Módulo 4

No módulo anterior evoluímos bastante nosso projeto. Experimentamos o mecanismo de tratamento de exceções e a parametrização da aplicação através do arquivo de configuração. Fizemos refatorações importantes como a criação da classe ***Aluno***. Neste módulo vamos explorar as facilidades do Demoiselle para persistência de dados e controle transacional. Sim, vamos utilizar JPA.

Extensões

O Framework Demoiselle está estruturado em um núcleo (Core) e extensões (Extensions). O núcleo provê funcionalidades básicas, como todas que vimos até agora. Porém, muitas vezes as aplicações precisam acessar bancos de dados ou serem acessadas via web, é aí que entram as extensões.

Você pode estar se perguntando: “Por que o Core não oferece funcionalidades de persistência?”. Lembre que o Core é o núcleo do Framework, então tudo que tem nele estará disponível para todas aplicações. Não é recomendável que o Framework seja impositivo. Onde estaria a liberdade de escolha? E a flexibilidade?

Como você é atento e curioso, deve pensar: “Ah! Então o ***demoiselle-junit*** é uma extensão?”. A resposta é não! O Framework, composto pelo Core e Extensions, só faz referências às especificações (Java Specification Requests - JSR). Como o JUnit não é uma especificação, ele não pode ser Extension, e sim um Component.

Dependências

Vamos agora estender nosso projeto incluindo as dependências à JPA Extension e seus agregados:

1. Abra o arquivo ***pom.xml***.
2. Inclua as dependências ao *demoiselle-jpa*, *eclipselink* e *hsqldb*.

A sessão ***dependencies*** do ***pom.xml*** vai ficar assim:

```
<dependencies>
	<dependency>
		<groupId>br.gov.frameworkdemoiselle</groupId>
		<artifactId>demoiselle-jpa</artifactId>
	</dependency>
	<dependency>
		<groupId>org.eclipse.persistence</groupId>
		<artifactId>eclipselink</artifactId>
	</dependency>
	<dependency>
		<groupId>hsqldb</groupId>
		<artifactId>hsqldb</artifactId>
	</dependency>
</dependencies>
```

A dependência ***demoiselle-jpa*** inclui a JPA Extension no seu projeto, porém é necessário escolher o fornecedor da implementação de persistência. Esta é a vantagem das JSRs, você não fica preso ao fornecedor. Vamos escolher EclipseLink, que é a implementação de referência da especificação (JSR-317), mas poderíamos escolher Hibernate ou qualquer outro aderente à especificação JPA. O HSQLDB é um banco de dados relacional, totalmente escrito em Java, que vai facilitar nossa vida.

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-4-video-1

Entidade

Na JPA as entidades refletem as tabelas no banco. Então indicaremos que a classe ***Aluno*** é uma entidade:

1. Abra a classe ***Aluno***.
2. Inclua a anotação ***@Entity*** na classe.
3. Acrescente o atributo ***@Id @GeneratedValue private Integer matricula***.
4. Gere os métodos ***get*** e ***set*** para o atributo ***matricula***.

A entidade *Aluno* vai ficar assim:

```
@Entity
public class Aluno {

	@Id
	@GeneratedValue
	private Integer matricula;

	. . .

	public Integer getMatricula() {
		return matricula;
	}

	public void setMatricula(Integer matricula) {
		this.matricula = matricula;
	}
}
```

O que fizemos? Indicamos que a entidade *Aluno* possui uma chave-primária chamada ***matricula*** gerada automaticamente e uma coluna chamada *nome*. Não é foco deste curso explorar detalhes do mapeamento objeto-relacional da JPA. Vamos seguir com a configuração do ***persistence.xml***:

1. Abra a pasta ***src/test/resources/***.
2. Crie o arquivo ***persistence.xml*** dentro de ***META-INF***.

Copie estas linhas para o seu ***persistence.xml***:

```
<?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="inscricao-teste-ds" transaction-type="RESOURCE_LOCAL">

		<class>br.gov.serpro.inscricao.entity.Aluno</class>

		<properties>
			<property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
			<property name="javax.persistence.jdbc.user" value="sa" />
			<property name="javax.persistence.jdbc.password" value="" />
			<property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:." />

			<property name="eclipselink.logging.level" value="FINE" />
			<property name="eclipselink.ddl-generation" value="create-tables" />
			<property name="eclipselink.ddl-generation.output-mode" value="database" />
		</properties>
	</persistence-unit>

</persistence>
```

O ***persistence.xml*** é o arquivo de configuração da JPA que contém informações sobre a conexão. O nosso banco HSQLDB armazenará temporariamente os dados na memória, como especificado na propriedade *javax.persistence.jdbc.url*. Utilizaremos a funcionalidade de criação automática das estruturas do banco de dados (*DDL Generation*) do EclipseLink.

Tudo configurado, bola pra frente!

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-4-video-2

Gerenciador de entidades

Abra a classe ***Turma***, vamos persistir os dados no banco.

1. Acrescente o atributo ***@Inject private EntityManager em***.
2. Remova o atributo ***alunosMatriculados***.
3. Crie o método ***private List<Aluno> obterAlunosMatriculados()*** que obtém os dados persistidos.
4. Ajuste a validação no método ***matricular*** que fazia referência ao atributo removido.
5. Modifique o método ***matricular*** para persistir no banco de dados.
6. Modifique o método ***estaMatriculado*** para verificar se o aluno está persistido.
7. Importe a classe ***javax.persistence.Query***

As modificações vão deixar a classe *Turma* assim:

```
...

@Inject
private EntityManager em;

public void matricular(Aluno aluno) {
	if (... obterAlunosMatriculados().size() == config.getCapacidadeTurma()) {
		...
	}

	em.getTransaction().begin();
	em.persist(aluno);
	em.getTransaction().commit();
	...
}

public boolean estaMatriculado(Aluno aluno) {
	return obterAlunosMatriculados().contains(aluno);
}

private List<Aluno> obterAlunosMatriculados() {
	return em.createQuery("select a from Aluno a").getResultList();
}

...
```

Fique atento, a injeção do ***EntityManager*** não é suportada nativamente pela JPA. O Demoiselle provê esta funcionalidade através da JPA Extension. Se não fosse pela injeção, você precisaria escrever algumas linhas de código para obter um ***EntityManager***.

O método ***persist*** do ***javax.persistence.EntityManager*** converte o objeto no comando ***insert*** do banco de dados. Para qualquer operação de atualização, deve-se iniciar (***begin***) e finalizar (***commit***) a transação. Na operação de consulta utiliza-se uma linguagem similar ao SQL, porém orientada a objetos.

Por fim, rode os testes e viva o verde! Veja no console os comandos SQL que você não escreveu.

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-3-video-3

Controle transacional

Controlar a transação manualmente é chato! Portanto delegaremos esta responsabilidade para o Demoiselle:

1. Abra a classe ***Turma***.
2. Anote o método ***matricular*** com ***@Transactional***.
3. Remova as linhas ***em.getTransaction().begin()*** e ***em.getTransaction().commit()***.

O método *matricular* ficará assim:

```
@Transactional
public void matricular(Aluno aluno) {
	...
	em.persist(aluno);
	...
}
```

A anotação ***@br.gov.frameworkdemoiselle.transaction.Transactional*** faz com que o Demoiselle abra transação antes da execução do ***matricular*** e finalize após o seu término. Caso ocorra alguma exceção, será feito ***rollback*** automaticamente.

A estratégia ***br.gov.frameworkdemoiselle.transaction.JPATransaction*** delega o controle para o ***EntityManager*** de forma similar a que fizemos manualmente, porém preocupando-se com mais detalhes, como o ***rollback*** por exemplo. A ***JPATransaction*** é mais uma das facilidades providas pela JPA Extension.

Rode os testes, agora vai passar! Confira as mensagens do controle transacional no console.

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-4-video-4

Exceção da aplicação

E se não quisermos que o ***rollback*** ocorra automaticamente? Podemos mudar o comportamento do gerenciador de transação:

1. Abra a classe ***TurmaException***.
2. Anote-a com ***@ApplicationException(rollback = false)***.

A anotação ***@br.gov.frameworkdemoiselle.exception.ApplicationException*** deve ser utilizada para identificar as exceções da aplicação. Além do atributo ***rollback***, existem outras facilidades que serão exploradas nos módulos seguintes.

Rode os testes e analise o console. Perceba que não ocorre mais ***rollback***. Agora que já experimentou, coloque o valor correto no atributo:

1. Abra a classe ***TurmaException***.
2. Ajuste o atributo para ***rollback = true***.

A classe ***TurmaException*** vai ficar assim:

```
@ApplicationException(rollback = true)
public class TurmaException extends RuntimeException {
}
```

Use a criatividade e refatore o método ***matricular*** da classe ***Turma*** para explorar melhor o ***rollback***. Não modifique os testes, ao final do experimento tem que ficar tudo verde.

Confira o vídeo de demonstração clicando no link abaixo:

http://www.frameworkdemoiselle.gov.br/documentacaodoprojeto/manuais-e-tutoriais/tutorial-da-versao-2-2-3-0/videos/modulo-4-video-5

Retrospectiva

Aprendemos como o Demoiselle Framework está dividido (Core e Extensions) e o porquê. Descobrimos a diferença entre Extensions e Components. Partimos para a configuração da persistência da nossa aplicação utilizando a extensão *demoiselle-jpa*. Fizemos ajustes no projeto para habilitar o JPA.

Modificamos a classe *Turma* para persistir os dados no banco. Experimentamos o mecanismo de controle transacional do Demoiselle, verificando o funcionamento do *rollback* com exceções da aplicação.

No próximo módulo focaremos na arquitetura da aplicação, dando uma cara mais profissional ao nosso projeto. Faremos diversas refatorações para adequá-lo à arquitetura em camadas.