Entra21 – Java Web

Plataforma Spring

O Spring consiste em uma plataforma completa de recursos para construção de aplicativos Java, que veio para simplificar o desenvolvimento em Java EE com diversos módulos que auxiliam na construção de sistemas reduzindo muito o tempo de desenvolvimento.

Essa plataforma conta com recursos avançados que abrangem várias áreas de uma aplicação com projetos/módulos prontos para uso, como:

- Spring Framework;
- Spring Boot;
- Spring Web(MVC);
- Spring Security;
- Spring Data;
- Spring Batch;
- Spring Cloud;
- outros.

Para visualizar e conhecer todos os projetos disponíveis na plataforma Spring, basta acessar o site Spring.io e conferir a lista completa através do link https://spring.io/projects.

Com esses projetos é possível construir aplicativos com funcionalidades avançadas e com uma produtividade muito maior, podendo focar mais nas regras de negócios e deixar as configurações de baixo nível por conta do Spring.

Spring Framework

É fácil confundir todo o ecossistema Spring com apenas o Spring Framework. Mas, o Spring Framework é apenas um, dentre o conjunto de projetos, que o Spring possui. Ele é o projeto do Spring que serve de base para todos os outros, talvez por isso haja essa pequena confusão.

O Spring foi pensado para que nossas aplicações pudessem focar mais na regra de negócio e menos na infraestrutura.

Como o projeto Spring Framework possui o módulo chamado Core Container, onde está implementado a **Inversão de Controle** que utiliza da **Injeção de Dependência**, ele se torna o projeto essencial para iniciar uma aplicação. Sendo assim a base de toda a plataforma Spring.

Talvez esse último parágrafo tenha ficado um pouco difícil de entender diante dessas terminologias. Primeiramente é preciso entender que todo framework é a aplicação de uma Inversão de Controle. Mas no que consiste essa Inversão de Controle afinal?

Inversão de Controle (IoC)

Inversão de Controle (IoC - Inversion of Control) é um processo em que um objeto define suas dependências sem criá-los. Este objeto delega a tarefa de construir tais dependências para um contêiner IoC.

Por exemplo, vamos considerar que temos duas classes, A e B, onde a classe A possui uma dependência da classe B, já que ela utiliza um método de B. Assim, a classe A sempre teria que criar uma nova instância da classe B para que assim pudesse utilizar seu método, como mostra na imagem abaixo

```
public class A{

   private B b;

   public void metodoA() {
      b = new B();
      b.metodoB();
      ...
   }
}
```

```
public class B{

   public void metodoB(){
   ...
   }
}
```

Figura 2: Exemplo de uma aplicação que não utiliza IoC (Inversão de Controle)

Porém, quando se utiliza a Inversão de Controle, a classe A não precisa se preocupar em criar uma instância de B, pois essa responsabilidade passa a ser do container do Spring Framework que realiza essa Inversão de Controle através da Injeção de Dependência. E o que seria a Injeção de Dependência afinal?

Injeção de Dependência (ID)

A Injeção de Dependência consiste na maneira, ou seja, na implementação utilizada pelo Spring Framework de aplicar a Inversão de Controle quando for necessário.

A Injeção de Dependência define quais classes serão instanciadas e em quais lugares serão injetadas quando houver necessidade. Assim, basta que a classe A crie um ponto de injeção da classe B, pelo construtor por exemplo, e quando houver a necessidade o container do Spring Framework irá criar uma instância da classe B para que a classe A possa utilizar o método b.metodoB(), como mostra na imagem abaixo.

```
public class A{

   private B b;

   public A(B b) {
      this.b = b;
   }
   public void metodoA() {
      b.metodoB();
      ...
   }
}
```

```
public class B{

   public void metodoB(){
   ...
   }
}
```

Figura 3: Exemplo de Injeção de Dependência

Core Container

O Spring Framework utiliza da Injeção de Dependência para aplicar a Inversão de Controle no sistema e toda essa implementação está presente no Core Container, onde fica a base de configuração do Spring Framework.

Quando a aplicação é executada, o Core Container é iniciado, as configurações da aplicação pré-definidas em classes ou arquivos XML são lidas e as dependências necessárias são definidas e criadas através da IoC e destruídas quando não mais forem utilizadas. Essas dependências são denominadas beans dentro do contexto do Spring, que consistem em objetos os quais possuem seu ciclo de vida gerenciado pelo container de IoC/ID do Spring. Esses passos definem o ciclo de vida de um Container, como pode ser mostrado também na imagem abaixo.

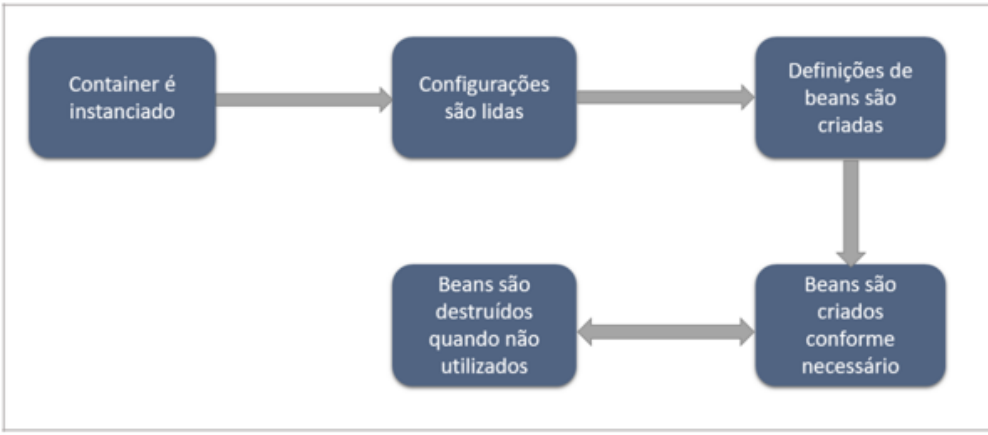


Figura 4: Ciclo de vida com Container

Beans

Como foi dito anteriormente, um bean consiste em um objeto que é instanciado, montado e gerenciado por um contêiner do Spring através da Inversão de Controle (IoC) e Injeção de Dependências.

Assim como o container, um bean também tem seu ciclo de vida, o qual é iniciado e criado pelo container, as dependências desse bean são injetadas, o método de inicialização é chamado e então, o bean assim é enviado para o cliente, no caso a classe que possui essa dependência, para ser utilizado e em seguida descartado.

Na prática, utilizando o exemplo anterior, quando o container é instanciado ele cria uma instância da classe B, chama o construtor da classe A para injetar esse bean e em seguida, a classe A utiliza esse bean através de b.metodoB(). Esse bean então é descartado quando não mais utilizado e tal ciclo pode ser visualizado na imagem abaixo.

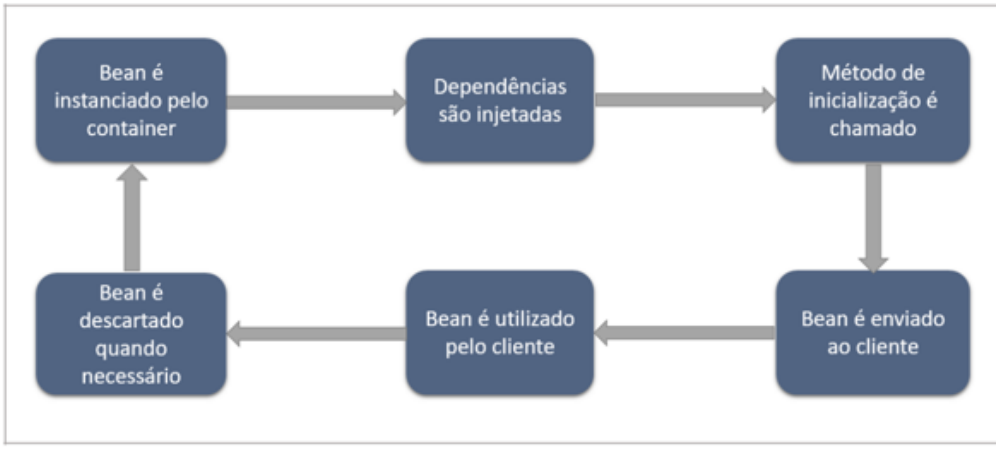


Figura 5: Ciclo de vida de um Bean

Configurando Beans no Spring

É preciso que o Spring conheça quais as classes da aplicação serão beans gerenciados por ele para que então seja aplicada a IoC/ID. Para isso há duas maneiras de configurar e determinar esses beans, utilizando configurações em arquivos XML ou através de anotações.

Na configuração por XML, não muito utilizada hoje em dia, é preciso definir tags <bean> dentro de uma tag principal <beans>

passando o path da classe, assim o Spring saberá quais classes ele irá gerenciar a criação de instâncias e a injeção de suas dependências.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean class="com.example.springboot.Produto"/>
    <bean class="com.example.springboot.ProdutoController"/>
    <bean class="com.example.springboot.ProdutoService"/>
    <bean class="com.example.springboot.ProdutoRepository"/>
</beans>
```

Figura 6: Configuração dos beans por xml

Na configuração por anotações, é possível utilizar os estereótipos do Spring para determinar de forma mais objetiva e específica qual o tipo de bean será cada classe. Há quatro principais tipos:

- @Component;
- @Service;
- @Controller;
- @Repository.

Assim, ao anotar determinada classe com algum desses estereótipos, o Spring entende que tal classe é um bean e será gerenciada por ele. Abaixo seguem alguns exemplos de beans utilizando configuração por anotações.

Exemplo 1: Bean do tipo Component

```
@Component
public class Product {

   private String name;
   private BigDecimal value;

   //... getters and setters
}
```

Exemplo 2: Bean do tipo Service

```
@Service
public class ProductService {

   // business rules

}
```

Exemplo 3: Bean do tipo Controller

```
@Controller
public class ProductController {

   // ... GET, POST, DELETE, UPDATE methods
```

Exemplo 4: Bean do tipo Repository

```
@Repository
public class ProductRepository {

    // database transaction methods

}
```

Considerando que os beans gerenciados pelo Spring já foram definidos a próxima questão é entender como o Spring saberá onde injetar as instâncias que ele irá criar com suas dependências. Para isso é preciso criar os pontos de injeção, que consistem em uma maneira de entregar as dependências ao objeto que necessita. Os dois tipos de pontos de injeção mais comuns são os construtores e setters, os quais podem ser visualizados nos exemplos abaixo.

Exemplo 5: Ponto de Injeção pelo método Construtor

```
@Service
public class ProductService {

   private ProductRepository productRepository;

   public ProductService(ProductRepository
productRepository) {
      this.productRepository = productRepository;
   }
   // business rules

}
```

Exemplo 6: Ponto de Injeção pelo método Setter

```
@Service
public class ProductService {

   private ProductRepository productRepository;

   public void setProductRepository(ProductRepository
productRepository) {
      this.productRepository = productRepository;
   }
   // business rules

}
```

Dentro do Spring, há uma outra maneira de se criar pontos de injeção de forma automática, utilizando a anotação @Autowired, como mostra o Exemplo 7.

Exemplo 7: Ponto de Injeção utilizando @Autowired

```
@Service
public class ProductService {

   @Autowired
   private ProductRepository productRepository;

   // business rules
```

Spring Boot

Iniciar uma aplicação do zero utilizando o Spring Framework pode ser um tanto quanto trabalhosa, pois é preciso fazer várias configurações em arquivos XML ou classe de configuração, configurar o Dispatcher Servlet, gerar um arquivo war, subir a aplicação dentro de um Servlet Container, como por exemplo o Tomcat, para então conseguir executar a aplicação e começar a implementar as regras de negócio.

O Spring Boot veio como uma extensão do Spring, que utiliza da base do Spring Framework para iniciar uma aplicação de uma forma bem mais simplificada, diminuindo a complexidade de configurações iniciais e o tempo para executar uma aplicação e deixá-la pronta para implementação das regras de negócio. Também já traz um servidor embutido que facilita ainda mais esse processo de start da aplicação.

Ao iniciar um projeto Spring Boot, basta uma dependência no arquivo pom.xml, spring-boot-starter, para que ele já traga internamente todas as dependências base do Spring Framework, como pode-se observar nas imagens abaixo.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Figura 7: Dependências iniciais de um projeto Spring Boot

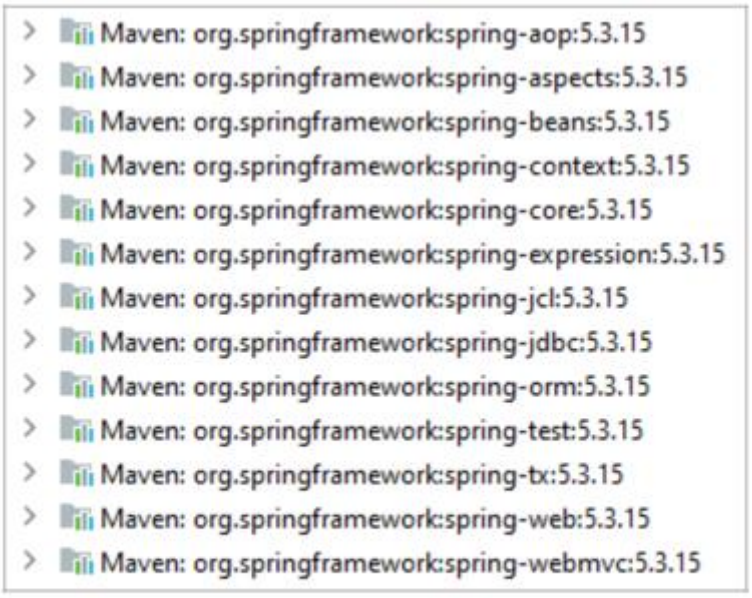


Figura 8: Módulos do Spring Core Container – Base do Framework

Resumindo, o Spring Boot é a soma do Spring Framework com um servidor embutido menos as configurações XML e classes de configurações.

Figura 9: Resumo Spring Boot

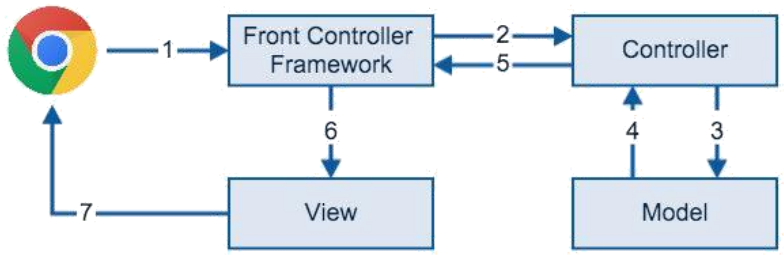
Spring MVC

Dentre os projetos Spring, o Spring MVC é o framework que te ajuda no desenvolvimento de aplicações web robustas, flexíveis e com uma clara separação de responsabilidades nos papéis do tratamento da requisição.

MVC é acrônimo de **Model**, **View** e **Controller**, e entender bem o que cada um deve fazer na aplicação é importante para termos uma aplicação bem escrita e fácil para dar manutenção. Vamos parar um pouco e pensar no que fazemos todos os dias quando estamos na internet.

Primeiro abrimos um browser (Chrome, Safari, Firefox), digitamos um endereço na “barra de endereços”, damos um “Enter” e pronto. Se nada der errado, uma página HTML será renderizada.

Mas, o que acontece entre o “Enter” e a página HTML ser renderizada? Claro que existem centenas de linguagens de programação e frameworks diferentes, mas nós vamos pensar no contexto do Spring MVC.



1. Acessamos uma URL no browser, que envia a requisição HTTP para o servidor que roda a aplicação web com Spring MVC. Esse servidor pode ser o Apache Tomcat, por exemplo. Perceba que quem recebe a requisição é o controlador do framework, o Spring MVC.

2. O controlador do framework irá procurar qual classe é responsável por tratar essa requisição, entregando a ela os dados enviados pelo browser. Essa classe faz o papel do ***controller***.

3. O ***controller*** passa os dados para o ***model***, que por sua vez executa todas as regras de negócio, como cálculos, validações e acesso ao banco de dados.

4. O resultado das operações realizadas pelo ***model*** é retornado ao ***controller***.

5. O ***controller*** retorna o nome da ***view***, junto com os dados que ela precisa para renderizar a página.

6. O framework encontra a ***view***, que processa os dados, transformando o resultado em um HTML.

7. Finalmente, o HTML é retornado ao browser do usuário.

Pare um pouco e volte na figura acima, leia mais uma vez todos os passos, desde a requisição do browser até a página ser renderizada. Como você deve ter notado, temos o ***Controller*** tratando a requisição. Ele é o primeiro componente que nós vamos programar para receber os dados enviados pelo usuário.

Mas é muito importante estar atento e não cometer erros adicionando regras de negócio, acessando banco de dados ou fazendo validações nessa camada, precisamos passar essa responsabilidade para o ***Model***.

No ***Model***, pensando em algo prático, é o local certo para usarmos o *JPA/Hibernate* para salvar ou consultar algo no banco de dados, é onde iremos calcular o valor do frete para entrega de um produto, por exemplo.

A ***View*** irá "desenhar", renderizar e transformar em HTML esses dados, para que o usuário consiga visualizar a informação, pois enquanto estávamos no

Controller e no ***Model***, estávamos programando em classes Java, e não em algo visual para o browser exibir ao usuário.

Essa é a ideia do MVC, separar claramente a responsabilidade de cada componente dentro de uma aplicação. Por quê? Para facilitar a manutenção do seu código, temos baixo acoplamento, e isso é uma boa prática de programação.

Spring Data JPA

O Spring Data JPA é um dos projetos do Spring Data.

O Spring Data é um projeto que tem o objetivo de simplificar o acesso a tecnologias de armazenamento de dados, sejam elas relacionais (MySQL, PostgreSQL, etc) ou não (MongoDB, Redis, etc).

O Spring Data já possui vários projetos dentro dele, como:

- Spring Data JPA
- Spring Data Commons
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data REST e outros

Mas aqui nós vamos focar no mais utilizado, que é o Spring Data JPA.

O suporte do Spring Framework para JPA já é muito bom, mas o projeto Spring Data JPA vai bem além. Ele ajuda os desenvolvedores padronizando o uso de algumas funcionalidades, e isso faz com que tenhamos menos esforço para implementar as mesmas coisas.

Um exemplo disso seria a implementação padrão que ele já nos dá em repositórios, incluindo métodos como save, delete, findOne, entre outros.

Esse exemplo não foi dado à toa. Mais para frente, faremos o uso da interface JpaRepository, que nos ajudará com essa funcionalidade.

JPA e Hibernate

Para entendermos o que é **JPA** e **Hibernate,** precisamos saber o que é **ORM**.

Mapeamento objeto-relacional (ou **ORM**, do inglês: Object-relational mapping) é uma técnica de desenvolvimento utilizada para reduzir a impedância da programação orientada a objetos utilizando bancos de dados relacionais. Técnica para aproximar o paradigma de desenvolvimento de aplicações orientadas a objetos ao paradigma do banco de dados relacional. O uso da técnica de mapeamento objeto-relacional é realizado através de um mapeador objeto-relacional que geralmente é a biblioteca ou framework que ajuda no mapeamento e uso do banco de dado.

Muitos confundem a diferença entre o **Hibernate** e o **JPA**. O **Hibernate** é um framework ORM, ou seja, a implementação física para persistir, remover, atualizar ou buscar dados no SGBD. É framework objeto relacional porque ajuda a representar tabelas de um banco de dados relacional através de classes. A vantagem dessa estratégia é a de automatizar as tarefas com banco de dados de forma que é possível simplificar o código da aplicação.

O **JPA** é uma camada que descreve uma interface comum para frameworks ORM. Java Persistence API (ou simplesmente **JPA**) é uma especificação oficial que descreve como deve ser o comportamento dos frameworks de persistência Java, que desejarem implementá-la. A **JPA** define um meio de mapeamento objeto-relacional para objetos Java simples e comuns (POJOs), denominados beans de entidade.

Spring Security

Como o próprio nome diz, esse projeto trata da segurança em nível de aplicação.

Ele tem um suporte excelente para autenticação e autorização.

Spring Security torna bem simples a parte de autenticação. Com algumas poucas configurações, já podemos ter uma autenticação via banco de dados, LDAP ou mesmo por memória. Sem falar nas várias integrações que ele suporta e na possibilidade de criar as suas próprias.

Quanto a autorização, ele é bem flexível também. Através das permissões que atribuímos aos usuários autenticados, podemos proteger as requisições web (como as telas do nosso sistema, por exemplo), a simples invocação de um método e até a instância de um objeto.

Maven

O Maven é uma ferramenta de código aberto mantida pela Apache. Trata-se de uma ferramenta de gestão de dependências e um task runner. Em outras palavras, o Maven automatiza os processos de obtenção de dependências e de compilação de projetos Java.

Quando criamos um projeto Maven, este projeto fica atrelado a um arquivo principal: o `pom.xml`. Neste arquivo POM (Project Object Model), nós descrevemos as dependências de nosso projeto e a maneira como este deve ser compilado. Com o Maven, é possível, por exemplo, automatizar a execução de testes unitários durante a fase de build, entre outras automatizações.
Com o Maven, não temos mais a necessidade de baixarmos as dependências de nosso projeto e as configurar dentro do Build Path/Classpath de nossas aplicações. Se nós precisamos do driver do MySQL, por exemplo, simplesmente registramos essa dependência no pom.xml. As ferramentas de automação do Maven irão detectar esta dependência, baixa-la e configura-la no Build Path/Classpath de nosso projeto.

Thymeleaf

A *view* irá retornar apenas um HTML para o browser do cliente, mas isso deixa uma dúvida: Como ela recebe os objetos Java, enviados pelo *controller*, e os transforma em HTML?

Nessa hora que entra em ação o Thymeleaf!

Teremos um código HTML misturado com alguns atributos do Thymeleaf, que após processado, será gerado apenas o HTML para ser renderizado no browser do cliente.

O Thymeleaf é um template engine para projetos Java que facilita a criação de páginas HTML. Sendo assim, ele serve para gerar páginas HTML no lado servidor de forma dinâmica, permitindo a troca de informações entre o código Java e as página HTML, de tal maneira ele garante que o desenvolvedor consiga criar templates de forma mais fácil para suas aplicações.

Não é um projeto Spring, foi criado para facilitar a criação da camada de *view*, com uma forte integração com o Spring, e uma boa alternativa ao JSP.

O principal objetivo do Thymeleaf é prover uma forma elegante e bem formatada para criarmos nossas páginas. O dialeto do Thymeleaf é bem poderoso, como você verá no desenvolvimento da aplicação, mas você também pode estendê-lo para customizar, de acordo com suas necessidades.

Para você ver como ele funciona, vamos analisar o código abaixo.

```
<tr th:each="convidado : ${convidados}"> <td
    th:text="${convidado.nome}"></td>

    <td th:text="${convidado.quantidadeAcompanhantes}"></td>
</tr>
```

A expressão ${} interpreta variáveis locais ou disponibilizadas pelo *controller*.

O atributo th:each itera sobre a lista convidados, atribuindo cada objeto na variável local convidado. Isso faz com que vários elementos tr sejam renderizados na página.

Dentro de cada tr existem 2 elementos td. O texto que eles irão exibir vem do atributo th:text, junto com a expressão ${}, lendo as propriedades da variável local convidado

Exercício:

Implementar uma aplicação Web.

Vamos utilizar o Modulo Spring Boot para criação do Projeto e a configuração das dependências.

Ferramentas utilizadas:

- IDE Eclipse
- MySql
- SpringBoot que já traz o Spring Framework
- Maven
- Spring MVC
- Spring Data JPA
- Thymeleaf
- HTML
- Materialize