# Módulo 6

No módulo anterior aprendemos um pouco sobre arquitetura em camadas e refatoramos nossa aplicação. Criamos objetos da camada de negócio, *Business Controllers*, e da camada de persistência, *Persistence Controllers*. Criamos alguns CRUD utilizando as facilidades providas pelo Demoiselle.

Neste módulo iremos incorporar uma solução que praticamente todas as aplicações corporativas necessitam, o controle de acesso. Vamos conhecer a funcionalidade do Framework que auxilia a implementação de mecanismos de autenticação e autorização da aplicação.

## *Autorização*

A primeira coisa que vamos fazer é marcar os pontos críticos da nossa aplicação:

1. Abra a classe ***TurmaBC***.
2. Anote o método ***matricular*** com *@RequiredPermission(resource = "turma", operation = "matricular").*
3. Anote o método ***estaMatriculado*** com *@RequiredPermission(resource = "turma", operation = "consultar").*

A classe ***TurmaBC*** ficará assim:

```
...
@Transactional
@RequiredPermission(resource = "turma", operation = "matricular")
public void matricular(Aluno aluno) {
    ...
}

@RequiredPermission(resource = "turma", operation = "consultar")
private  List<Aluno> obterAlunosMatriculados() {
    ...
}
```

A anotação *@br.gov.frameworkdemoiselle.security.RequiredPermission* define um ponto de checagem de segurança. O Framework só permitirá que os métodos anotados sejam executados caso o usuário autenticado tenha permissão. O atributo *resource* indica o recurso protegido e *operation* a ação executada sobre o recurso. Poderíamos ter escolhido qualquer nome para o recurso ou operações.

Definidos os pontos de checagem, agora implementaremos a classe decisora:

1. Crie a classe Autorizador no pacote br.gov.serpro.inscricao.security implementando a interface br.gov.frameworkdemoiselle.security.Authorizer
2. Crie os métodos exigidos pela interface.
3. Implemente os métodos para que retornem true.
4. Inclua a classe autorizadora na propriedade do arquivo demoiselle.properties (frameworkdemoiselle.security.authorizer.class)

A classe ***Autorizador*** ficará assim:

```
import br.gov.frameworkdemoiselle.security.Authorizer;
public class Autorizador implements Authorizer{

	private static final long serialVersionUID = 1L;
    @Override
    public boolean hasPermission(String arg0, String arg1) {
        return true;
    }

    @Override
    public boolean hasRole(String arg0) {
        return true;
    }
}
```

Nosso autorizador será invocado automaticamente pelo Demoiselle. Se retornarmos *true* o Framework permitirá acesso ao recurso, caso contrário não. Como não utilizamos a anotação *@RequiredRole*, o método ***hasRole*** não será invocado. Focaremos então no ***hasPermission***, que responde pela anotação *@RequiredPermission*.

Os métodos decisores podem ser implementados de diversas maneiras. É possível decidir com base em arquivo XML, banco de dados, Web Service, consulta a outro sistema, etc. As possibilidades são infinitas. Para facilitar, simplesmente retornaremos *true*.

O arquivo demoiselle.properties terá o seguinte conteúdo:

```
frameworkdemoiselle.security.authorizer.class=br.gov.serpro.inscricao.security.Autorizador
```

Execute os testes, você verá o seguinte erro:

br.gov.frameworkdemoiselle.DemoiselleException: Nenhum mecanismo de autenticação foi definido. Para utilizar SecurityContext é preciso definir a propriedade frameworkdemoiselle.security.authenticator.class como mecanismo de autenticação desejado no arquivo demoiselle.properties.

***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-6-video-1

## *Autenticação*

Para utilizar funcionalidades de autorização, é preciso definir o mecanismo de autenticação. É isto que vamos fazer agora:

1. Crie a classe Autenticador no pacote br.gov.serpro.inscricao.security implementando a interface br.gov.frameworkdemoiselle.security.Authenticator.
2. Crie os métodos exigidos pela interface.
3. No método authenticate() retorne true.
4. No método getUser() retorne um novo usuário.
5. Defina no arquivo demoiselle.properties a classe de autenticação na propriedade frameworkdemoiselle.security.authenticator.class.

O ***Autenticador*** ficará assim:

```
public class Autenticador implements Authenticator{

	private static final long serialVersionUID = 1L;

	@Override
	public boolean authenticate() {
		return true;
	}

	@Override
	public User getUser() {
		return new User() {

			private static final long serialVersionUID = 1L;

			@Override
			public void setAttribute(Object arg0, Object arg1) {

			}

			@Override
			public String getId() {
				return null;
			}

			@Override
			public Object getAttribute(Object arg0) {
				return null;
			}
		};
	}

	@Override
	public void unAuthenticate() {
	}
}
```

Os métodos do autenticador serão invocados automaticamente pelo Framework. O ***authenticate*** reponderá *true*, indicando que o processo de autenticação foi bem sucedido. O ***getUser*** devolverá uma instância de *User*, representando o usuário autenticado.

Em momento algum invocamos o processo de *logon*, então vamos ajustar os testes:

1. Abra a classe ***TurmaTest***.
2. Acrescente o atributo *@Inject private* ***SecurityContext securityContext***.
3. Crie o método *@Before public* ***void setUp()*** que será invocado pelo JUnit automaticamente antes de cada teste.
4. Implemente o método *setUp* invocando ***securityContext.login()***.

As modificações na classe ***TurmaTest*** serão as seguintes:

```
...
@Inject
private SecurityContext securityContext;

@Before
public void setUp(){
        securityContext.login();
}
...
```

O arquivo demoiselle.properties terá o seguinte conteúdo:

```
frameworkdemoiselle.security.authorizer.class=br.gov.serpro.inscricao.security.Autorizador
frameworkdemoiselle.security.authenticator.class=br.gov.serpro.inscricao.security.Autenticador
```

Você nunca deve invocar o ***Autenticador*** diretamente, isto é responsabilidade do Demoiselle. Para acessar as funcionalidades de segurança utilize a interface *br.gov.frameworkdemoiselle.security.SecurityContext*, que dispõe dos seguintes métodos:

- *void login()*
- *void logout()*
- *boolean isLoggedIn()*
- *boolean hasPermission(String resource, String operation)*
- *boolean hasRole(String role)*
- *User getUser()*

Execute os testes novamente, agora 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-6-video-2

## *Credenciais*

Para o processo de autenticação ser efetivo, é fundamental checar as credenciais do usuário. Nunca vi efetuar o *logon* sem informar senha, certificado, impressão digital ou qualquer outra credencial. Você já viu?!

Evoluiremos nosso processo de autenticação exigindo o nome do usuário e a senha:

1. Crie a classe ***Credenciais*** no pacote *br.gov.serpro.inscricao****.security*** implementando a interface *Serializable*.
2. Anote-a com *@SessionScoped*.
3. Crie os atributos *private* ***String nome*** e *private* ***String senha***.
4. Gere os métodos *get* e *set* para os atributos.
5. Abra a classe ***TurmaTest***.
6. Acrescente o atributo *@Inject private* ***Credenciais credenciais***.

A classe ***Credenciais*** ficará assim:

```
@SessionScoped
public class Credenciais implements Serializable{
	private static final long serialVersionUID = 1L;

	private String nome;
	private String senha;

	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public String getSenha() {
		return senha;
	}
	public void setSenha(String senha) {
		this.senha = senha;
	}
}
```

A anotação *@javax.enterprise.context.SessionScoped*, pertencente ao CDI, mantém a instância na sessão do usuário. Como precisamos transportar as credenciais, é importante que a instância não seja recriada a cada nova injeção. Para que o armazenamento seja feito automaticamente pelo CDI, é preciso implementar a interface *java.io.Serializable*, caso contrário ocorrerá erro na injeção.

No método ***setUp*** de *TurmaTest*, defina o nome do usuário “secretaria” e a senha “segredo”. A classe ficará assim:

```
@Inject
private Credenciais credenciais;

@Before
public void setUp(){
	credenciais.setNome("secretaria");
	credenciais.setSenha("segredo");
	securityContext.login();
}
```

Para finalizar, faremos os ajustes no autenticador:

1. Abra a classe ***Autenticador***.
2. Acrescente o atributo *@Inject private* ***Credenciais credenciais***.
3. Implemente a lógica da autenticação.

A classe ***Autenticador*** ficará assim:

```
import javax.inject.Inject;
import br.gov.frameworkdemoiselle.security.Authenticator;
import br.gov.frameworkdemoiselle.security.User;
import br.gov.frameworkdemoiselle.util.ResourceBundle;

public class Autenticador implements Authenticator{
	private static final long serialVersionUID = 1L;

	@Inject private
	Credenciais credenciais;

	 @Inject
	 private ResourceBundle bundle;

	@Override
	public boolean authenticate() {

	 boolean autenticado = false;
	 if (credenciais.getNome().equals("secretaria") && credenciais.getSenha().equals("segredo")) {
				autenticado = true;
			} else {
				throw new RuntimeException(bundle.getString("usuarioNaoAutenticado"));
			}
			return autenticado;
	 }

    @Override
    public User getUser() {
	 return new User() {
		private static final long serialVersionUID = 1L;

	    @Override
	    public void setAttribute(Object arg0, Object arg1) {

	    }

	    @Override
	    public String getId() {
		return null;
	    }

	    @Override
	    public Object getAttribute(Object arg0) {
		return null;
	    }
	};
    }

    @Override
    public void unAuthenticate() {
    }
}
```

Rode os testes e veja que eles estão passando.

Não se esqueça de incluir a propriedade no arquivo message.properties:

```
usuarioNaoAutenticado=Erro ao validar usu\u00E1rio
```

***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-6-video-3

## *Componentes*

Se você quiser poupar tempo com implementação do mecanismo para controle de acesso, acesse a documentação de referência dos componentes e veja o que o Demoiselle preparou para você.

## *Retrospectiva*

Aprendemos como o Demoiselle trata alguns conceitos relacionados ao controle de acesso. Implementamos um autenticador e um autorizador personalizado para nossa aplicação. Conhecemos a porta de acesso para as funcionalidades de segurança do Framework, a interface *SecurityContext*. Experimentamos também o gerenciamento de escopo do CDI com a classe *Credenciais*.

No próximo módulo, vamos converter nosso projeto em uma aplicação *web* utilizando JSF e o servidor de aplicações JBoss AS.