



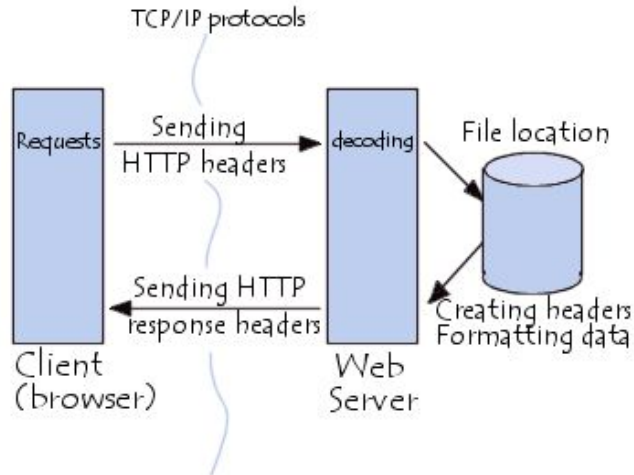
# API REST - Spring

Principais conceitos:

# Protocolo HTTP

O principal protocolo de comunicação na *Web* é o **HTTP**.

Ele funciona como um protocolo de requisição-resposta em um modelo que chamamos de cliente-servidor





# Métodos HTTP

- **GET:** método utilizado para ler e recuperar dados. Requisita uma representação do recurso especificado e retorna essa representação.

Ao enviar uma solicitação GET, o cliente (geralmente um navegador da web) solicita o recurso especificado pelo URL (Uniform Resource Locator) do servidor. O servidor, por sua vez, retorna o recurso solicitado como uma resposta HTTP, geralmente com um código de status 200 (OK) para indicar que a solicitação foi bem-sucedida.



# Métodos HTTP

- **POST:** método utilizado para criar um novo recurso. Envia dados ao servidor.

Ao contrário do método GET, que é usado para solicitar recursos específicos do servidor, o método POST é usado para enviar dados do cliente (geralmente um formulário HTML preenchido pelo usuário) para o servidor para que o servidor possa processá-los e responder com uma ação apropriada.

Ao enviar uma solicitação POST, o cliente inclui os dados a serem enviados no corpo da solicitação HTTP. Esses dados são geralmente codificados no formato de chave-valor (por exemplo, "nome=João&idade=25"). O servidor, por sua vez, processa os dados e pode enviar uma resposta HTTP indicando que a ação foi concluída com sucesso (geralmente com um código de status 200 ou 201) ou que houve um erro (com um código de status 4xx ou 5xx).

O método POST é frequentemente usado em formulários de inscrição, envio de mensagens, envio de dados de pagamento, entre outros casos em que os dados precisam ser enviados ao servidor para que possam ser processados. Ao contrário do método GET, às solicitações POST não são armazenadas em cache pelo navegador e pelos proxies intermediários, o que significa que cada solicitação POST é tratada como uma solicitação separada pelo servidor.



# Métodos HTTP

- **PUT**: cria um novo recurso ou substitui uma representação do recurso de destino com os novos dados. A diferença entre **PUT** e **POST** é que **PUT** é idempotente: ao chamá-lo uma ou várias vezes sucessivamente o efeito é o mesmo, enquanto se chamar o **POST** repetidamente pode ter efeitos adicionais. Por exemplo, se criarmos um produto com **POST**, se a URL definida na API for chamada 20 vezes, 20 produtos serão criados e cada um deles terá um ID diferente. Já com o **PUT**, se você executar a URL definida na API 20 vezes, o resultado tem que ser o mesmo: o mesmo produto atualizado 20 vezes.



# Métodos HTTP

- **DELETE:** exclui o recurso.

O método DELETE é considerado uma operação idempotente, o que significa que uma solicitação DELETE repetida produzirá o mesmo resultado que uma solicitação DELETE única. Se o recurso especificado pelo URL não existir, o servidor retorna um código de status 404 (Not Found) indicando que o recurso não foi encontrado. Se a solicitação DELETE for bem-sucedida, o servidor retorna um código de status 204 (No Content) para indicar que a solicitação foi concluída com sucesso.



## Criando o Projeto Spring:

Primeiro passo é criar um projeto Spring Boot no [Spring Initializr](#).



### Project

- ☒ Gradle - Groovy
- ☐ Gradle - Kotlin
- ☐ Maven

### Language

- ☒ Java ☐ Kotlin
- ☐ Groovy

### Spring Boot

- ☐ 3.1.0 (SNAPSHOT) ☐ 3.1.0 (M1)
- ☐ 3.0.5 (SNAPSHOT) ☒ 3.0.4 ☐ 2.7.10 (SNAPSHOT)
- ☐ 2.7.9

### Project Metadata

Group: org.springframework.samples

### Dependencies

ADD DEPENDENCIES... CTRL + B

#### Spring Boot DevTools

DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

#### Lombok

DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

#### Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.


GENERATE CTRL + ↵

EXPLORE CTRL + SPACE

SHARE...







Como dependências, vamos selecionar o **Spring Web (Spring MVC)** e **Lombok**. O **Spring MVC** é um framework que ajuda no desenvolvimento de aplicações web no padrão **MVC (model-view-controller)**. O Lombok é uma biblioteca Java focada em produtividade e redução de código por meio de anotações que ensinam o compilador a criar e manipular código Java. Ou seja, você não vai mais precisar escrever métodos *getter*, *setter*, *equals*, *hashCode*, construtores de classe, etc.



## **Model: Classes de Domínio e Mapeamento de Entidades:**



# Introdução às Classes de Domínio

- Classes de domínio representam os objetos do seu negócio.
- São responsáveis por conter a lógica de negócio e as regras associadas.
- Na arquitetura da aplicação, atuam como a camada de representação dos dados.



## Entidades e Mapeamento Objeto-Relacional:

- Mapeamento objeto-relacional (ORM) é a técnica de mapear objetos de software para tabelas de banco de dados.
- ORM permite abstrair a complexidade das operações do banco de dados e facilita a manutenção do código.
- Hibernate é um dos provedores de ORM mais populares e é integrado ao Spring Boot.



## Anotações para Mapeamento de Entidades

- @Entity: marca a classe como uma entidade mapeada no banco de dados.
- @Table: especifica o nome da tabela correspondente no banco de dados.
- @Id: indica o campo que representa a chave primária da entidade.
- @GeneratedValue: define a estratégia de geração automática de valores para a chave primária.
- @Column: mapeia atributos da classe para colunas no banco de dados.
- @Temporal: define o tipo de data e hora a ser armazenado no banco de dados (DATE, TIME ou TIMESTAMP).
- @Enumerated: mapeia atributos do tipo Enum para o banco de dados.



# Relacionamentos entre Entidades

- Relacionamentos são associações entre entidades no banco de dados.
- Tipos de relacionamentos:

Um para um (One-to-One): uma entidade está associada a exatamente uma outra entidade.

Um para muitos (One-to-Many): uma entidade está associada a várias outras entidades.

Muitos para um (Many-to-One): várias entidades estão associadas a uma única entidade.

Muitos para muitos (Many-to-Many): várias entidades estão associadas a várias outras entidades.




## Anotações para Relacionamentos

- @OneToOne: define um relacionamento um para um entre entidades.
- @OneToMany: define um relacionamento um para muitos entre entidades.
- @ManyToOne: define um relacionamento muitos para um entre entidades.
- @ManyToMany: define um relacionamento muitos para muitos entre entidades.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "users")
public class User {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    no usages
    @Column(nullable = false)
    private String firstName;
    no usages
    @Column(nullable = false)
    private String lastName;
    no usages
    @Column(nullable = false, unique = true)
    private String email;
    no usages
    @Column(nullable = false)
    private String password;
}
```





Nesse exemplo, estamos usando as anotações do **Lombok** para gerar automaticamente os métodos *getters*, *setters*, *equals*, *hashCode* e *toString*, bem como o construtor padrão e o construtor com argumentos.

Além disso, estamos usando as anotações do **JPA** para mapear essa classe para uma tabela no banco de dados. A anotação **@Entity** indica que essa classe é uma entidade e será mapeada para uma tabela. A anotação **@Table** é usada para especificar o nome da tabela no banco de dados.

Também estamos usando a anotação **@Id** para indicar que o atributo `id` é a chave primária da tabela e a anotação **@GeneratedValue** para indicar que o valor desse atributo será gerado automaticamente pelo banco de dados.

As anotações **@Column** são usadas para especificar as colunas da tabela e suas propriedades, como nome, se são obrigatórias ou únicas, etc.



## Repository: Interação com o Banco de Dados



# Introdução à Camada Repository

- A camada Repository é responsável por interagir com o banco de dados e gerenciar as operações de persistência de dados.
- Atua como uma ponte entre a camada de negócios (Services) e a camada de acesso a dados (Model).
- Isola a lógica de acesso a dados, facilitando a manutenção e o teste do aplicativo.
- No Spring Boot, os repositórios são geralmente implementados usando o Spring Data JPA, que fornece uma abstração de alto nível para operações de banco de dados.



# Spring Data JPA

- Spring Data JPA é um subprojeto do Spring Data que simplifica a implementação de repositórios para aplicações Java.
- Fornece uma abstração de alto nível para operações comuns de banco de dados, como criar, ler, atualizar e excluir (CRUD).
- Baseado no Java Persistence API (JPA), permite a integração com diversos provedores de ORM (Object-Relational Mapping), como Hibernate.
- Reduz a quantidade de código e a complexidade, permitindo aos desenvolvedores focar na lógica de negócios.



## Criando e Configurando Repositórios

- Interface JpaRepository: herde desta interface para criar repositórios Spring Data JPA.
  - JpaRepository<Entity, ID>: os parâmetros de tipo representam a entidade e o tipo de ID.
  - Fornece métodos padrão para operações CRUD e consultas simples.
- Criando um repositório para uma entidade:
  - Crie uma interface que estenda JpaRepository.
  - Defina a entidade e o tipo de ID como parâmetros de tipo.
- Configurando transações e isolamento:
  - O Spring Data JPA gerencia automaticamente as transações.
  - Use a anotação @Transactional para personalizar o comportamento das transações.



# Operações Básicas do Spring Data JPA

- Métodos padrão do JpaRepository:
  - `save(entity)`: salva ou atualiza uma entidade no banco de dados.
  - `findById(id)`: busca uma entidade pelo ID.
  - `findAll()`: retorna todas as entidades.
  - `count()`: retorna a quantidade de entidades.
  - `deleteById(id)`: exclui uma entidade pelo ID.
  - `existsById(id)`: verifica se uma entidade existe pelo ID.
- Consultas derivadas a partir de nomes de métodos:
  - O Spring Data JPA gera consultas automaticamente com base nos nomes dos métodos de repositório.
  - Exemplos: `findByName`, `findByCategory`, `findByPriceGreaterThan`.
- Consultas personalizadas com `@Query`:
  - Use a anotação `@Query` para criar consultas personalizadas usando JPQL ou SQL nativo.
  - Exemplo: `@Query("SELECT p FROM Produto p WHERE p.preco >= :precoMin")`



## Exemplo Prático - Criando Repositórios

- Exemplo - Repositório Categoria :

```
import org.springframework.data.jpa.repository.JpaRepository;
import com.example.domain.Categoria;

public interface CategoriaRepository extends JpaRepository<Categoria, Long> {
}
```



## **Service: Lógica de Negócios e Coordenação**





# Introdução à Camada Service

- A camada Service é responsável pela lógica de negócios e coordenação entre as camadas Repository e Controller.
- Atua como uma ponte entre a camada de apresentação (Controller) e a camada de acesso a dados (Repository).
- Encapsula regras de negócio, validações e operações comuns para garantir a consistência dos dados e a integridade da aplicação.
- No Spring Boot, as classes de serviço são implementadas como beans gerenciados pelo Spring e podem ser facilmente injetadas em outras classes.



# Criando e Organizando Serviços

- Como criar uma classe de serviço:
  - Defina uma classe com a anotação `@Service`.
  - Essa anotação indica que a classe é um bean gerenciado pelo Spring e é parte da camada de serviço.
- Injetando repositórios em serviços:
  - Use a anotação `@Autowired` para injetar automaticamente os repositórios necessários.
  - O Spring cuida da injeção de dependências e garante que os repositórios estejam disponíveis.
- Estruturação e organização de classes de serviço:
  - Separe os serviços por domínio ou funcionalidade.
  - Mantenha as classes de serviço focadas e com responsabilidades bem definidas.



# Implementando a Lógica de Negócios

- Regras de negócio e validações:
  - A camada Service é responsável por implementar as regras de negócio e validações específicas do domínio.
  - Isso garante que os dados permaneçam consistentes e a aplicação funcione conforme o esperado.
  - Por exemplo, verifique se o produto está disponível em estoque antes de criar um pedido ou valide se as informações do usuário estão completas antes de salvar no banco de dados.
- Manipulação de exceções e erros:
  - A camada Service também deve lidar com exceções e erros que podem ocorrer durante a execução das operações de negócio.
  - Crie exceções personalizadas para casos específicos do domínio, como "ProdutoIndisponivelException" ou "UsuarioInvalidoException". Isso ajuda a identificar e tratar erros de forma mais clara e específica.
  - Capture e trate exceções lançadas pelas camadas Repository e outras classes auxiliares, garantindo que o controlador receba apenas informações relevantes e úteis.

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
import com.example.domain.Categoria;  
import com.example.repository.CategoriaRepository;
```

```
@Service
```

```
public class CategoriaService {
```

```
    @Autowired
```

```
    private CategoriaRepository categoriaRepository;
```

```
    // Implemente os métodos para gerenciar as operações de negócio relacionadas a
```

```
}
```



**Controller: Manipulação de requisições e  
comunicação com o cliente**



# Introdução à Camada Controller

- O Controller é a camada responsável por manipular requisições HTTP recebidas pelo servidor.
- Ele é responsável por interpretar as informações contidas na requisição e determinar qual ação deve ser tomada.
- O Controller também é responsável por enviar uma resposta HTTP adequada de volta ao cliente.
- Em resumo, a camada Controller é a responsável por orquestrar a comunicação entre o cliente e a aplicação.



## Criando e Organizando Controladores


- Para criar um controlador no Spring Boot, é necessário criar uma classe Java com a anotação `@RestController`. Essa anotação indica que a classe é um controlador que responderá a requisições HTTP.
- O método dessa classe deve ser anotado com a anotação correspondente ao método HTTP desejado (por exemplo, `@GetMapping` para GET, `@PostMapping` para POST, `@PutMapping` para PUT e `@DeleteMapping` para DELETE). Essa anotação indica qual método HTTP deve ser mapeado para esse método.
- A estruturação de classes de controlador pode ser feita por domínio ou por funcionalidade. Organizando por domínio, as classes de controlador são agrupadas por entidades de domínio (por exemplo, um controlador para a entidade Produto e outro para a entidade Categoria). Organizando por funcionalidade, as classes de controlador são agrupadas por funcionalidades da aplicação (por exemplo, um controlador para as operações de autenticação e outro para as operações de pagamento).



## Trabalhando com Dados de Requisição e Resposta

- O Controller é responsável por manipular dados de entrada e saída da requisição HTTP. Ele recebe os dados da requisição e envia uma resposta adequada ao cliente.
- Os parâmetros de entrada podem ser obtidos pelo Controller por meio de anotações como `@RequestParam`, `@PathVariable` e `@RequestBody`. A escolha da anotação a ser usada depende da forma como a informação é passada na requisição. Por exemplo, a anotação `@RequestParam` é usada para receber parâmetros passados na URL, enquanto a anotação `@RequestBody` é usada para receber dados em formato JSON ou XML.
- O Controller pode enviar uma resposta HTTP adequada utilizando um objeto `ResponseEntity` e o código HTTP apropriado (por exemplo, `HttpStatus.OK` para indicar uma resposta bem-sucedida). O objeto `ResponseEntity` permite ao Controller enviar uma resposta personalizada contendo o corpo da resposta e o código de status HTTP.



- 
- A validação de entrada pode ser realizada utilizando anotações de validação, como @NotNull, @Size, @Email, entre outras. Essas anotações garantem que os dados recebidos na requisição estejam no formato correto e dentro das expectativas da aplicação.
  - A manipulação de erros pode ser tratada por meio de exceções. O Controller pode lançar exceções personalizadas para lidar com erros específicos da aplicação. O Spring Boot possui um mecanismo de tratamento de exceções que permite ao desenvolvedor personalizar o comportamento da aplicação em caso de erros.