Módulo 5

No módulo anterior adaptamos nosso projeto para persistir as informações no banco de dados utilizando as facilidades da extensão demoiselle-jpa. Experimentamos o controle transacional do Framework utilizando a estratégia ***JPATransaction***.

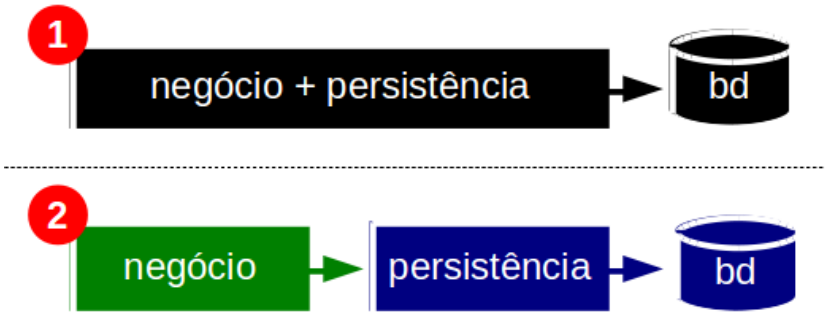
O Demoiselle propõe diversos padrões e arquiteturas para sua aplicação. A ideia não é obrigá-lo a fazer deste ou daquele jeito, e sim apresentar uma sugestão abrangente e facilmente adaptável às especificidades de cada aplicação.. Cada aplicação pode ter necessidades específicas, por isto é necessário que o Framework seja flexível.

Considerar que seu projeto é diferente de tudo que existe, antes de fazer uma análise mais aprofundada, pode ser um grande equívoco. Constata-se que na maioria das as aplicações corporativas possuem arquiteturas similares, variando apenas em pequenos detalhes. Neste módulo vamos refatorar nosso projeto para abordar estas questões.

Camadas

Por enquanto a nossa aplicação não possui divisão em camadas. Note que a classe ***Turma*** mistura validações de negócio com persistência em banco. Em pequenas aplicações isto não chega atrapalhar, mas, quando o projeto ganha maiores proporções, a falta de organização impacta negativamente na manutenibilidade e legibilidade do código-fonte.

Na figura abaixo ① representa o estágio atual da nossa aplicação. A partir de agora iremos fazer os ajustes necessários para criar uma estrutura ilustrada por ②.



Vamos refatorar nossa aplicação para adequá-la à arquitetura em camadas:

1. Abra a classe ***Turma***.
2. Renomeia-a para ***TurmaBC*** utilizando as ferramentas de refatoração do Eclipse.
3. Mova a nova classe para o pacote *br.gov.serpro.inscricao****.business***.
4. Substitua a anotação *@Controller* por ***@BusinessController***.

O sufixo sugerido (BC) significa Controlador de Negócio (*Business Controller*). Um objeto de negócio representa as operações que podemos fazer sobre uma determinada entidade. A classe ***TurmaBC*** sintetiza o que podemos fazer numa turma, tal como matricular ou verificar se um aluno está matriculado.

O estereótipo *@BusinessController* indica que a classe ***TurmaBC*** é um controlador específico da camada de negócio. Você lembra por que utilizamos a anotação *@Controller*? Não?! Então recorra ao módulo 3 para refrescar a memória.

Com as mudanças a classe ***TurmaBC*** vai ficar assim:

```
@BusinessController
public class TurmaBC {
...
}
```

Rode os testes, tem que passar.

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-5-video-1

Agora vamos transpor os detalhes de acesso ao banco para a camada de persistência.

1. Crie a classe ***AlunoDAO*** no pacote *br.gov.serpro.inscricao****.persistence***.
2. Anote-a com ***@PersistenceController***.
3. Crie os métodos *public void* ***insert****(Aluno aluno)* e *public List<Aluno>* ***findAll****()*.
4. Mova o atributo *@Inject private* ***EntityManager em*** do BC para o DAO.
5. Mova a linha de código ***em.persist(aluno);*** do método ***matricular*** para o método ***insert***.
6. Em seguida mova a linha ***return em.createQuery("select a from Aluno a").getResultList();*** do método ***obterAlunosMatriculados*** para o método ***findAll***.
7. Crie o atributo *@Inject private* ***AlunoDAO alunoDAO*** no BC.
8. Adicione a linha **alunoDAO.insert(aluno);** ao método ***matricular***.
9. E adicione a linha ***return alunoDAO.findAll();*** ao método ***obterAlunosMatriculados***.

Basicamente o que fizemos até agora foi separar as responsabilidades. O BC é responsável pelas regras de negócio e o DAO pela persistência, via *EntityManager*. Agora o BC desconhece onde os dados estão sendo persistidos. O acrônimo DAO significa Objeto de Acesso aos Dados (*Data Access Objects*). Estamos começando a isolar as coisas!

Ao final do processo, os métodos alterados da ***TurmaBC*** vão ficar assim:

```
@Transactional
public void matricular(Aluno aluno) {
	if (estaMatriculado(aluno) ||
		obterAlunosMatriculados().size() >= config.getCapacidadeTurma()) {
		throw new TurmaException();
	}
	alunoDAO.insert(aluno);
	logger.info(bundle.getString("matricula.sucesso", aluno.getNome()));
}

private List<Aluno> obterAlunosMatriculados() {
	return alunoDAO.findAll();
}
```

E a classe ***AlunoDAO***, assim:

```java
@PersistenceController
public class AlunoDAO {

	@Inject
	private EntityManager em;

	public void insert(Aluno aluno){
		em.persist(aluno);
	}

	public List<Aluno> findAll(){
		return em.createQuery("select a from Aluno a").getResultList();
	}
}
```

Resumo da ópera: tornamos uma aplicação monolítica ① em uma estrutura modularizada ②. “Dai a César o que é de César”.

Para garantir que a refatoração foi bem sucedida, rode novamente os testes. Tem que continuar 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-5-video-2

Simplificando as coisas

É verdade que nossa aplicação está bem simples. É de propósito!

A intenção não é criar regras complexas, explorar o JPA, criar relacionamentos no banco de dados ou dificultar as coisas. Tudo aqui tem uma razão de existir.

Se você está com a mão coçando para criar novas entidades, acrescentar relacionamentos ou possibilitar a matrícula de alunos em diversas turmas, controle a ansiedade. Vamos manter a estrutura simples e suficiente para o que precisamos neste momento.

Para as regras do nosso projeto, o ato de matricular um aluno nada mais é do que persisti-lo numa tabela do banco de dados. Se o aluno está lá, convencionamos que ele está matriculado.

CRUD

Você sabe o que é CRUD? É a sigla para Create, Read, Update e Delete. Em outras palavras, é

como chamamos as telas de cadastro básico da aplicação. Vamos conhecer algumas facilidades que o Framework provê:

1. Abra a classe ***AlunoDAO***.
2. Herde de ***JPACrud<Aluno, Integer>***.
3. Apague todos os atributos e métodos.

A classe ***AlunoDAO*** vai ficar assim:

```
@PersistenceController
public class AlunoDAO extends JPACrud<Aluno, Integer>{
	private static final long serialVersionUID = 1L;
}
```

Você deve estar se perguntando: “O que significa <Aluno, Integer>”? Isto é um recurso da linguagem conhecido como *Generics*. Assim é possível determinar que *AlunoDAO* é CRUD para a entidade *Aluno*, a qual possui uma chave-primária do tipo *Integer*.

A classe ***AlunoDAO*** herdará os seguintes métodos previamente implementados em *br.gov.frameworkdemoiselle.template.JPACrud*:

- *Aluno load(Integer id)*
- *void insert(Aluno aluno)*
- *void update(Aluno aluno)*
- *void delete(Integer id)*
- *List<Aluno> findAll()*

A classe *JPACrud* é mais uma funcionalidade da extensão *demoiselle-jpa*. Rode os testes, vai passar!

Conceitualmente não é recomendado que uma classe de negócio referencie uma classe de persistência de outra entidade. Na prática, TurmaBC não deve referenciar AlunoDAO. Vamos fazer os devidos ajustes:

1. Crie a classe ***AlunoBC*** no pacote *br.gov.serpro.inscricao****.business***.
2. Anote-a com ***@BusinessController***.
3. Herde de ***DelegateCrud<Aluno, Integer, AlunoDAO>***.
4. Abra a classe *TurmaBC*.
5. Remova o atributo *@Inject private* ***AlunoDAO alunosDAO***.
6. Acrescente o atributo *@Inject private* ***AlunoBC alunoBC***.
7. Modifique os métodos que estão utilizando ***alunoDAO*** para utilizarem ***alunoBC***.

A classe ***AlunoBC*** vai ficar assim:

```
@BusinessController
public class AlunoBC extends DelegateCrud<Aluno, Integer, AlunoDAO>{
	private static final long serialVersionUID = 1L;
}
```

A modificação em ***TurmaBC*** será a seguinte:

```
@BusinessController
public class TurmaBC {

	...

	@Inject
	private AlunoBC alunoBC;

	@Transactional
	public void matricular(Aluno aluno) {
		...
		alunoBC.insert(aluno);
		...
	}

	public List<Aluno> obterAlunosMatriculados() {
		return alunoBC.findAll();
	}
}
```

A classe ***AlunoBC*** herdará de *br.gov.frameworkdemoiselle.template.DelegateCrud*, que delega as chamadas para outra classe CRUD, no nosso caso *AlunoDAO*.

Rode os testes, vai passar!

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-5-video-3

Inicializadores e finalizadores

Muitas vezes precisamos executar tarefas na inicialização ou finalização da aplicação, tais como:

- Gerar carga inicial
- Limpar arquivos temporários
- Consultar informações de outro sistema
- Enviar e-mails
- Gerar informações no *log*

Para capturar os eventos de inicialização e finalização da aplicação, o Demoiselle disponibiliza as anotações ***@Startup*** e ***@Shutdown***. Para exemplificar, vamos gerar mensagens no *log* quando a aplicação iniciar e finalizar:

1. Abra a classe ***TurmaBC***.
2. Crie o método *@Startup public void* ***iniciar()***.
3. Gere uma mensagem informativa utilizando o *logger*.

As modificações em ***TurmaBC*** serão estas:

```
@BusinessController
public class TurmaBC {

	...

	@Startup
	public void iniciar(){
		logger.info("Iniciando ...");
	}
}
```

Rode os testes, que continuarão passando, e observe a mensagem no console.

18:00:54,240 INFO [business.TurmaBC] Iniciando …

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-5-video-4

Retrospectiva

Aprendemos sobre arquitetura em camadas. Modularizamos a classe *Turma* e delegamos a manutenção do aluno para as classes AlunoBC e Aluno*DAO*. Renomeamos a classe *Turma* para *TurmaBC*. Experimentamos as facilidades providas pelas abstrações *JPACrud* e *DelegateCrud*. Criamos um método inicializador em nosso projeto.

Nossa aplicação está ficando cada vez mais robusta. No próximo módulo, iremos garantir o controle de acesso utilizando os recursos providos pelo Demoiselle. Vamos implementar nosso autenticador e autorizador personalizado.