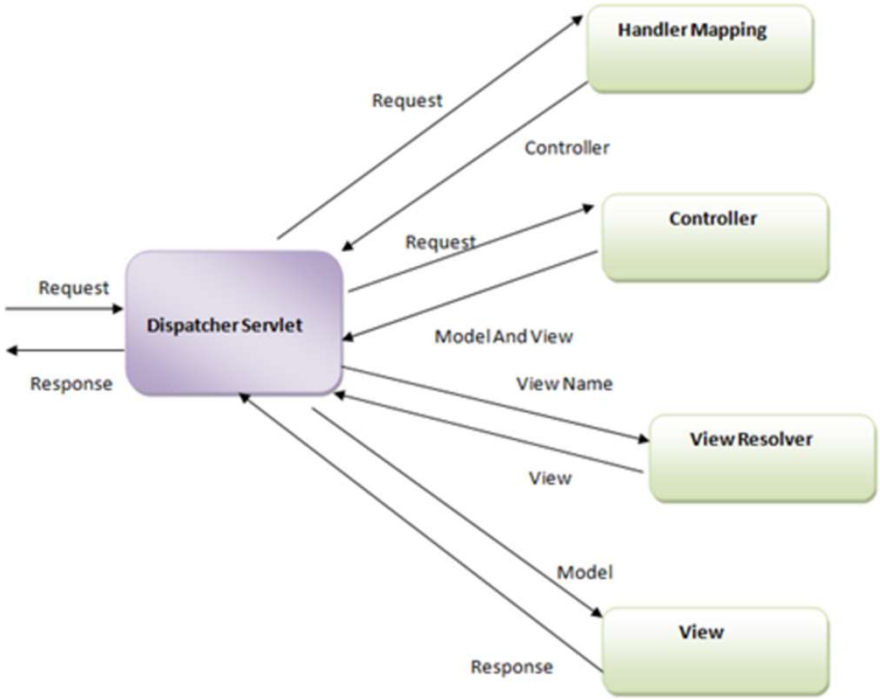
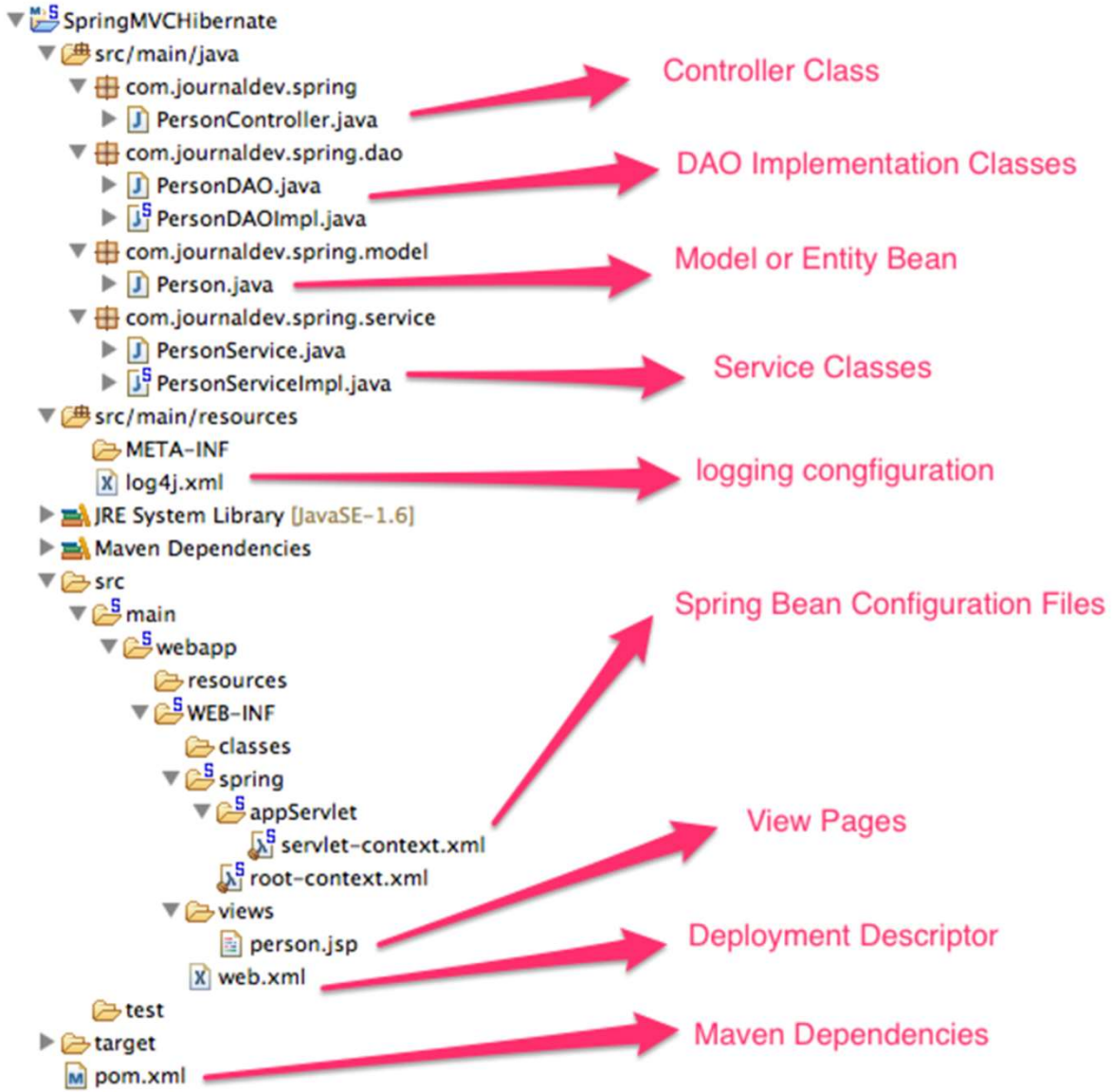
TÓPICOS EM SPRING MVC

Estrutura de uma Aplicação padrão SringMVC



Estrutura de uma Aplicação padrão SringMVC



BEANS

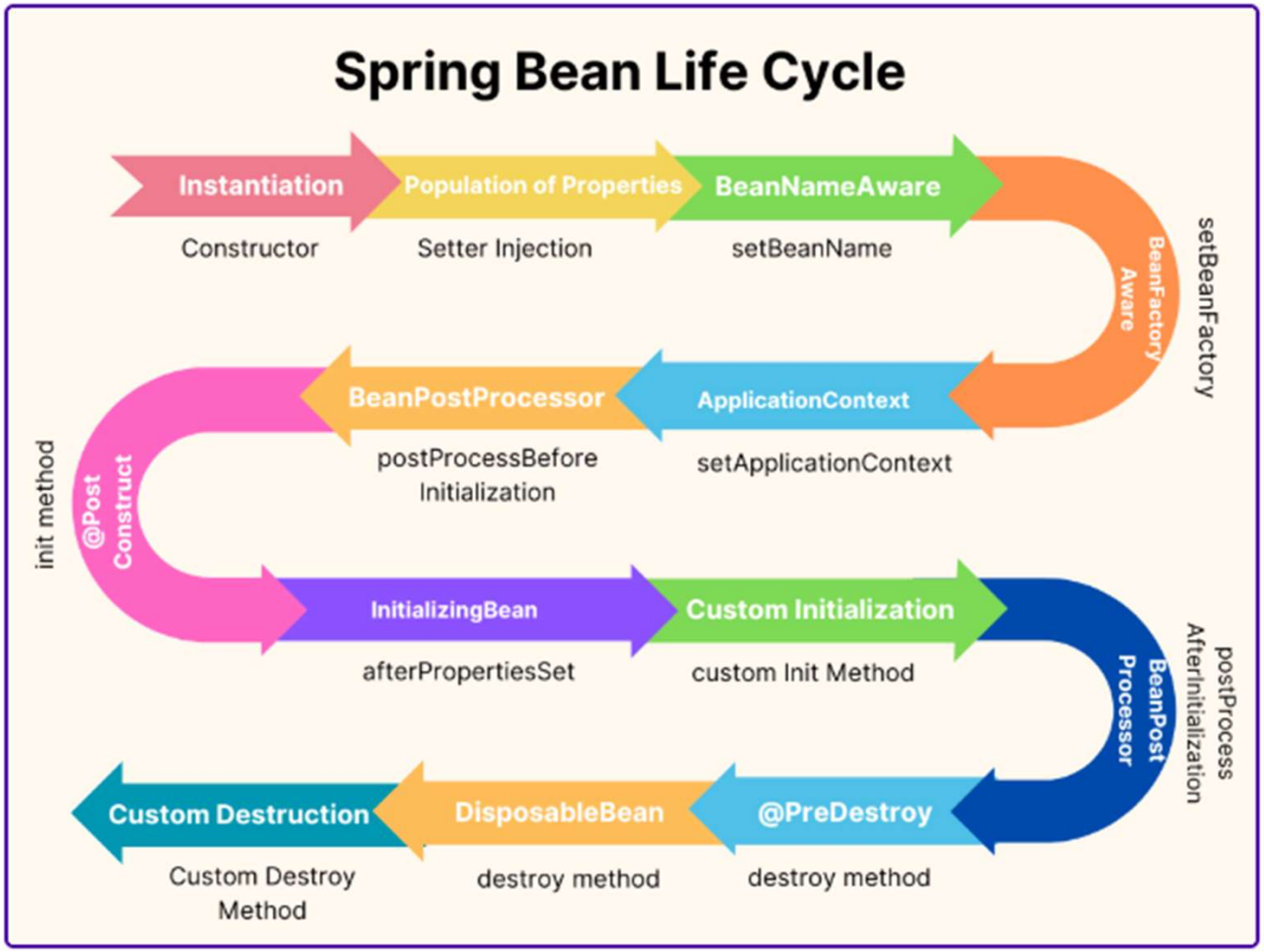
O que é um Bean no Contexto do Spring MVC

Um bean no Spring MVC, dentro do contexto do JSVS (JavaServer Virtual Machine), é um objeto gerenciado pelo framework Spring. Ele é instanciado, configurado e mantido pelo container Spring, que se encarrega de seu ciclo de vida completo.

BEANS - Configuração

Configurar beans no Spring MVC é essencial para definir os componentes da sua aplicação que serão gerenciados pelo framework.

BEANS - Life Cycle



BEANS - Life Cycle

1. Instantiation: Setting the Foundation
2. Population of Properties: Filling the Gaps
3. BeanNameAware: Giving Identity
4. BeanFactoryAware and ApplicationContextAware: Embracing Context
5. BeanPostProcessor: Adding Magic
6. @PostConstruct: Customizing Bean Initialization
7. InitializingBean: Preparing for Action
8. Custom Initialization: Tailored Setup
9. @PreDestroy: Preparing for Cleanup
10. DisposableBean: Bidding Farewell
11. Custom Destruction: Parting Moments

BEANS - Configuração

Existem diversas formas de realizar essa configuração, cada uma com suas particularidades e melhor utilizada em diferentes cenários.

XML: Ideal para configurações complexas e quando você precisa de um alto grau de customização.

Anotações: Ótimo para projetos menores e médios, onde a simplicidade é importante.

JavaConfig: Recomendado para grandes projetos e quando você precisa de um controle total sobre a configuração.

BEANS - Configuração

XML:

•**Tradicional:** Essa é uma das formas mais antigas, onde você define os beans em um arquivo XML.

Exemplo:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="myBean" class="com.example.MyBean">
    <property name="property1" value="value1" />
  </bean>

</beans>
```

Vantagens: Flexível, visual, bom para configurações complexas.

Desvantagens: Verborrágico, menos intuitivo para desenvolvedores Java.

•**Desvantagens:** Menos flexível para configurações complexas.

BEANS - Configuração

Anotações:

•**Moderna:** Utilizando anotações, você marca as classes que serão beans diretamente no código.

Exemplo:

```
@Component
public class MyBean {
   // ...
}
```

Vantagens: Mais limpa, integrada ao código, fácil de entender**.**

Desvantagens: Menos flexível para configurações complexas**.**

•**Desvantagens:** Menos flexível para configurações complexas.

BEANS - Configuração

JavaConfig:

•**Pura Java:** Utilizando classes Java, você define a configuração dos **beans** de forma programática.

Exemplo:

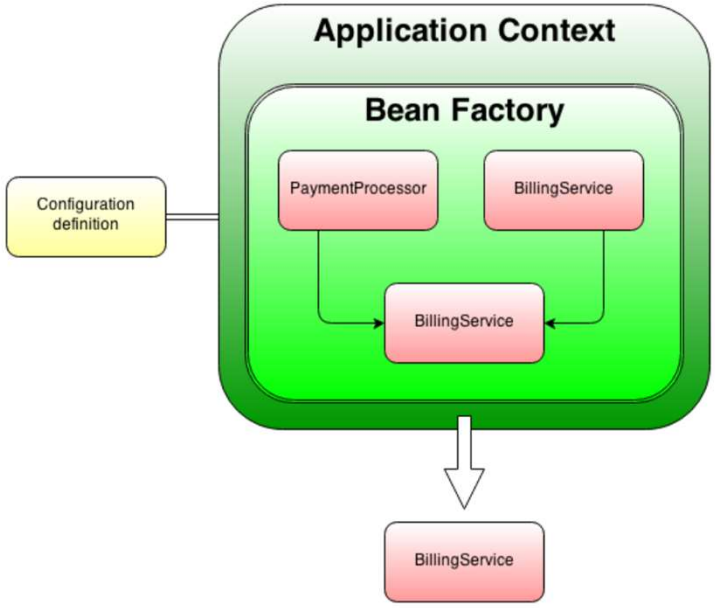
```
@Configuration
public class AppConfig {
  @Bean
  public MyBean myBean() {
    MyBean bean = new MyBean();
    bean.setProperty1("value1");
    return bean;
  }
}
```

Vantagens: Totalmente orientada a objetos, fácil de testar, ideal para grandes aplicações**.**

Desvantagens: Requer mais código do que as anotações**.**

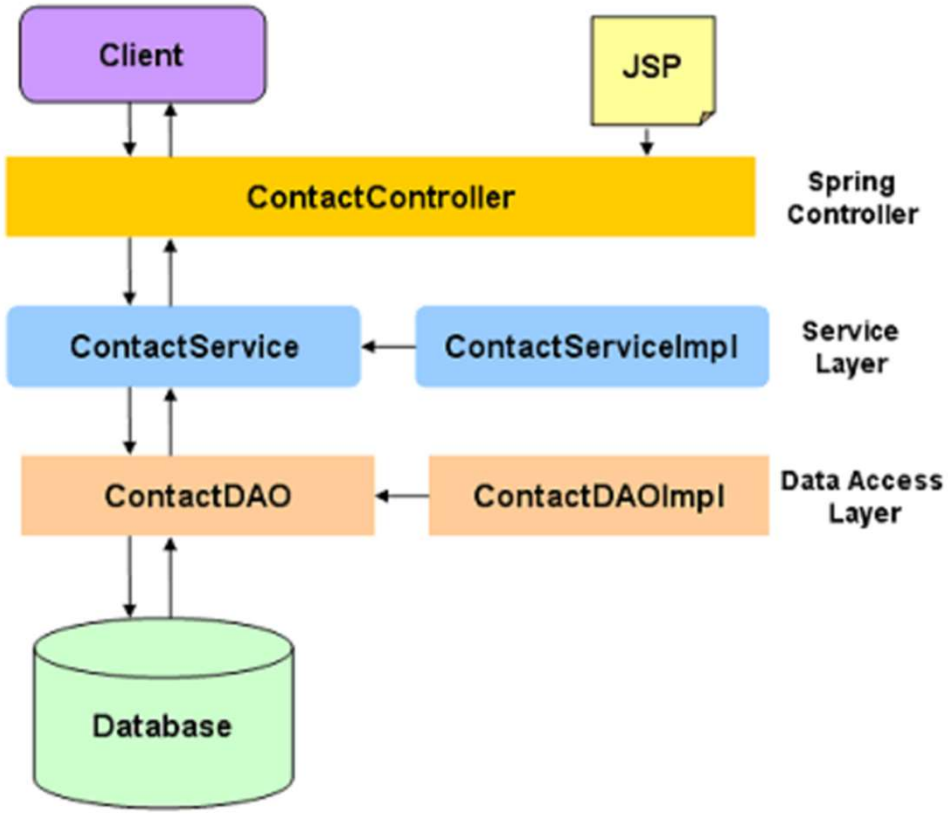
Application Context

O Application Context no Spring MVC é o coração da sua aplicação. Ele serve como um contêiner que gerencia todos os beans (objetos) da sua aplicação, desde os mais simples até os mais complexos. Pense nele como um diretório central onde todos os componentes são registrados e podem ser facilmente acessados



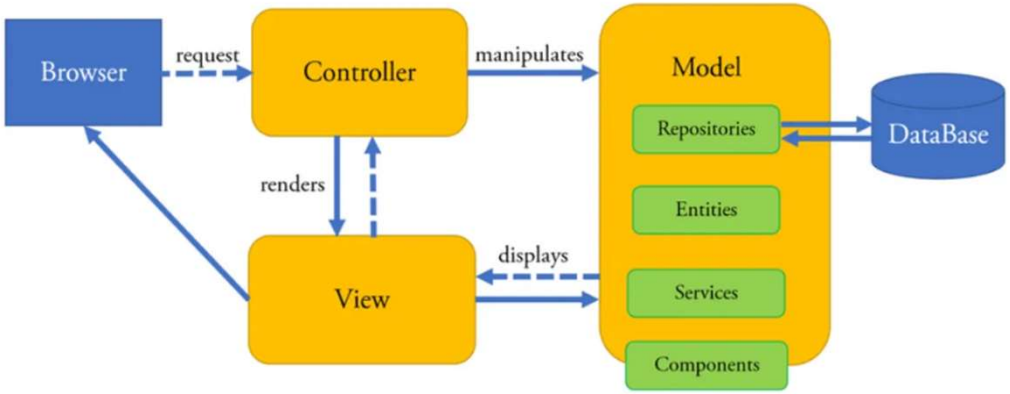
DAO - Data Access Object

Um DAO é uma classe responsável por encapsular todas as operações de acesso a um banco de dados

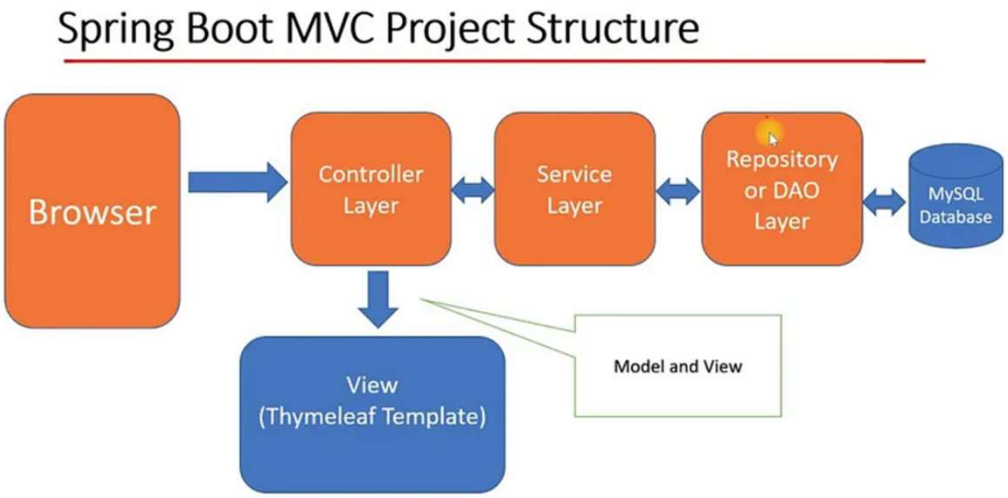


REPOSITORY - É responsável por fornecer operações de acesso a dados na entidade.

Como salvar, atualizar, recuperar e excluir dados.



ESTRUTURA


Spring Boot MVC Project Structure
Browser
Controller Layer
Service Layer
Repository or DAO Layer
MySQL Database
View (Thymeleaf Template)
Model and View

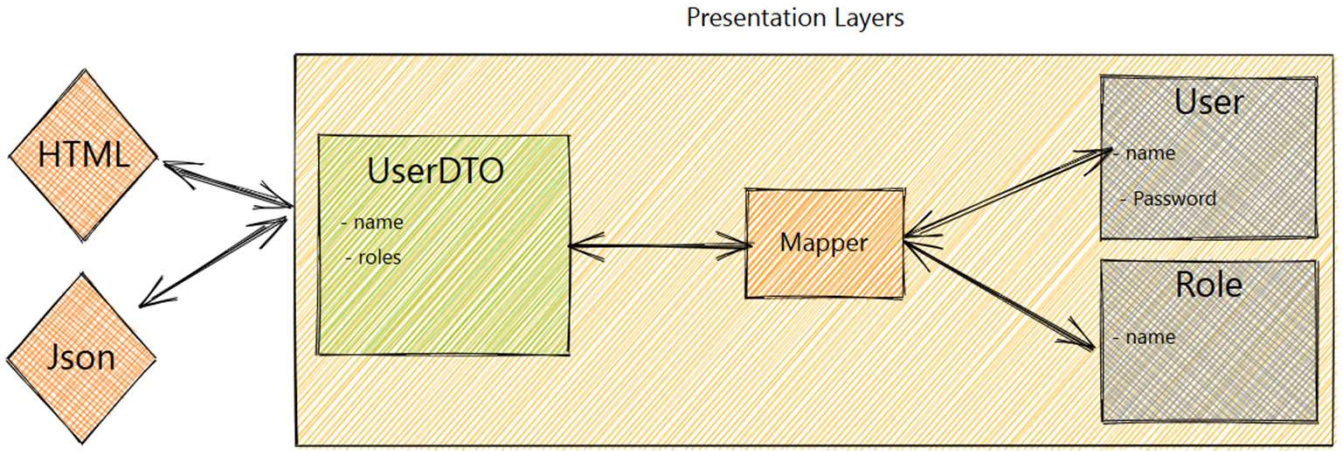
DAO vs REPOSITORY

DAO é uma escolha mais tradicional e oferece um controle mais granular sobre o acesso aos dados.

Repository é uma escolha mais moderna e alinhada com os princípios de DDD, oferecendo uma interface mais intuitiva e aproveitando as funcionalidades avançadas dos ORMs.

DTO - Data Transfer Object

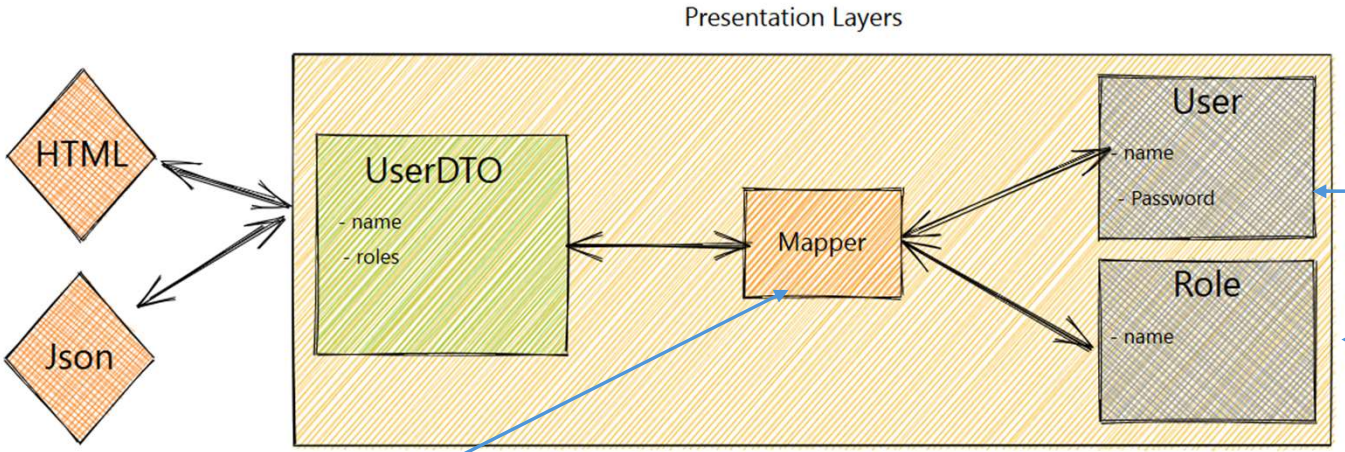
Funciona como um contêiner para transportar dados entre diferentes camadas de uma aplicação



Benefícios do uso de DTOs:

- **Melhora a segurança:** Evita a exposição de dados sensíveis.
- **Aumenta a performance:** Reduz a quantidade de dados transferidos.
- **Aumenta a flexibilidade:** Permite adaptar os dados para diferentes cenários.
- **Melhora a organização do código:** Separa as responsabilidades das camadas da aplicação.

DTO - Data Transfer Object




```java
@Component
class Mapper {
    public UserDTO toDto(User user) {
        String name = user.getName();
        List<String> roles = user
          .getRoles()
          .stream()
          .map(Role::getName)
          .collect(toList());

        return new UserDTO(name, roles);
    }

    public User toUser(UserCreationDTO userDTO) {
        return new User(userDTO.getName(), userDTO.getPassword(), new ArrayList<>());
    }
}
```

DTO - Data Transfer Object

```java
public class User {

    private String id;
    private String name;
    private String password;
    private List<Role> roles;

    public User(String name, String password, List<Role> roles) {
        this.name = Objects.requireNonNull(name);
        this.password = this.encrypt(password);
        this.roles = Objects.requireNonNull(roles);
    }

    // Getters and Setters

   String encrypt(String password) {
       // encryption logic
   }
}
```

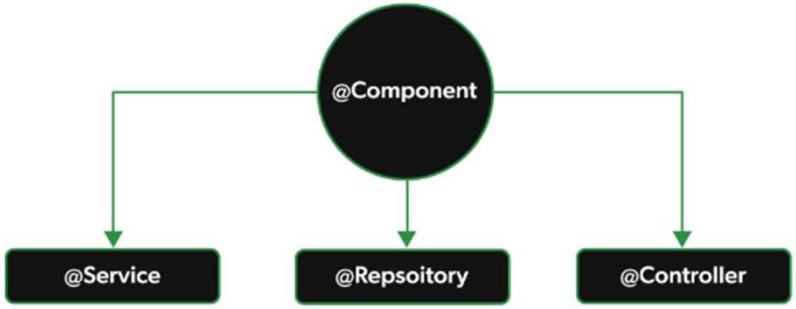
```java
public class Role {

    private String id;
    private String name;

    // Constructors, getters and setters
}
```

DTO

```java
public class UserDTO {
    private String name;
    private List<String> roles;

    // standard getters and setters
}
```

ANNOTATIONS

@Component
Usado para marcar uma classe como um componente, tornando-a elegível para detecção automática e registro como bean pelo mecanismo de verificação de componentes do Spring.



@Service
É uma especialização de @Component.
É usado para marcar a classe como provedora de serviços.

@Repository
Também é uma especialização de @Component.
É usado para marcar uma classe/interface como DAO (Data Access Object) .

@Controller
Também é uma especialização de @Component.
É usado para marcar uma classe como manipulador de solicitações da web.

ANNOTATIONS

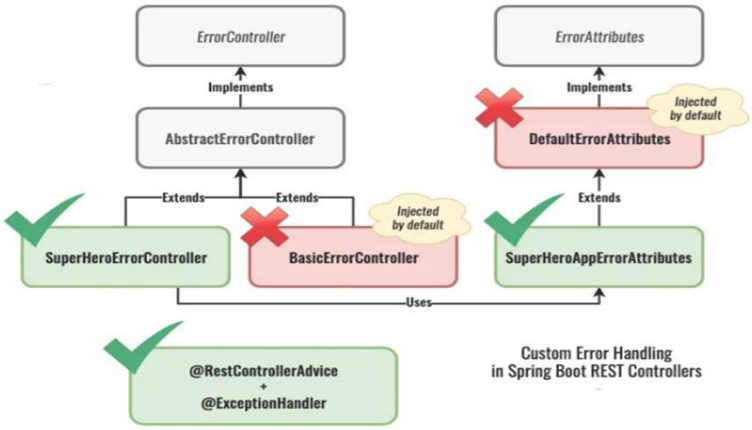
@Configuration
Esta anotação do ***Spring*** marca uma classe como uma fonte de definições de ***bean*** e é uma das anotações essenciais se você estiver usando a configuração baseada em Java.

@ComponentScan
Esta anotação instrui o ***Spring*** a varrer o pacote para todas as classes ***@Configuration***.

@Bean
Esta anotação do ***Spring Core*** indica que um método produz um ***bean*** a ser gerenciado pelo contêiner do ***Spring***.
Basicamente você está dizendo para ***Spring*** que quer criar uma classe de objeto e deixar ela disponível para outras classes utilizarem ele como dependência.

@Autowired
Para referenciar uma classe em outro contexto (obter a instância).
Faz a injeção de dependência do ***Spring.***

@ControllerAdvice
Serve para gerenciar exceções em uma aplicação.



ANNOTATIONS

@Required
Especifica uma dependência obrigatória.

@Lazy
Esta anotação da estrutura ***Spring*** faz com que **@Bean** ou **@Component** sejam inicializados sob demanda.

@Qualifier
Você pode usar esta anotação em um campo ou parâmetro como um qualificador para ***beans*** candidatos usados com o ***@Autowire*** .
É usado principalmente para resolver conflitos de injeção de dependência quando existem vários ***beans*** do mesmo tipo.

@Value
Esta anotação do ***Spring Core*** indica uma expressão de valor padrão para o campo .

```
@Value("NA")
private String name
```

ANNOTATIONS

@initBinder

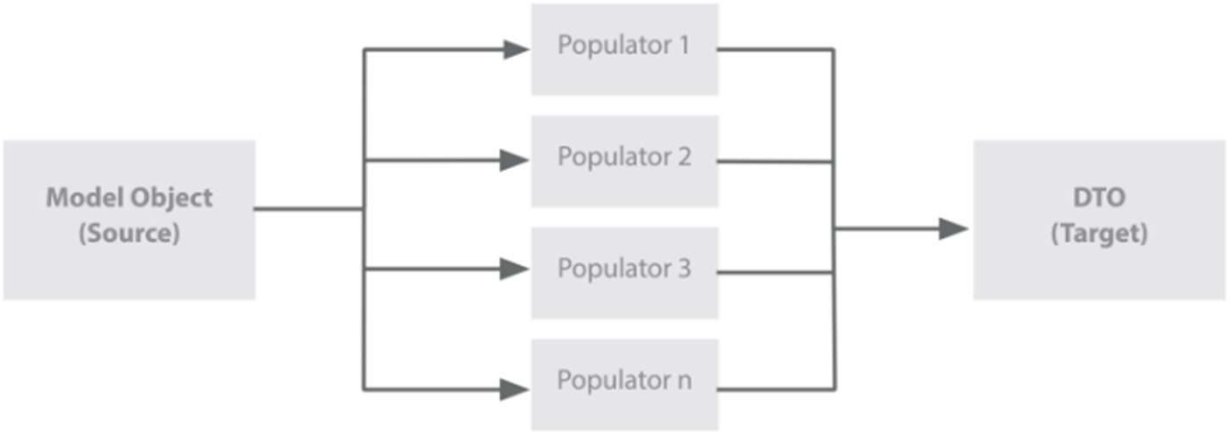
Ele permite editar, converter e formatar dados durante o processo de tratamento de solicitações.

```
@Controller
public class MyController {

   @InitBinder
   public void initBinder(WebDataBinder binder) {
      SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
     binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));
   }

   // ... other controller methods
}
```

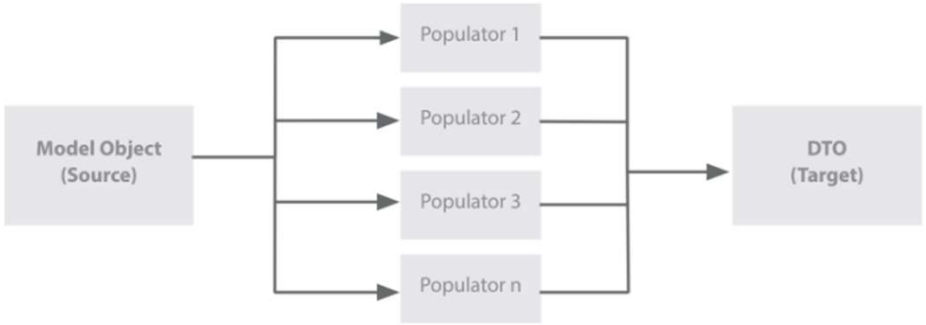
POPULATORS

São ferramentas poderosas para personalizar o processo de mapeamento(biding) de dados no Spring MVC.
Eles são especialmente úteis quando precisamos mapear dados de um formulário HTML.

Os populators são invocados durante o processo de binding de dados, ou seja, quando os dados de uma requisição HTTP são convertidos e atribuídos a objetos Java.



POPULATORS



Model

```java
@Table(name = "estudante")
public class Estudante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "nome")
    private String nome;

    @Column(name = "email")
    private String email;

    @OneToMany(mappedBy = "estudante")
    private List<Inscricao> inscricoes;
    public Long getId() {
        return id;
    }
```

Populator

```java
// Converter Estudante para EstudanteDTO
public static EstudanteDTO convertToEstudanteDTO(Estudante estudante) {
    EstudanteDTO estudanteDTO = new EstudanteDTO(estudante);
    estudanteDTO.setId(estudante.getId());
    estudanteDTO.setNome(estudante.getNome());
    estudanteDTO.setEmail(estudante.getEmail());
    return estudanteDTO;
}

// Converter Curso para CursoDTO
public static CursoDTO convertToCursoDTO(Curso curso) {
    CursoDTO cursoDTO = new CursoDTO(curso);
    cursoDTO.setId(curso.getId());
    cursoDTO.setNome(curso.getNome());
    cursoDTO.setDescricao(curso.getDescricao());
    return cursoDTO;
}
```

DTO

```java
public class EstudanteDTO {
    private Long id;
    private String nome;
    private String email;
```

POPULATORS

Service

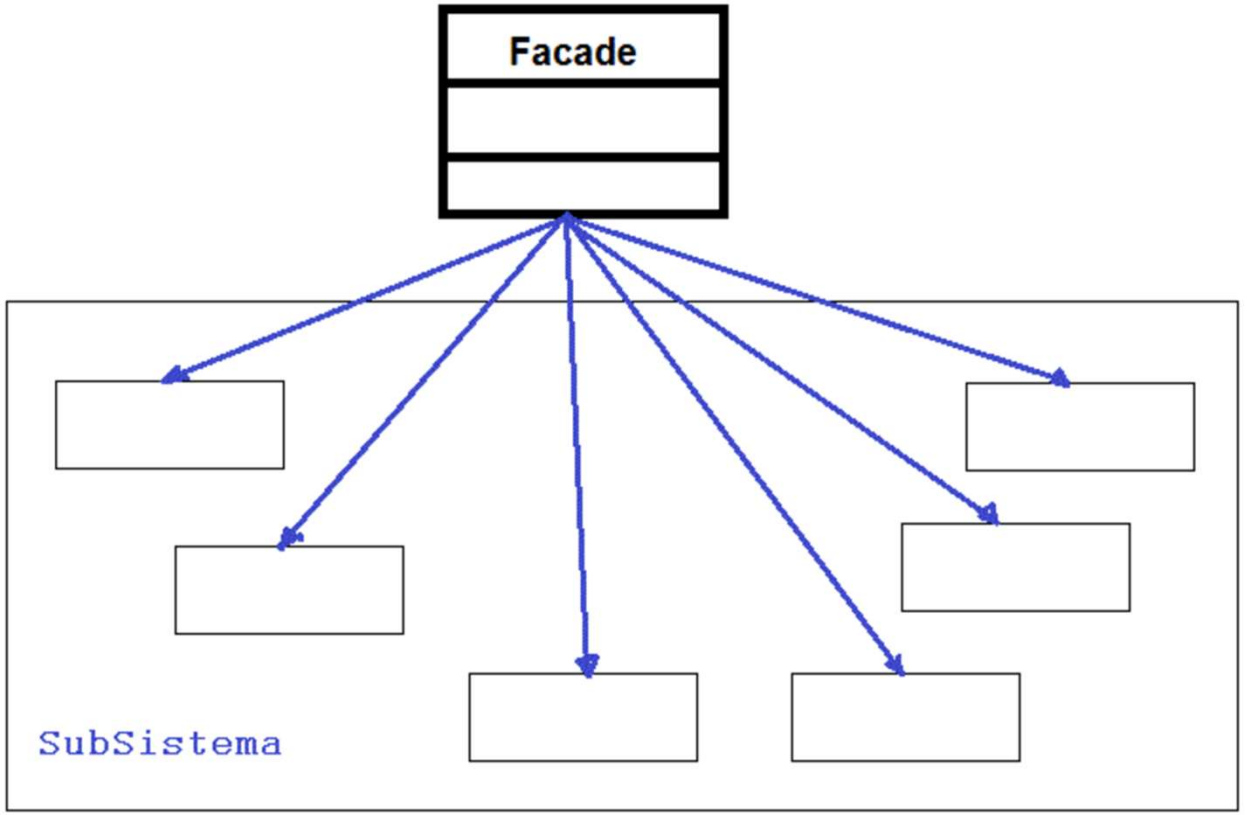
```java
@Table(name = "estudante")
public class Estudante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "nome")
    private String nome;

    @Column(name = "email")
    private String email;

    @OneToMany(mappedBy = "estudante")
    private List<Inscricao> inscricoes;
    public Long getId() {
        return id;
    }
```

Controller

```java
    // Endpoint para listar todos os cursos em que um aluno está inscrito
    @GetMapping("/{estudanteId}/cursos")
    public ResponseEntity<List<CursoDTO>> listarCursosDoEstudante(@PathVariable Long estudanteId) {
        List<CursoDTO> cursosDTO = inscricaoService.listarCursosPorEstudante(estudanteId);
        return ResponseEntity.ok(cursosDTO);
    }
```

FACADE

O Padrão de projeto Facade (ou Fachada) é um padrão de design de software usado comumente com programação orientada a objetos.
Este nome é uma analogia para uma fachada arquitetural.
Um Facade é um objeto que provê uma interface simplificada para um corpo de código maior, como por exemplo, uma biblioteca de classes.



FACADE

O Padrão de projeto Facade (ou Fachada) é um padrão de design de software usado comumente com programação orientada a objetos.
Este nome é uma analogia para uma fachada arquitetural.
Um Facade é um objeto que provê uma interface simplificada para um corpo de código maior, como por exemplo, uma biblioteca de classes.