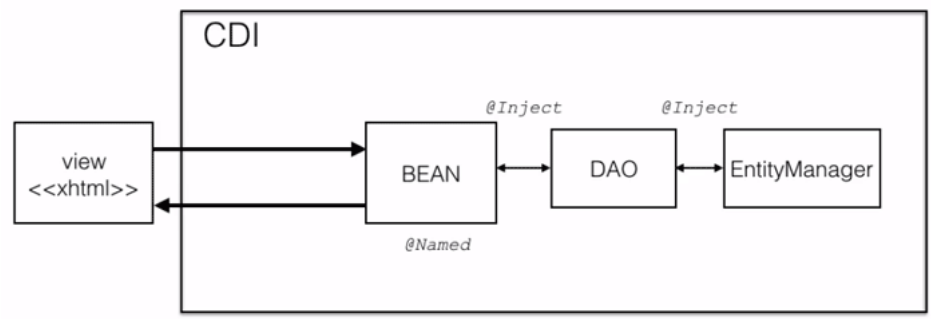
02

Integração com Spring

Transcrição

Hoje recebi um e-mail de um aluno sobre como utilizar Spring com JSF. Dada a importância, vamos dar uma leve passada, não era o nosso objetivo mas vamos fazer uma introdução a esse framework tão famoso e importante no mercado.

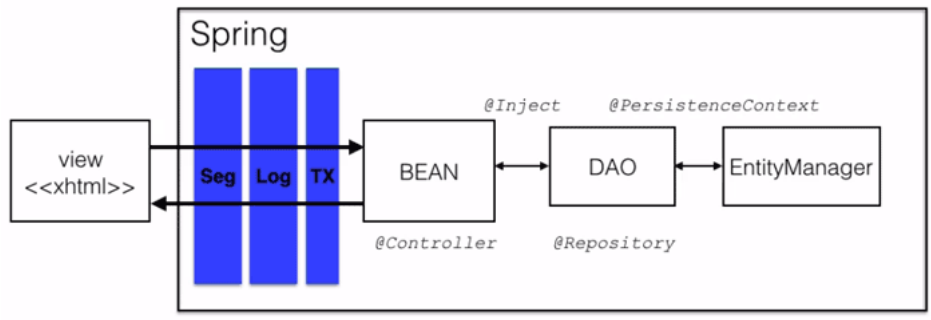
Antes de entrarmos no universo de Spring, vamos revisar o que acontecia quando usávamos o CDI junto ao projeto desenvolvido. Observe a imagem a seguir:



Então o CDI é um *container* que administra os nossos objetos: nossos *bean`s*, nossos `DAO` s e também levanta toda a camada do JPA. Nós ficávamos via CDI injetando esses objetos para não ficarmos presos às implementações e usar menos `new` s, e usar ainda menos o `JPAUtil` . Para fazer a transação, nós criamos um interceptador com CDI para deixar bem modularizado e centralizado. Logo o CDI ajuda com essas classes de infraestrutura de camadas: criando e amarrando esses objetos para nós.

Aí você deve estar se perguntando... Mas e o Spring? Onde ele entra nesse contexto!? Na verdade, o Spring faz o mesmo que o CDI, só que com mais opções. A popularização dele foi justamente nessa parte central de amarração de objetos. Aliás, o CDI, que é uma especificação, só existe pelo sucesso que esse núcleo do Spring fez. Logo o CDI e o Spring são concorrentes diretos.

Adaptando nosso projeto ao uso do Spring poderíamos usar a mesma representação já usada com CDI, com poucas alterações, como pode-se perceber pela imagem abaixo:



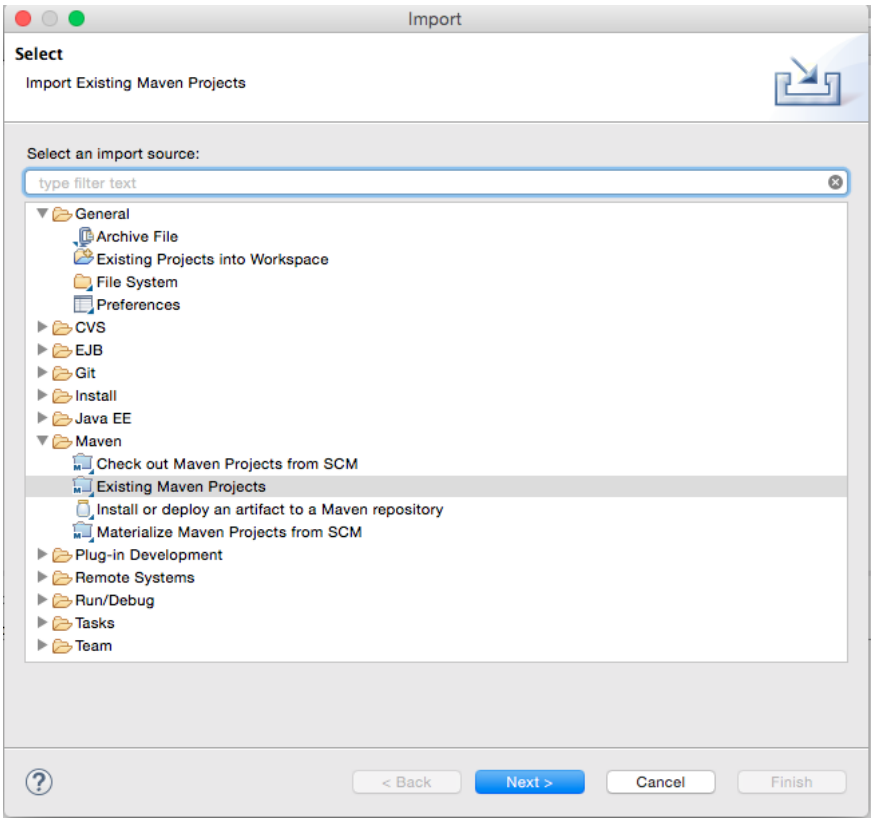
Da mesma forma que o CDI, o Spring é um *container* que irá criar nosso *bean*, os `DAO` s e inicializar o JPA. Além disso, também possui interceptadores para transação, *log* e há até sofisticados interceptadores sobre segurança, que é o *Spring Security*.

A diferença dele é que as configurações são um pouco diferentes e vem mais preparado para a utilização, sem a necessidade de criarmos *producer*s e interceptadores da transação, por exemplo, além de outras coisas. Devemos ficar atentos para as anotações que também mudam: o *bean* se torna um `@Controller`, apesar de `@Named` (usado no CDI) também funcionar, e os `DAO`s se tornam um `@Repository`. Para as amarrações, temos algumas alternativas, embora o mais comum é utilizarmos um `@Inject` (similar ao do CDI), atente-se que no caso de um `EntityManager` devemos usar um `@PersistenceContext`, em vez de `@Inject`.

Em resumo, poderíamos dizer que o que vai mudar não é o conceito, continuamos usando um *container* que administra nossos objetos, vamos fazer utilização de interceptadores e injeção de dependências. Tudo isso deixou o Spring popular e ele foi o pioneiro, pode-se dizer que o CDI na verdade só especificou isso dentro de JavaEE. Sabendo disso tudo vamos analisar como ficaria o nosso projeto para trabalharmos com esse framework tão famoso.

Faça o download do projeto já configurado aqui (https://s3.amazonaws.com/caelum-online-public/jsf-cdi/stages/livraria-maven-spring-completo.zip).

No Eclipse, vamos importar esse projeto como um *Existing Maven Project*:



Selecione o projeto baixado anteriormente. O Eclipse já identifica o `pom.xml` e vamos dar um *Finish* para que as dependências sejam baixadas.

Abrindo o projeto e expandindo o **src/main/java**, é possível perceber que não temos mais o pacote `br.com.caelum.livraria.tx`, já que o Spring tem isso pronto para ser utilizado, não havendo a necessidade de criarmos um interceptador.

Expandindo o **src/main/resources/META-INF** percebemos um `persistence.xml` que é igual ao anterior.

Vamos entrar agora nas configurações do projeto web, fazemos isso expandindo **src/main/webapp/WEB-INF** e verificando o arquivo `web.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:web="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web
    version="3.0">

    <display-name>livraria</display-name>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>

    <!-- restante do arquivo -->

</web-app>
```

Note que há uma configuração de dois *listeners*, usados justamente para o servidor iniciar o *container* do Spring. Essa configuração é análoga ao usada no CDI, com o `bean.xml` e o `context.xml`.

Ainda analisando o arquivo `web.xml`, é possível perceber que temos um filtro:

```
<filter>
    <filter-name>openEntityManagerInViewFilter</filter-name>
    <filter-class> org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter</filter-cla
</filter>

<filter-mapping>
    <filter-name>openEntityManagerInViewFilter</filter-name>
    <url-pattern>*.xhtml</url-pattern>
</filter-mapping>
```

Esse filtro, `OpenEntityManagerInViewFilter`, é usado para abertura e fechamento de um `EntityManager` no início e fim de uma requisição, respectivamente. Fazendo um comparativo com o CDI, nós tínhamos utilizado um *producer*, onde implementávamos essa forma de abrir e fechar o `EntityManager`. Com o Spring, já temos esses facilitadores prontos e nos preocupamos somente em configurá-lo.

Além dessas configurações comentadas temos as básicas do JSF com o Primefaces.

Ainda dentro de **src/main/webapp/WEB-INF**, vamos verificar o arquivo `faces-config.xml`, que possui uma configuração a mais:

```
<?xml version="1.0" encoding="UTF-8"?>

<faces-config version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/w

    <application>
        <message-bundle>resources.application</message-bundle>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
        <locale-config>
            <default-locale>en</default-locale>
        </locale-config>
    </application>

    <lifecycle>
        <phase-listener>br.com.caelum.livraria.util.Autorizador</phase-listener>
        <phase-listener>br.com.caelum.livraria.util.LogPhaseListener</phase-listener>
    </lifecycle>

</faces-config>
```

Esse `el-resolver`, como pode-se imaginar, serve justamente para que as `Strings` das *Expression Languages* utilizadas nas nossas *views* do JSF possam ser processadas por um *bean*. Lembre-se que quem faz essa ligação é o próprio Spring.

Por último, mas não menos importante, temos o arquivo `applicationContext.xml`, que é bem similar ao arquivo `bean.xml` do CDI. Vamos analisá-lo:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:tx="http://www.springframework.
    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-4.1.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.1.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.1.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">

    <!-- For Scanning the packages net.javaonline.spring.inventory & net.javaonline.web.jsf.inv
        and registering the beans with the applicationContext -->
    <context:component-scan base-package="br.com.caelum.livraria" />
    <context:annotation-config />
    <tx:annotation-driven />


    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="livraria" />
    </bean>
```

```
</beans>
```

Na última configuração desse arquivo, temos:

```
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="livraria" />
</bean>
```

Esse *bean* é o responsável por ler as informações do nosso `persistence.xml` . A *property* de *name* `persistenceUnitName` recebe como **livraria** como valor, que é o mesmo valor usado no **src/main/resources/META-INF/persistence.xml**:

```
<persistence-unit name="livraria" transaction-type="RESOURCE_LOCAL">
```

Logo que a aplicação é inicializada pelo servidor, essa configuração sobe a JPA com essa unidade de persistência.

Ainda no `applicationContext.xml` , encontramos a configuração do gerenciador de transação. Nele fazemos uso de uma própria classe do Spring, que resolve isso para nós:

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Perceba que além das configurações vistas acima, temos ainda as que habilitam o uso de anotações:

```
<context:component-scan base-package="br.com.caelum.livraria" />
<context:annotation-config />
<tx:annotation-driven />
```

Com todas as configurações feitas, vamos analisar como ficariam as nossas classes de `DAO` s. Lembre-se que as anotações recebem nomes diferentes no mundo Spring:

```
@Repository
@SuppressWarnings("serial")
public class AutorDao implements Serializable{


    @PersistenceContext
    EntityManager em;

    private DAO<Autor> dao;

    @PostConstruct
    void init() {
        this.dao = new DAO<Autor>(this.em, Autor.class);
    }

    // restante do código
```

```
}
```

Perceba que os `DAO`s são chamados de `Repository` e além disso injetamos um `EntityManager` através de um `@PersistenceContext`, e não um `@Inject`.

Analisando agora nossos *beans*, temos:

```
@SuppressWarnings("serial")
@Controller
public class AutorBean implements Serializable{

    private Autor autor = new Autor();

    @Inject
    private AutorDao dao;

    private Integer autorId;

    // restante do código
}
```

Perceba que ao invés de usar um `@Named`, que até funcionaria, o Spring usa como nomenclatura um `@Controller`. Já na injeção de dependência do `AutorDao`, basta usarmos um `@Inject`.

Portanto, usar o Spring no nosso projeto não exigiu grandes esforços. Aqui demos uma pincelada nele, se você tem interesse em se aprofundar nesse framework tão consagrado e poderoso veja o nosso curso aqui mesmo no Alura: **Curso Spring MVC: É hora de criar uma webapp com Spring MVC4** (https://cursos.alura.com.br/course/spring-mvc).