CAPÍTULO 16

Apêndice - Integração do Spring com JPA

"O caminho do inferno está pavimentado de boas intenções."
— Marx

Neste capítulo, você aprenderá a:

- Integrar JPA com Spring;
- Injetar o EntityManager dentro do DAO;
- Gerenciar as transações com Spring;

16.1 - GERENCIANDO O ENTITYMANAGER

Dentre as diversas características do Spring uma das que mais chama a atenção é a integração nativa com diversas tecnologias importantes do mercado. Podemos citar o **Hibernate** e o **JPA**.

Vimos que o Spring é um container que controla objetos (ou componentes), administrando o ciclo da vida deles, inclusive ligando uns aos outros (amarrando-os). No caso do JPA, queremos que o Spring cuide da abertura e do fechamento da `EntityManagerFactory` e do `EntityManager`. Com isso, **todos** os componentes do Spring podem receber como dependência um `EntityManager`, mas agora controlado pelo Spring. Novamente aplicando a inversão de controle.

Mas a inversão de controle não se limita a inicialização de objetos. O Spring também tira do desenvolvedor a responsabilidade de controlar a transação. Ao delegar o controle do `EntityManager` para o Spring, ele consegue abrir e fechar transações automaticamente.

Veja como é a inicialização do JPA sem a ajuda do Spring:

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();

manager.getTransaction().begin();

//aqui usa o EntityManager

manager.getTransaction().commit();
manager.close();
```

Nesse pequeno trecho de código podemos ver como é trabalhoso inicializar o JPA manualmente. É preciso abrir e fechar todos os recursos para realmente começar a usar o `EntityManager`. O Spring deve assumir essas responsabilidades e facilitar assim o uso do JPA.

16.2 - Configurando o JPA no Spring

Para integrar-se com o JPA, o Spring disponibiliza um *Bean* que devemos cadastrar no arquivo XML. Ele representa a `EntityManagerFactory`, mas agora gerenciada pelo Spring. Ou seja, toda inicialização da fábrica fica ao encargo do Spring:

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

Repare que o *Bean* define *Hibernate* como implementação do JPA e recebe a `mysqlDataSource` que já definimos anteriormente dentro do XML do Spring. Como a nossa `Datasource` já sabe os dados do driver, login e senha, o arquivo `persistence.xml` do JPA também fica mais simples:

```
<persistence 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"
    version="2.0">

 <persistence-unit name="tarefas">

   <provider>org.hibernate.ejb.HibernatePersistence</provider>

   <class>br.com.caelum.tarefas.modelo.Tarefa</class>

   <properties>
```

```
  <!-- SEM as propriedades URL, login, senha e driver -->

  <property name="hibernate.dialect"
      value="org.hibernate.dialect.MySQL5InnoDBDialect" />
  <property name="hibernate.show_sql" value="true" />
  <property name="hibernate.format_sql" value="true" />
  <property name="hibernate.hbm2ddl.auto" value="update" />
   </properties>
 </persistence-unit>

</persistence>
```


16.3 - Injetando o EntityManager

Com JPA configurado podemos aproveitar a inversão de controle e injetar o `EntityManager` dentro de qualquer componente administrado pelo Spring.

Vamos criar uma classe `JpaTarefaDao` para usar o `EntityManager`. A classe `JpaTarefaDao` precisa do `EntityManager`. Como JPA é uma especificação, o Spring também aproveita uma anotação de especificação para receber a dependência. Nesse caso não podemos usar a anotação `@Autowired` do Spring e sim `@PersistenceContext`. Infelizmente a anotação `@PersistenceContext` não funciona com construtores, exigindo um outro *ponto de injeção*. Usaremos o atributo para declarar a dependência:

```
@Repository
public class JpaTarefaDao{

  @PersistenceContext
  private EntityManager manager;

  //sem construtor
```

```
    //aqui vem os métodos
}
```

Nos métodos do `JpaTarefaDao` faremos o uso do `EntityManager` usando os métodos já conhecidos como `persist(..)` para persistir uma tarefa ou `remove(..)` para remover:

```
@Repository
public class JpaTarefaDao{

  @PersistenceContext
  private EntityManager manager;

  //sem construtor

    public void adiciona(Tarefa tarefa) {
      manager.persist(tarefa);
    }

    public void altera(Tarefa tarefa) {
      manager.merge(tarefa);
    }

    public List<Tarefa> lista() {
      return manager.createQuery("select t from Tarefa t").getResultList();
    }

    public Tarefa buscaPorId(Long id) {
      return manager.find(Tarefa.class, id);
    }

    public void remove(Tarefa tarefa) {
      Tarefa tarefaARemover = buscaPorId(tarefa.getId());
      manager.remove(tarefaARemover);
    }

    public void finaliza(Long id) {
      Tarefa tarefa = buscaPorId(id);
      tarefa.setFinalizado(true);
      tarefa.setDataFinalizacao(Calendar.getInstance());
    manager.merge(tarefa);
    }
}
```

A implementação do `JpaTarefaDao` ficou muito mais simples se comparada com o `JdbcTarefaDao`, em alguns métodos é apenas uma linha de código.

16.4 - Baixo acoplamento pelo uso de interface

Voltando para nossa classe `TarefasController` vamos lembrar que injetamos o `JdbcTarefaDao` para trabalhar com o banco de dados:

```
@Controller
```

```
public class TarefasController {

  private final JdbcTarefaDao dao;

  @Autowired
  public TarefasController(JdbcTarefaDao dao) {
    this.dao = dao;
  }

  @RequestMapping("mostraTarefa")
  public String mostra(Long id, Model model) {
    model.addAttribute("tarefa", dao.buscaPorId(id));
    return "tarefa/mostra";
  }

  //outros métodos omitidos
}
```

Quando olhamos para os métodos propriamente ditos, por exemplo, `mostra`, percebemos que o que é realmente necessário para executá-lo é *algum DAO*. A lógica em si é independente da instância específica que estamos instanciando, ou seja para a classe `TarefasController` não importa se estamos usando JDBC ou JPA. Queremos um desacoplamento da implementação do DAO especifico, e algo deve decidir qual implementação usaremos por baixo dos panos.

O problema desse tipo de chamada é que, no dia em que precisarmos mudar a implementação do DAO, precisaremos mudar nossa classe. Agora imagine que tenhamos 10 controladores no sistema que usem nosso `JdbcTarefaDao`. Se todas usam a implementação específica, quando formos mudar a implementação para usar JPA para persistência por exemplo, digamos, `JpaTarefaDao`, precisaremos mudar em vários lugares.

O que precisamos então é apenas uma `TarefaDao` dentro da classe `TarefasController`, então vamos definir (*extrair*) uma nova interface `TarefaDao` :

```
public interface TarefaDao {

  Tarefa buscaPorId(Long id);
  List<Tarefa> lista();
  void adiciona(Tarefa t);
  void altera(Tarefa t);
  void remove(Tarefa t);
  void finaliza(Long id);
}
```

E a classe `JdbcTarefaDao` implementará essa interface:

```
@Repository
public class JdbcTarefaDao implements TarefaDao {

  //implementação do nosso dao usando jdbc
}
```

Dessa maneira vamos injetar uma implementação compatível com a interface dentro da classe `TarefasController`

```
@Controller
public class TarefasController {

  private TarefaDao dao; //usando a interface apenas!

  @Autowired
  public TarefasController(TarefaDao dao) {
    this.dao = dao;
  }

  //métodos omitidos
}
```

Agora a vantagem dessa abordagem é que podemos usar uma outra implementação da interface `TarefaDao` sem alterar a classe `TarefasController` , no nosso caso vamos usar `JpaTarefaDao`:

```
@Repository
public class JpaTarefaDao implements TarefaDao{

  @PersistenceContext
  EntityManager manager;

  //sem construtor

  //métodos omitidos
}
```

Repare que o DAO também vai implementar a interface `TarefaDao`. Assim temos duas implementações da mesma interface:

Como o Spring não sabe qual das duas implementações deve ser utilizada é preciso qualificar a dependência:

```
@Controller
public class TarefasController {

  private TarefaDao dao; //usa apenas a interface!

  @Autowired
  @Qualifier("jpaTarefaDao")
  public TarefasController(TarefaDao dao) {
    this.dao = dao;
  }
```

```
  //métodos omitidos
}
```

Dessa maneira o Spring injetará o `JpaTarefaDao`.

16.5 - Gerenciando a transação

Para o suporte à transação ser ativado precisamos fazer duas configurações no XML do Spring. A primeira é habilitar o **gerenciador de transação** (`TransactionManager`). Porém, como o Spring pode controlar JPA, Hibernate e outros, devemos configurar o gerenciador exatamente para uma dessas tecnologias. No caso do JPA, a única dependência que o `JpaTransactionManager` precisa é uma `entityManagerFactory`:

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

`entityManagerFactory` é o nome do *Bean* configurado anteriormente:

A segunda parte é avisar que o controle de transações será feito via anotação, parecido com a forma de habilitar o uso de anotações para o Spring MVC.

```
<tx:annotation-driven/>
```

Por fim, podemos usar o gerenciamento da transação dentro das nossas classes. Aqui fica muito simples, é só usar a anotação `@Transactional` no método que precisa de uma transação, por exemplo no método `adiciona` da classe `TarefasController`:

```
@Transactional
@RequestMapping("adicionaTarefa")
public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

  if(result.hasFieldErrors("descricao")) {
    return "tarefa/formulario";
  }

  dao.adiciona(tarefa);
  return "redirect:listaTarefas";
}
```

A mesma anotação também pode ser utilizada na classe. Isso significa que todos os métodos da classe serão executados dentro de uma transação:

```
@Transactional
@Controller
public class TarefasController {
```

Repare aqui a beleza da inversão de controle. Não há necessidade de chamar `begin()`, `commit()` ou `rollback()`, basta usar `@Transactional` e o Spring assume a responsabilidade em gerenciar a transação.

Há um problema ainda, usando o gerenciamento de transação pelo Spring exige a presença do construtor padrão sem parâmetros. Vamos trocar aqui também o ponto de injeção do construtor para o atributo. A classe completa fica como:

```
package br.com.caelum.tarefas.controller;

//imports

@Controller
@Transactional
public class TarefasController {

  @Autowired
  TarefaDao dao;

  @RequestMapping("novaTarefa")
  public String form() {
    return "tarefa/formulario";
  }

  @RequestMapping("adicionaTarefa")
  public String adiciona(@Valid Tarefa tarefa, BindingResult result) {

    if (result.hasFieldErrors("descricao")) {
      return "tarefa/formulario";
    }

    dao.adiciona(tarefa);
    return "tarefa/adicionada";
  }

  @RequestMapping("listaTarefas")
  public String lista(Model model) {
    model.addAttribute("tarefas", dao.lista());
    return "tarefa/lista";
  }

  @RequestMapping("removeTarefa")
  public String remove(Tarefa tarefa) {
    dao.remove(tarefa);
    return "redirect:listaTarefas";
  }

  @RequestMapping("mostraTarefa")
  public String mostra(Long id, Model model) {
    model.addAttribute("tarefa", dao.buscaPorId(id));
    return "tarefa/mostra";
  }

  @RequestMapping("alteraTarefa")
```

```
  public String altera(Tarefa tarefa) {
    dao.altera(tarefa);
    return "redirect:listaTarefas";
  }

  @RequestMapping("finalizaTarefa")
  public String finaliza(Long id, Model model) {
    dao.finaliza(id);
    model.addAttribute("tarefa", dao.buscaPorId(id));
    return "tarefa/finalizada";
  }
}
```


16.6 - Exercícios: Integrando JPA com Spring

1. Vamos usar JPA através do Spring. Para isso é preciso copiar os JAR seguintes:

- `aopalliance.jar`
- `spring-orm-4.x.x.RELEASE.jar`
- `spring-tx-4.x.x.RELEASE.jar`

Para isso:

- Vá ao Desktop, e entre no diretório `Caelum/21/jars-jpa/spring4`.
- Copie os JARs (`CTRL+C`) e cole-o (`CTRL+V`) dentro de `workspace/fj21-tarefas/WebContent/WEB-INF/lib`

2. No projeto `fj21-tarefas`, vá a pasta `WebContent/WEB-INF` e abra o arquivo `spring-context.xml`.

Nele é preciso declarar o `entityManagerFactory` e o gerenciador de transações.

Você pode copiar essa parte do XML do arquivo `Caelum/21/jars-jpa/spring4/spring-`

jpa.xml.txt. Copie o conteúdo do arquivo e cole dentro do spring-context.xml:

```
<!-- gerenciamento de jpa pelo spring -->
<bean id="entityManagerFactory"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="mysqlDataSource" />
  <property name="jpaVendorAdapter">
    <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
  </property>
</bean>

<!-- gerenciamento da transação pelo spring -->
<bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven/>
```

Com essas declarações, Spring gerencia a EntityManagerFactory e habilita o gerenciamento de transações.

3. O próximo passo é criar a interface TarefaDao. Crie uma nova interface dentro do package br.com.caelum.tarefas.dao:

```
package br.com.caelum.tarefas.dao;

// imports omitidos

public interface TarefaDao {

  Tarefa buscaPorId(Long id);
  List<Tarefa> lista();
  void adiciona(Tarefa t);
  void altera(Tarefa t);
  void remove(Tarefa t);
  void finaliza(Long id);
}
```

4. Crie uma classe **JpaTarefaDao** que recebe o EntityManager. Implemente a interface TarefaDao com todos os métodos.

```
package br.com.caelum.tarefas.dao;

// imports omitidos

@Repository
public class JpaTarefaDao implements TarefaDao{

  @PersistenceContext
  EntityManager manager;
```

```
  //sem construtor

    public void adiciona(Tarefa tarefa) {
      manager.persist(tarefa);
    }

    public void altera(Tarefa tarefa) {
      manager.merge(tarefa);
    }

    public List<Tarefa> lista() {
      return manager.createQuery("select t from Tarefa t")
        .getResultList();
    }

    public Tarefa buscaPorId(Long id) {
      return manager.find(Tarefa.class, id);
    }

    public void remove(Tarefa tarefa) {
      Tarefa tarefaARemover = buscaPorId(tarefa.getId());
      manager.remove(tarefaARemover);
    }

    public void finaliza(Long id) {
      Tarefa tarefa = buscaPorId(id);
      tarefa.setFinalizado(true);
      tarefa.setDataFinalizacao(Calendar.getInstance());
    }
}
```

5. Altere a classe `TarefasController`, use a interface `TarefaDao` apenas. Assim a classe `TarefasController` fica desacoplado da implementação. Não esquece de apagar o construtor:

```
@Controller
public class TarefasController {

  @Autowired
  TarefaDao dao; //usa apenas a interface!

  //sem construtor

  //métodos omitidos, sem mudança
}
```

6. Por fim, vamos habilitar o gerenciamento de transação para qualquer método da classe `TarefasController`.

Abra a classe e use a anotação `@Transactional` em cima da classe:

```
@Transactional
@Controller
public class TarefasController {
```

7. Reinicie o Tomcat e acesse a aplicação de tarefas em http://localhost:8080/fj21-tarefas/listaTarefas.

CAPÍTULO ANTERIOR:

E agora?

PRÓXIMO CAPÍTULO:

Apêndice - VRaptor3 e produtividade na Web

Casa do Código

Twitter