01

Entendendo MVC e integrando o banco de dados com JPA 2

Transcrição

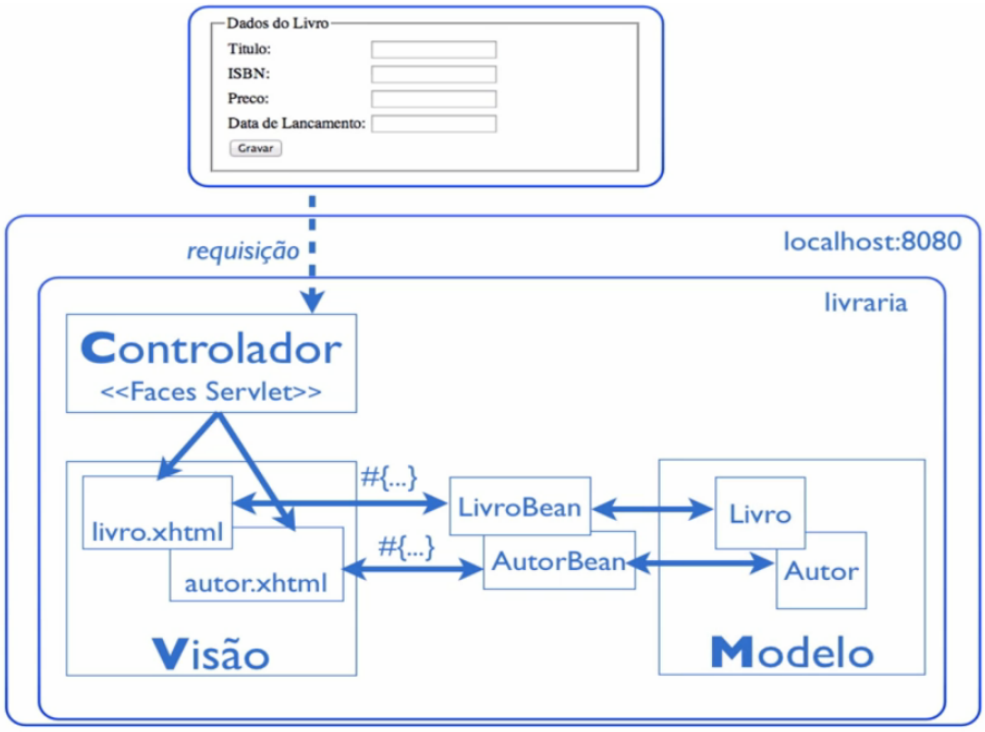
Segue o link do projeto utilizado nessa aula: Projeto livraria a importar no Eclipse (http://s3.amazonaws.com/caelum-online-public/JSF/livraria-capitulo3.zip)

Introdução

Na última aula criamos o projeto web e utilizamos os componentes da especificação JSF. Aprendemos como definir um formulário dentro de um arquivo `xhtml`. Os componentes foram associados com um `ManagedBean`, uma classe gerenciadada pelo JSF. O `ManagedBean`, por sua vez, utilizou o modelo para passar os valores.

Modelo arquitetural MVC: Model-View-Controller

Seguimos um modelo arquitetural de separação em três camadas na qual cada camada possui uma responsabilidade bem definida. A primeira camada é a do controlador, que recebe a requisição e decide qual página chamar. A segunda é a da visão (a definição da interface gráfica). E por último, temos o nosso modelo (que representa o domínio da aplicação). O `ManagedBean` é um intermediário e sua responsabilidade pode variar. Este modelo arquitetural é chamado *Model-View-Controller* ou `MVC`.



Utilizando o JPA para persistência

Para esta aula, continuaremos com o projeto livraria, mas agora com um banco de dados para realmente persistir e recuperar os dados. Utilizaremos JPA para a persistência. Como JPA não é o foco deste treinamento, já preparamos um projeto que importaremos no Eclipse:

Projeto livraria a importar no Eclipse (http://s3.amazonaws.com/caelum-online-public/JSF/livraria-capitulo3.zip)

Para importar o projeto, vamos no menu *File->Import*. No item *General*, escolhemos *Existing project into workspace*. Na próxima tela escolheremos o arquivo **livraria-capitulo3.zip**, que estará disponível para download. Após a confirmação, o Eclipse criará um novo projeto importando os arquivos do ZIP.

O projeto é igual ao projeto da aula anterior. Temos os arquivos `livro.xhtml` e `autor.xhtml`, que fazem parte do exercício. Ambos com um formulário para cadastrar o modelo, o livro ou autor, respectivamente.

Na pasta src, encontramos os *ManagedBeans*, um para cada tela. O LivroBean usa a classe Livro como modelo. Ao abrir o Livro, podemos ver uma novidade relacionada com a persistência. Mapeamos o Livro para uma tabela usando as anotações do JPA como por exemplo `@Entity` e `@Id`.

```
@Entity
public class Livro {

    @Id @GeneratedValue
    private Integer id;

    private String titulo;
    private String isbn;
    private double preco;
    private String dataLancamento;

    @ManyToMany
    private List<Autor> autores = new ArrayList<Autor>();

    //getters e setters omitidos
}
```

A tela do autor é semelhante, temos um AutorBean que usa o Autor. Dentro do método `gravar()`, em cada ManagedBean, já estamos usando o JPA através de um DAO. Podemos ver que a última linha do método instancia um DAO e chama o método `adicionar` para salvar o modelo.

```
@ManagedBean
public class LivroBean {

    //outros códigos omitidos

    public void gravar() {
        System.out.println("Gravando livro " + this.livro.getTitulo());

        if(livro.getAutores().isEmpty()) {
            throw new RuntimeException("Livro deve ter pelo menos um Autor");
        }

        new DAO<Livro>(Livro.class).adiciona(this.livro);
    }

    //outros códigos omitidos
}
```

Vamos dar uma olhada no `DAO` também. Nele utilizamos JPA. Ou seja, o `EntityManager`, que é responsável pela persistência de entidades dentro de uma transação. A classe `EntityManager` se encontra nos JARs que vieram com o projeto importado.

```
public class DAO<T> {
    private final Class<T> classe;
```

```
    public DAO(Class<T> classe) {
        this.classe = classe;
    }

    public void adiciona(T t) {
        //consegue a entity manager
        EntityManager em = new JPAUtil().getEntityManager();
        //abre transacao
        em.getTransaction().begin();

        //persiste o objeto
        em.persist(t);

        //commita a transacao
        em.getTransaction().commit();

        //fecha a entity manager
        em.close();
    }

    //outros métodos omitidos

}
```

Estamos usando JPA com Hibernate, isso significa que a maioria dos JARs são do projeto Hibernate. Além dos JARs do Hibernate, também temos na pasta `lib`, o JAR do JSF e o driver do banco MySQL.

Configurando o banco de dados

A configuração do banco está dentro da pasta `src/META-INF` com o nome `persistence.xml`. Ao abrí-lo vemos algumas configurações. Primeiro a declaração das entidades, `Autor` e `Livro`, segundo os dados da configuração do banco como login e senha, e por fim as propriedades do Hibernate.

```
<persistence ...>

    <persistence-unit name="livraria" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <class>br.com.caelum.livraria.modelo.Livro</class>
        <class>br.com.caelum.livraria.modelo.Autor</class>

        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/livrariadb
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="" />

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

Analisando a URL de conexão, vemos o banco `livrariadb`, que precisa ser criado no MySQL. Já temos o MySQL rodando nessa maquina e nos conectamos através do comando `mysql -u root`. No prompt do MySQL criaremos o banco pelo comando: `create database livrariadb`.

Populando o banco de dados através do Hibernate

Ao voltar para o Eclipse, já temos uma classe pronta que ajuda a popular o banco. A classe `PopulaBanco` apenas insere alguns livros e autores. Além disso, o Hibernate automaticamente gera as tabelas caso não existam. Vamos executar a classe e verificar o resultado no MySQL.

No prompt do MySQL, digitamos `use livrariadb;`, e para selecionar o banco e `show tables;`, para mostrar as tabelas. Foram criadas as tabelas `Autor, Livro` e o relacionamento entre os dois. Selecionaremos uma vez todos os dados do Autor e do Livro, como também os dados da tabela de relacionamento.

Testando nosso formulário

Com o banco configurado e populado, associaremos a aplicação ao Tomcat iniciando-o logo em seguida. Com o navegador aberto, testaremos o formulário do livro. Cuidado, pois o nome da aplicação mudou para livraria.

Vamos testar também o formulário do Autor. Os dados aparecem como esperado.

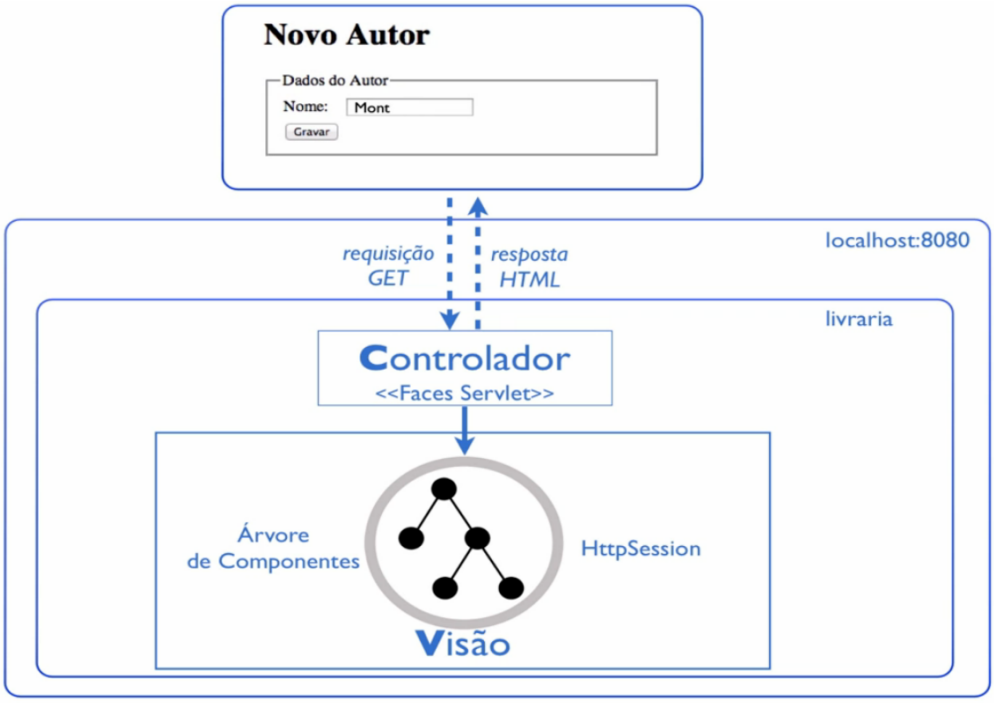
Já que fizemos a integração com o banco MySQL, inseriremos uma vez um autor pelo formulário. Ao digitar o nome do Autor e submeter o formulário, podemos verificar o resultado no banco. Selecionaremos novamente os dados - o Autor aparece!

Voltando ao navegador, percebemos que o formulário continua preenchido mesmo após ter enviado a requisição. Para entender isso melhor, vamos entrar um pouco mais a fundo no ciclo de vida dos componentes JSF.

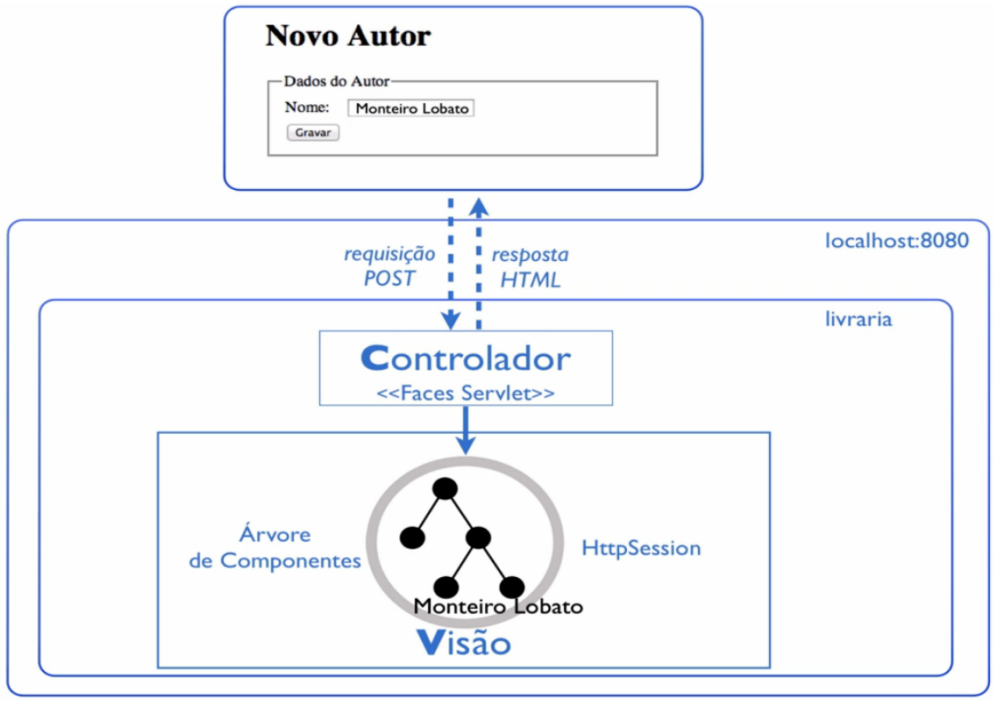
O ciclo de vida básico dos componentes JSF

Primeiramente chamamos o `autor.xhtml` pelo navegador e no lado do servidor, o controlador lê o XHTML. Como já dissemos antes, o controlador instancia os componentes declarados. O resultado é uma árvore de componentes. Nossa tela organiza-se numa estrutura hierárquica. No final, o componente `body` possui um `form` que possui um `input` e assim para frente.

É importante saber que esta árvore de componentes é criada só na primeira requisição (no primeiro GET) e depois fica guardada no sessão HTTP do usuário. O controlador pede dessa árvore em memória e devolve HTML para o navegador.



No próximo passo digitamos o nome do autor e submetemos o formulário. Agora a requisição é do tipo `POST` e o controlador recuperará apenas a árvore de componentes. Como digitamos o nome do autor no formulario, o controlador passa esse parâmetro da requisição para o componente correspondente. Ou seja, para o `inputText`. Quando o controlador pede o HTML no final, o componente também devolve o valor dele. Ou seja, o formulário continua preenchido.



Limpando o nosso formulário

Agora o nosso objetivo é limpar o formulário. Faz todo sentido limpar os inputs após a inserção do autor. Aqui é importante lembrar que ligamos o input ao modelo pela *expression language*. Quando usamos a expressão, o input passa o valor recebido para o modelo e chama o setter da classe `Autor`. Mas o input não só passa o valor, como também pede o valor do modelo antes de renderizar o HTML. A ligação é bidirecional.

Podemos provar esta ligação através do getter do modelo. Vamos abrir a classe `Autor` e sempre concatenar um `String` no getter.

Após reiniciar o servidor, e recarregar a página no navegador, percebemos que esse `String` já aparece na tela. Ou seja, o componente chamou o getter do modelo.

Para limpar o formulário então, podemos limpar o nosso modelo, mas antes vamos desfazer a alteração no getter. Para limpar o modelo instanciamos um novo autor no método `gravar()` da classe `AutorBean`. Assim o input recebe um novo autor sem nenhum valor. Novamente vamos testar no navegador. Ao preencher e submeter o formulário, o input fica vazio.

No próximo capítulo vamos completar o formulário do Livro para associar o Livro com autores.