

FIAP GRADUAÇÃO

DIGITAL BUSINESS ENABLEMENT

Prof. Me. Thiago T. I. Yamamoto

#14 –INTEGRAÇÃO SPRING MVC JPA/HIBERNATE



thiagoyama



thiagoyama@gmail.com

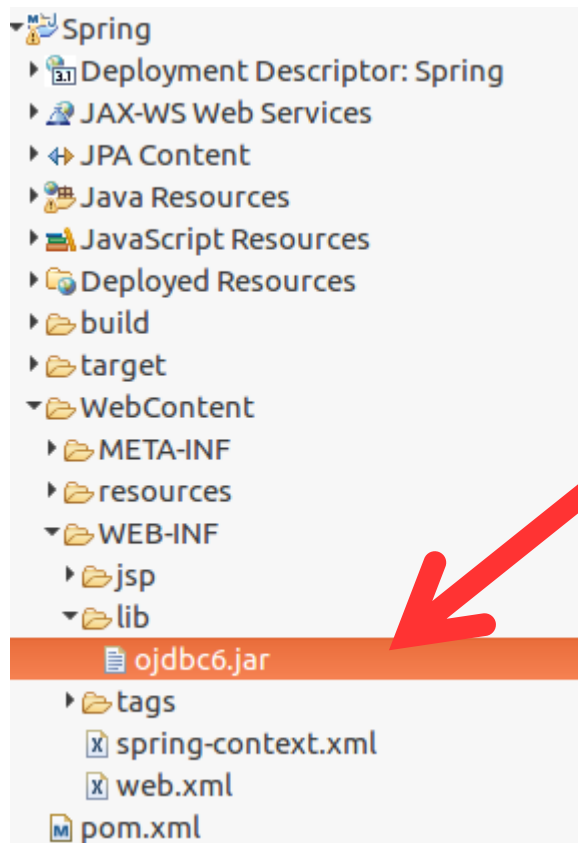
#14 – INTEGRAÇÃO SPRING MVC E JPA

- Configuração do JPA/Hibernate
- Data Source
- Persistence Context
- Repository e Autowire
- Controle de Transação

Vamos configurar a aplicação para utilizar o JPA/Hibernate. Para isso precisamos adicionar as **dependências** (bibliotecas) no arquivo **pom.xml**.

```
<!-- JPA/Hibernate -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.1.3.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.2.4.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.1.3.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>4.1.0.RELEASE</version>
</dependency>
```

Para utilizar o drive do oracle com o maven é preciso realizar um registro na oracle ou realizar a instalação do driver no repositório local de forma manual.



Por isso, vamos adicionar a biblioteca diretamente no nosso projeto, copiando no diretório **/WebContent/WEB-INF/lib**

https://blogs.oracle.com/dev2dev/entry/oracle_maven_repository_instructions_for

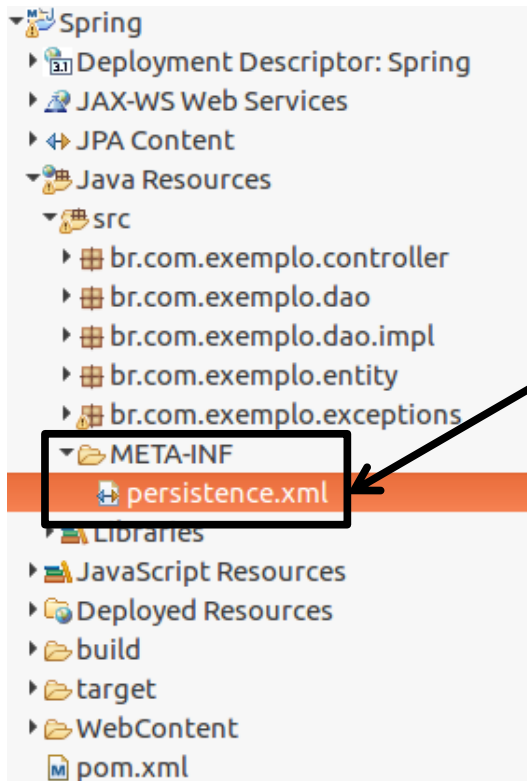
- O framework **Spring** controla o ciclo de vida dos objetos (componentes), dessa forma, o framework pode controlar também os **EntityManager** e **EntityManagerFactory**.
- Precisamos configurar no **spring-context.xml** o **Data Source**, responsável pela conexão com o banco de dados e o bean que representa o **EntityManagerFactory**, que é controlado pelo spring.

No **Data Source** é preciso configurar o **driver** que será utilizado, a **url**, **usuário** e **senha** do banco de dados.

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="CLIENTE_ORACLE" />
  <property name="dataSource" ref="oracleDataSource" />
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
  </property>
</bean>

<bean id="oracleDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="oracle.jdbc.OracleDriver" />
  <property name="url" value="jdbc:oracle:thin:@oracle.fiap.com.br:1521:orcl" />
  <property name="username" value="OPS$PF0392" />
  <property name="password" value="123456" />
</bean>
```

- Precisamos do arquivo **persistence.xml** que deve estar dentro da pasta **src/META-INF**.
- Neste arquivo **não** será mais preciso as **configurações** de **driver**, **url**, **usuário** e **senha** do banco, pois elas estão configuradas no data source.



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="CLIENTE_ORACLE">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.dialect">
    </properties>
  </persistence-unit>
</persistence>
```


- Após as configurações podemos utilizar a Inversão de Controle para injetar o **EntityManager** dentro do **DAO**.
- Dessa forma, no **GenericDao**, vamos anotar o **EntityManager** com **@PersistenceContext**. Com isso não será mais necessário receber o EntityManager no **construtor**, já que o spring será responsável por controlar o EntityManager.

```
public class GenericDAOImpl<T, K> implements GenericDao<T, K> {  
    → @PersistenceContext  
    protected EntityManager em;  
  
    private Class<T> classe;  
  
    @SuppressWarnings("unchecked")  
    public GenericDAOImpl() { ←  
        classe =  
            (Class<T>) ((ParameterizedType)  
                getClass().getGenericSuperclass())  
                    .getActualTypeArguments()[0];  
    }  
    //...  
}
```

Construtor sem argumentos

- O Spring possui 4 anotações para transformar classes em componentes do Spring:
 - **@Controller**: componente de controle para a camada de apresentação;
 - **@Repository**: componente de DAO para a camada de persistência;
 - **@Service**: componente de serviço para a camada de negócio;
 - **@Component**: componente genérico;
- Esses componentes são administrado pelo Spring que controla o seu ciclo de vida e também ligando uns aos outros.

- Na implementação de um DAO específico vamos adicionar a anotação `@Repository` para se tornar um componente do spring, assim o spring pode injeta-lo no controller.
- Para o spring injetar qualquer componente dentro de outro componente basta utilizar a anotação `@Autowired`.

→ `@Repository`
`public class ProdutoDaoImpl extends GenericDAOImpl<Produto, Integer> implements ProdutoDao{`
`}`

```
@Controller
@RequestMapping("/produto")
public class ProdutoController {
    → @Autowired
      private ProdutoDao dao;

    //..
}
```

- Vamos configurar o spring para gerenciar as transações;
- No arquivo **spring-context.xml** vamos habilitar o **gerenciador de transações**:

```
<!-- Transações -->  
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">  
    <property name="entityManagerFactory" ref="entityManagerFactory" />  
</bean>  
  
<tx:annotation-driven transaction-manager="transactionManager" />
```

Com o controle de transação não será preciso realizar **begin**, **commit** ou **rollback**. Assim, podemos remover do código:

```
public class GenericDAOImpl<T, K> implements GenericDao<T, K> {

    @PersistenceContext
    protected EntityManager em;

    private Class<T> classe;


    @SuppressWarnings("unchecked")
    public GenericDAOImpl() {
        classe = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass())
            .getActualTypeArguments()[0];
    }

    @Override
    public void insert(T entity) throws DBCommitException {
        em.persist(entity);
    }

    @Override
    public void update(T entity) throws DBCommitException {
        em.merge(entity);
    }

    @Override
    public void delete(K id) throws DBCommitException, IdNotFoundException {
        T entity = findById(id);
        if (entity == null)
            throw new IdNotFoundException();
        em.remove(entity);
    }
}
```

- Agora podemos anotar o método que precisa de uma transação com **@Transactional**, dessa forma o **begin()**, **commit()**, **rollback()** é controlado pelo spring.
- Podemos anotar a **classe** com **@Transactional**, assim **todos** os **métodos** vão receber uma transação.



```
@Controller
@RequestMapping("/produto")
public class ProdutoController {

    @Autowired
    private ProdutoDao dao;

    @PostMapping(value="cadastrar")
    @Transactional
    public ModelAndView processarForm(Produto produto){
        ModelAndView retorno = new ModelAndView("produto/sucesso");
        try {
            dao.insert(produto);
        } catch (DBCommitException e) {
            e.printStackTrace();
        }
        retorno.addObject("prod", produto);
        return retorno;
    }
}
```

Copyright © 2017 - Prof. Me. Thiago T. I. Yamamoto

Todos direitos reservados. Reprodução ou divulgação total ou parcial deste documento é expressamente proibido sem o consentimento formal, por escrito, do Professor (autor).

*“Aprender é a única coisa que a mente nunca se cansa,
nunca tem medo e nunca se arrepende”*