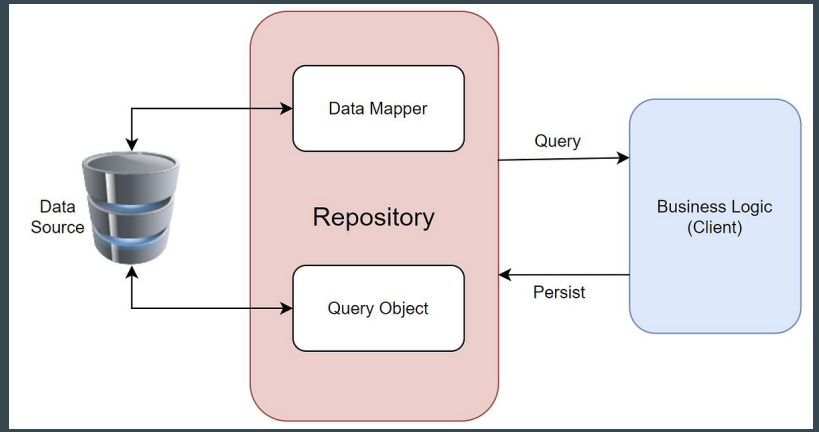
Database

```
// src/config/database.ts

import { Sequelize } from 'sequelize';

const sequelize = new Sequelize({
  database: 'meu_banco',
  username: 'root',
  password: 'senha',
  host: 'localhost',
  dialect: 'mysql',
});

export default sequelize;
```

Por que é importante separar a configuração do banco de dados em um arquivo específico?

Repository

- Responsabilidade:
  - Fornecer uma interface para acessar o banco de dados;
  - Isolar a lógica de acesso a dados.
- Vantagens:
  - Facilita a troca do banco de dados sem afetar as outras camadas;
  - Centraliza operações de CRUD.
- Padrão Repository:
  - Cada entidade do sistema tem um repositório correspondente.



Repository

```ts
// src/repositories/UserRepository.ts
import { User } from '../models/User';
import sequelize from '../config/database';

export class UserRepository {
  private userModel: typeof User;

  constructor() {
    this.userModel = sequelize.model('User'); // Acessa o modelo User
  }

  // Criar um usuário com validação de email único
  async create(userData: Omit<User, 'id'>): Promise<User> {
    // Verifica se o email já está em uso
    const existingUser = await this.userModel.findOne({ where: { email: userData.email } });
    if (existingUser) {
      throw new Error('Email já está em uso.');
    }

    // Cria o usuário
    return this.userModel.create(userData);
  }

  // Buscar todos os usuários ativos
  async findActiveUsers(): Promise<User[]> {
    return this.userModel.findAll({ where: { isActive: true } });
  }

  // Buscar um usuário por ID
  async findById(id: number): Promise<User | null> {
    return this.userModel.findByPk(id);
  }
```

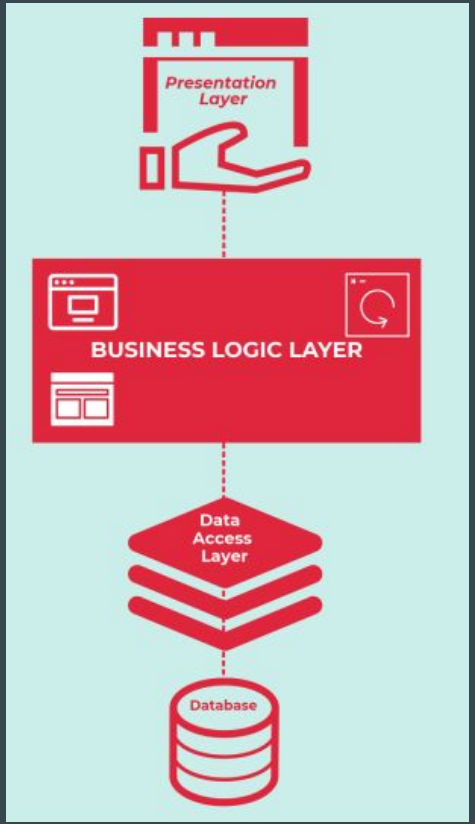
```ts
// Atualizar um usuário
async update(id: number, userData: Partial<User>): Promise<[number, User[]]> {
  return this.userModel.update(userData, { where: { id } });
}

// Desativar um usuário (marcar como inativo)
async deactivateUser(id: number): Promise<[number, User[]]> {
  return this.userModel.update({ isActive: false }, { where: { id } });
}

// Buscar um usuário por email
async findByEmail(email: string): Promise<User | null> {
  return this.userModel.findOne({ where: { email } });
}
```

Pensem em quais métodos excepcionais a API de vocês vai precisar ter...

Services

- Responsabilidade:
  - Contém a lógica de negócios do sistema;
  - Coordena operações entre repositórios e outras camadas.

- Vantagens:
  - Separa a lógica de negócio da lógica de acesso aos dados;
  - Facilita a reutilização de código.



Services

```ts
export class UserService {
  private userRepository: UserRepository;

  constructor() {
    this.userRepository = new UserRepository();
  }

  // Criar um usuário
  async createUser(userData: Omit<User, 'id'>): Promise<User> {
    // Validação adicional (ex: senha forte)
    if (userData.password.length < 6) {
      throw new Error('A senha deve ter pelo menos 6 caracteres.');
    }

    // Cria o usuário usando o repositório
    return this.userRepository.create(userData);
  }

  // Buscar todos os usuários ativos
  async getActiveUsers(): Promise<User[]> {
    return this.userRepository.findActiveUsers();
  }
```

Entra boa parte dos **requisitos funcionais** e validações.

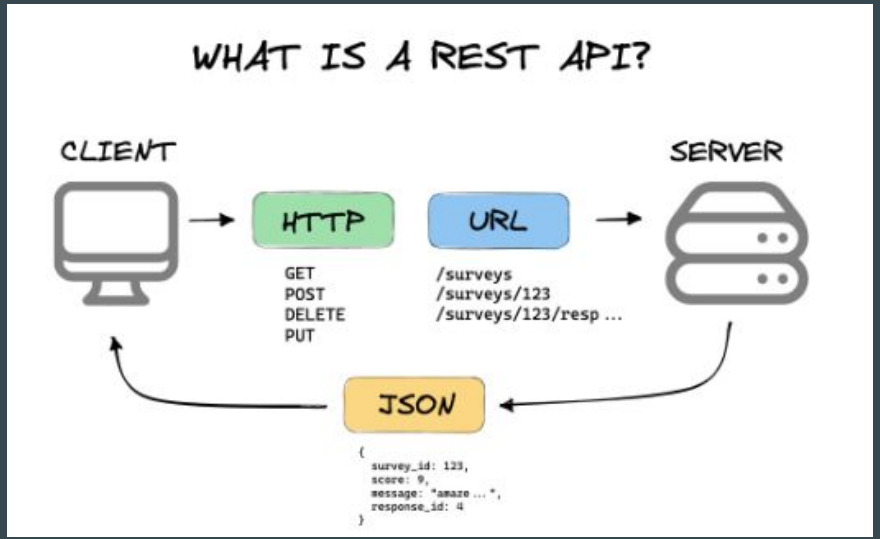
```ts
// Autenticar um usuário (exemplo de lógica de negócio)
async authenticate(email: string, password: string): Promise<User> {
  const user = await this.userRepository.findByEmail(email);
  if (!user) {
    throw new Error('Email ou senha incorretos.');
  }

  // Verifica a senha (em um caso real, use bcrypt ou similar)
  if (user.password !== password) {
    throw new Error('Email ou senha incorretos.');
  }

  return user;
}
```

Por que a camada de **Services** não deve acessar diretamente o banco de dados?

Controllers

- Responsabilidade:
  - Gerencia a interação entre o usuário e o sistema.
  - Recebe requisições HTTP e retorna respostas;

- Vantagens:
  - Separa a lógica de negócio da lógica de apresentação;
  - Facilita a criação de APIs RESTful.



Controllers

```ts
// src/controllers/UserController.ts
import { Request, Response } from 'express';
import { UserService } from '../services/UserService';

export class UserController {
  private userService: UserService;

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

  // Criar um usuário
  async createUser(req: Request, res: Response): Promise<void> {
    try {
      const { name, email, password } = req.body;

      // Verifica se todos os campos foram fornecidos
      if (!name || !email || !password) {
        res.status(400).json({ message: 'Todos os campos são obrigatórios.' });
        return;
      }

      // Cria o usuário
      const newUser = await this.userService.createUser({ name, email, password });
      res.status(201).json(newUser);
    } catch (error) {
      res.status(500).json({ message: error.message });
    }
  }
```

```ts
  // Autenticar um usuário
  async authenticateUser(req: Request, res: Response): Promise<void> {
    try {
      const { email, password } = req.body;

      // Verifica se o email e a senha foram fornecidos
      if (!email || !password) {
        res.status(400).json({ message: 'Email e senha são obrigatórios.' });
        return;
      }

      // Autentica o usuário
      const user = await this.userService.authenticate(email, password);
      res.status(200).json(user);
    } catch (error) {
      if (error.message === 'Email ou senha incorretos.') {
        res.status(401).json({ message: error.message });
      } else {
        res.status(500).json({ message: error.message });
      }
    }
  }
```

Pensem em quais tratamentos os dados que vem na requisição/query precisarão ter à nível de controller...

Perguntas!!

- Qual a diferença entre a camada de Repository e a de Service?
- Por que é importante evitar a lógica de negócio no Controller?
- Como a arquitetura em camadas facilita a manutenção de testes?

Exercício extra... (porém obrigatório)

https://docs.google.com/document/d/1HQzxpYmkO36R3Cd3taT9b5QgLocNVcgrrS_BC67hB1Q/edit?usp=sharing