## Framework Demoiselle

# Guia de Referência

**Cleverson Sacramento**

**Danilo Viana**

**Emerson Oliveira**

**Emerson Saito**

**Luciano Borges**

**Marlon Carvalho**

**Rodrigo Hjort**

**Serge Rehem**

**Thiago Mariano**

**Wilson Guimarães**

## Sobre o Guia de Referência - versão 2.5.0-RC1

Este documento tem como objetivo ser um guia de referência destinado a todos que desejem conhecer melhor o *Framework Demoiselle 2.5.0-RC1* e suas funcionalidades.

### Nota

Esta documentação refere-se à versão 2.5.0-RC1 do *Demoiselle Framework* e pode diferir significativamente das versões anteriores.

# Introdução

## 1.1. O que é o Demoiselle?

O *Demoiselle Framework* implementa o conceito de *framework integrador*. Seu objetivo é facilitar a construção de aplicações minimizando tempo dedicado à escolha e integração de frameworks especialistas, o que resulta no aumento da produtividade e garante a manutenibilidade dos sistemas. Disponibiliza mecanismos reusáveis voltados as funcionalidades mais comuns de uma aplicação (arquitetura, segurança, transação, mensagem, configuração, tratamento de exceções, etc).

O nome *Demoiselle* é uma homenagem à série de aeroplanos construídos por Santos Dummont entre 1907 e 1909. Também conhecido como *Libellule*, as Demoiselles foram os melhores, menores e mais baratos aviões da sua época. Como sua intenção era popularizar a aviação com fabricação em larga escala, o inventor disponibilizou os planos em revistas técnicas para qualquer pessoa que se interessasse.

O *Demoiselle Framework* usa a mesma filosofia do “Pai da Aviação”, tendo sido disponibilizado como software livre em abril de 2009, sob a licença livre LGPL version 3. Mais informações no portal *"www.frameworkdemoiselle.gov.br* [http://www.frameworkdemoiselle.gov.br]

## 1.2. Sobre a versão 2

O principal objetivo da versão 2 do *Demoiselle Framework* é a completa aderência à especificação *JSR 316: Java Platform, Enterprise Edition 6* [http://jcp.org/en/jsr/detail?id=316], ou simplesmente *Java EE 6*. Para saber mais, recomendamos os links *Introducing the Java EE 6 Platform* [http://www.oracle.com/technetwork/articles/javaee/javaee6overview-141808.html] e *As novidades da JEE 6* [http://cleversonsacramento.wordpress.com/2010/08/15/as-novidades-da-jee-6/].

Esta documentação é referente às especificações da versão 2 cadastradas no *tracker* [https://sourceforge.net/apps/mantisbt/demoiselle/changelog_page.php] do Demoiselle, as quais foram publicamente discutidas no fórum *demoiselle-proposal* [https://sourceforge.net/apps/phpbb/demoiselle/viewtopic.php?f=35&t=63&start=0].

Os capítulos a seguir entram em detalhes sobre cada uma das principais funcionalidades do framework. Tenha uma boa leitura!

# Arquitetura

## 2.1. Estrutura

Visando uma melhor modularização, o Demoiselle está dividido por funcionalidades. Isto significa que o framework não é monolítico, no qual todas as suas funcionalidades estão contidas em um único pacote. Aliás, esta estratégia não é a mais indicada, pois projetos com um propósito específico, que não necessitam de persistência ou interface Web, por exemplo, teriam dependências desnecessárias. Assim, o Demoiselle é separado em Core, Extensões e Componentes.

O Core do Demoiselle contém aquelas funcionalidades comuns a todas as extensões e aplicações. O core é simples, leve e formado majoritariamente por interfaces e poucas implementações. O Core é a base do framework, sem ele, as extensões e a própria aplicação não funcionariam.

As Extensões, como o próprio nome sugere, estendem o Core com funcionalidades extras e bem específicas a um domínio ou tecnologia. Neste contexto, caso sua aplicação necessite de persistência com JPA, o framework fornecerá facilidades para você; contudo, estas funcionalidades não estão no Core. Para este propósito existem as extensões como a `demoiselle-jpa`, por exemplo. Cabe destacar que as extensões não possuem vida própria, pois estão diretamente ligadas ao núcleo do framework, inclusive o ciclo de vida das extensões está totalmente acoplado ao do Core.

Já os Componentes são artefatos separados e que, portanto, não são dependentes diretamente do Core. Aliás, os Componentes podem até mesmo existir sem referenciar o Core. Desta forma, o seu ciclo de vida é totalmente independente do Core e Extensões. Um componente não precisa, necessariamente, estender o comportamento do Core, mas permitir disponibilizar novas funcionalidades ao usuário. Outra diferença importante é que, diferente de Core e Extensões, os Componentes não necessariamente são aderentes a alguma especificação. Um exemplo é o `demoiselle-validation`.

## 2.2. Pacote Internal

As boas práticas de programação nos alertam para que nunca sejamos dependentes de implementações, mas sempre de interfaces ou, como alguns costumam dizer, “depender de contratos”. Portanto a sua aplicação precisará apenas depender das interfaces que o Demoiselle provê. As implementações específicas e internas do Framework serão injetadas automaticamente pelo CDI.

**Dica**

As classes do pacote `internal` nunca devem ser referenciadas pela sua aplicação!

Qual o motivo de toda esta explicação? Os programadores mais curiosos irão encontrar classes do framework que estão inseridas no pacote `br.gov.frameworkdemoiselle.internal`. As classes deste pacote não devem ser usadas diretamente pela sua aplicação, caso contrário você estará acoplando-a com a implementação interna do Framework. A equipe do Demoiselle possui atenção especial quanto às suas interfaces (contratos) e não irá modificá-las sem antes tornar públicas as mudanças. Contudo, tudo que consta no pacote `br.gov.frameworkdemoiselle.internal` pode sofrer mudanças repentinas. Se você referenciar tais classes internas, a sua aplicação pode deixar de funcionar ao atualizar a versão do Demoiselle.

## 2.3. Arquitetura das aplicações

É importante reforçar que o Demoiselle não obriga nenhum tipo de arquitetura para as aplicações, que podem ser constituídas por quantas camadas forem necessárias. Contudo, é prudente não exagerar! Para quem não sabe

por onde começar, sugerimos uma arquitetura e padrões largamente utilizados pelo mercado, de forma a facilitar a manutenção e para melhor modularização de seu projeto.

Usualmente, as aplicações são constituídas por pelo menos três camadas, desta forma é comum separar as lógicas de apresentação, regras de negócio e persistência. O Demoiselle já fornece estereótipos que visam tornar esta separação mais clara, respectivamente: `@ViewController`, `@BusinessController` e `@PersistenceController`. Maiores detalhes sobre cada anotação serão dados no decorrer desta documentação.

Cabe destacar que estamos falando de uma macro-visão arquitetural. Cada camada pode ser organizada internamente da melhor forma possível, ou conforme os padrões vigentes no mercado. Para uma aplicação Swing, por exemplo, o padrão de projeto *Presentation Model* é bastante indicado. Para aplicações Web, os frameworks especialistas geralmente aplicam o padrão MVC (Model/View/Controller).

# Parent POM

O Demoiselle faz uso da solução proposta pelo Apache Maven para diversas fases do desenvolvimento de software. O artefato principal do Maven é o pom.xml, que é o arquivo XML que contém todas as informações necessárias para a ferramenta gerenciar o projeto, entre as quais está o gerenciamento de dependências (bibliotecas), build do projeto, etc. Mas é muito comum que vários projetos, vinculados ou não, utilizem muitas configurações em comum. Para o "reaproveitamento" dessas configurações, evitando a cópia de texto, o Maven provê dois tipos de estratégia:

-Por *herança* [http://maven.apache.org/pom.html#Inheritance] ou *agregação* [http://maven.apache.org/pom.html#Aggregation].

No Demoiselle 2 a estratégia adota foi também o da herança. E o termo usado no Demoiselle para identificar essa estratégia é que chamamos de Parent POM.

Seguindo esse conceito, foram criados alguns arquivos (pom.xml) e também disponibilizados no repositório Maven do Demoiselle, que facilitam a configuração dos projetos, e inclusive para o desenvolvimento do próprio Demoiselle. Os arquivos gerados foram divididos em perfis, para que o desenvolvedor possa escolher qual usar de acordo com o tipo de aplicação que está desenvolvendo. Assim, a alteração no pom.xml da aplicação será a minima possível. Outra vantagem é que as bibliotecas apontadas como dependências são testadas pela equipe do Demoiselle, o que evita eventuais incompatibilidades.

**Dica**

Para excluir uma dependência desnecessária vinda do Parent, utilize a tag Exclusions.

## 3.1. demoiselle-minimal-parent

Configurações úteis para todas as aplicações que utilizam o framework. O ideal é que toda aplicação que utiliza o Demoiselle herde deste POM ou de uma de suas especializações.

## 3.2. demoiselle-se-parent

Especialização do POM mínimo, contendo configurações úteis para todas as aplicações Desktop que utilizam o framework, mas sem definição da camada de apresentação que será utilizada.

## 3.3. demoiselle-servlet-parent

Especialização do POM mínimo, contendo profiles para Tomcat 6, Tomcat 7, GAE, Glassfish 3, JBoss 6 e JBoss 7, e outras configurações úteis para todas as aplicações JEE6/Web que utilizam o Demoiselle, mas sem a definição de qual camada de apresentação utilizará. Entre as dependências referenciadas por este POM está a extensão *demoiselle-servlet*.

## 3.4. demoiselle-jsf-parent

Especialização do POM demoiselle-servlet-parent, contendo configurações úteis e necessárias para todas as aplicações que utilizarão a tecnologia JSF2 para camada de apresentação. Entre as dependências referênciadas por este POM está obviamente a extensão *demoiselle-jsf*.

## 3.5. demoiselle-archetype-parent

Contém configurações comuns a todos os projetos geradores de arquétipos.

# Arquétipos

O projeto Demoiselle recomenda e usa a ferramenta *Apache-Maven* [http://maven.apache.org/], para gerenciamento do ciclo de vida do desenvolvimento de projeto. Baseada nesta ferramenta, além do fornecimento dos POMs Parentes, também fornece as estruturas chamadas *arquétipos* [http://maven.apache.org/archetype/maven-archetype-plugin/] para facilitar a criação de aplicações, garantido a estrutura recomendada pelo framework e o conceito de gerenciamento do próprio Maven. Atualmente estão disponíveis os seguintes artefatos:

## 4.1. demoiselle-minimal

Fornece um conjunto mínimo de artefatos para criar uma aplicação Java, utiliza o Demoiselle-Minimal-Parent, sendo útil quando os outros arquétipos disponíveis não se enquadram nas características do projeto a ser criado.

## 4.2. demoiselle-jsf-jpa

Útil para os projetos que precisam de uma arquitetura que utilize as tecnologias JSF e JPA, é baseado no *demoiselle-jsf-parent* e já traz uma estrutura padrão de pacotes e todas as dependências necessárias para rodar a aplicação. Ao usar este arquétipo, você terá uma pequena aplicação de Bookmarks já pronta para rodar. Para isto, basta instalá-la em um servidor da sua preferência! Para mais detalhes sobre esta aplicação de exemplo e em como usar o arquétipo, acesse a sessão de documentação chamada *QuickStart* [http://demoiselle.sourceforge.net/docs/quickstart/].

# Controlador

No *Demoiselle Framework* os controladores ou controllers servem para identificar as camadas da arquitetura de sua aplicação. É comum que as aplicações utilizem apenas três camadas: visão, negócio e persistência. Existem aplicações que utilizam fachadas. Por esse motivo, foram implementados nessa versão do framework cinco controllers:

- `ViewController`
- `FacadeController`
- `BusinessController`
- `PersistenceController`
- `ManagementController`

Além de identificar as camadas, os controllers são pré-requisitos para utilização da funcionalidade de tratamento de exceções, através do uso da anotação `@ExceptionHandler`. Isso quer dizer que para utilizar essa funcionalidade, a classe precisa usar um dos controllers citados acima ou a própria anotação `@Controller`, ou ainda um controller criado exclusivamente para sua aplicação. Todos os controllers criados no framework são estereótipos e podem ser usados também para definição de características como, por exemplo, o escopo. Isso quer dizer que se um controller tem um determinado escopo, todas as classes desse controller também terão o mesmo escopo. Foi falado que é possível criar um controller para uso exclusivo em sua aplicação, mas como fazer isso? Veja na seção abaixo.

## 5.1. Como criar seu controlador

É comum nos depararmos com situações onde precisamos criar controllers exclusivos com determinadas características ou que sirvam apenas para determinar algum tipo de funcionalidade. Para criar um novo controller no Demoiselle, basta que ele esteja anotado com `@Controller`, como no exemplo abaixo.

```
@Controller
@Stereotype
@ViewScoped
public @interface SeuController {
}
```

Neste exemplo foi criado um controlador chamado `SeuController` que tem a característica de ter um escopo de View. Isto quer dizer que toda classe que seja desse tipo de controlador também terá o escopo de View.

# Persistência

Persistência é um dos aspectos mais importantes de sistemas corporativos - grande parte desses sistemas devem em algum ponto persistir informações em um sistema gerenciador de banco de dados. A tecnologia Java conta hoje com algumas formas de facilitar o acesso a SGBD's - algumas são especificações Java como o JDBC e o JPA, outras são tecnologias desenvolvidas por terceiros como o Hibernate.

O Framework Demoiselle facilita o acesso e a configuração a algumas dessas tecnologias fornecendo produtores padrão para seus pontos de entrada e centralizando a configuração. Tudo que o desenvolvedor deve fazer é apenas injetar o recurso adequado em seu código e o Framework Demoiselle se encarregará de produzi-lo e configurá-lo.

## 6.1. JPA

O Framework Demoiselle fornece um produtor padrão para contextos de persistência da JPA. Esse produtor lê o arquivo de configuração `persistence.xml` de seu projeto e toma as providências necessárias para fabricar uma instância da classe `EntityManager` que pode ser usada para gerenciar as entidades de sua aplicação. Além disso, instâncias de `EntityManager` produzidas pelo Framework Demoiselle participam automaticamente de transações abertas através da anotação `@Transactional`, conforme apresentado no capítulo sobre *Transações*.

### Dica

Para acrescentar a dependência à extensão `demoiselle-jpa`, adicione esse código em seu arquivo `pom.xml`, na seção `dependencies`.

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle</groupId>
    <artifactId>demoiselle-jpa</artifactId>
    <scope>compile</scope>
</dependency>
```

### 6.1.1. Introdução ao mecanismo

Para injetar uma instância de `EntityManager` em sua aplicação, basta usar a anotação `@Inject`.

```
@PersistenceController
public class BookmarkDAO extends JPACrud<Bookmark, Long> {

    private static final long serialVersionUID = 1L;

    @Inject
    private EntityManager entityManager;

    public void persistBookmark(Bookmark bookmark){
        entityManager.persist(bookmark);
    }

}
```

O produtor padrão injetará o `EntityManager` configurado no arquivo `persistence.xml`. Se houver mais de um contexto de persistência configurado em `persistence.xml`, será necessário especificar qual será injetado no ponto de injeção. Para isso use a anotação `@Name`.

```
@PersistenceController
public class BookmarkDAO extends JPACrud<Bookmark, Long> {

    private static final long serialVersionUID = 1L;

    @Inject
    @Name("persistence_unit_1")
    private EntityManager entityManager;

    public void persistBookmark(Bookmark bookmark){
        entityManager.persist(bookmark);
    }

}
```

É possível invocar o utilitário `Beans` para injetar instâncias de `EntityManager` programaticamente.

```
@PersistenceController
public class BookmarkDAO extends JPACrud<Bookmark, Long> {

    private static final long serialVersionUID = 1L;

    public void persistBookmark(Bookmark bookmark){
        EntityManager entityManager = Beans.getReference(EntityManager.class);
        entityManager.persist(bookmark);
    }

    public void persistBookmarkInHistory(Bookmark bookmark){
        EntityManager entityManager = Beans.getReference(EntityManager.class , new NameQualifier("history_persist
        entityManager.persist(bookmark);
    }

}
```

## 6.1.2. Configuração

Alguns comportamentos do produtor podem ser configurados através das propriedades abaixo, que devem ser configuradas no arquivo `demoiselle.properties`.

| Propriedade | Descrição | Padrão |
|---|---|---|
| frameworkdemoiselle.persistence.default.unit.name | Define o nome da unidade de persistência padrão (configurada em `persistence.xml`) que será injetada caso a anotação `@Name` não seja usada. Não é necessário se apenas uma unidade de persistência for configurada. | |

| Propriedade | Descrição | Padrão |
|---|---|---|
| frameworkdemoiselle.persistence.entitymanager.scope | Permite determinar o escopo de unidades de persistência injetadas. Dentro do escopo determinado, todos os pontos de injeção receberão a mesma instância de `EntityManager`.<br><br>Os valores possíveis são: request, session, view, conversation, application, noscope | request |

### Dica

O escopo especial *noscope* desliga o gerenciamento de escopo de instâncias de `EntityManager` produzidas pelo Framework Demoiselle. Isso permite ao desenvolvedor controlar totalmente o ciclo de vida de um `EntityManager` injetado e ainda reter o recurso do produtor padrão.

Note que ao usar a opção *noscope*, o desenvolvedor é o responsável por controlar o ciclo de vida do gerenciador de persistência. Ele não participará de transações JPA abertas através da anotação `@Transactional` (transações JTA funcionam normalmente) e multiplos pontos de injeção durante uma requisição receberão múltiplas instâncias de `EntityManager`.

### Cuidado

Deve-se usar cautela ao alterar o escopo padrão das instâncias de `EntityManager`. Na grande maioria dos casos o escopo padrão *request* é o suficiente e alterar esse padrão deve ser feito apenas após extensa análise dos prós e contras de cada escopo.

Dê especial atenção aos escopos que podem ser serializados pelo servidor de aplicação (*session*, *view* e *conversation*) pois a especificação não define o comportamento de instâncias de `EntityManager` que são serializadas.

## 6.2. JDBC

O Framework Demoiselle fornece um produtor padrão para conexões JDBC puras. Esse produtor possui suporte ao acesso direto utilizando uma URL e ao acesso via `DataSource`, acessando a conexão através de um nome `JNDI` configurado em um servidor de aplicação.

A persistência de dados usando JDBC está disponível na extensão `demoiselle-jdbc`. Para ter acesso a essa extensão em um projeto Maven declare sua dependência no arquivo `pom.xml` de seu projeto.

### Dica

Para acrescentar a dependência à extensão `demoiselle-jdbc`, adicione esse código em seu arquivo `pom.xml`, na seção `dependencies`.

```
<dependency>
```

```
        <groupId>br.gov.frameworkdemoiselle</groupId>
        <artifactId>demoiselle-jdbc</artifactId>
        <scope>compile</scope>
    </dependency>
```

## 6.2.1. Configuração

A conexão será criada pela fábrica do Demoiselle de acordo com as configurações no arquivo de propriedade (*demoiselle.properties*). Para configurar uma conexão diretamente através de uma URL utilize as propriedades abaixo:

| Propriedade | Descrição |
|---|---|
| frameworkdemoiselle.persistence.driver.class | Implementação da interface `java.sql.Driver` que dá acesso ao SGBD utilizado pela aplicação. |
| frameworkdemoiselle.persistence.url | URL de conexão no formato `jdbc:vendor:database-properties`. |
| frameworkdemoiselle.persistence.username | Login de acesso ao SGBD. |
| frameworkdemoiselle.persistence.password | Senha de acesso ao SGBD. |

Também é possível configurar o acesso indicando um nome `JNDI` que esteja configurado no servidor de aplicação.

| Propriedade | Descrição |
|---|---|
| frameworkdemoiselle.persistence.jndi.name | Nome `JNDI` criado no servidor de aplicação para dar acesso à conexão ao banco de dados. |

É possível configurar mais de uma conexão JDBC. Para isso acrescente nas propriedades nomes separados para cada conexão como no exemplo abaixo:

### Exemplo 6.1. Criando múltiplas conexões

```
frameworkdemoiselle.persistence.conn1.driver.class=MinhaClasse
frameworkdemoiselle.persistence.conn1.url=MinhaURL
frameworkdemoiselle.persistence.conn1.username=MeuLogin
frameworkdemoiselle.persistence.conn1.password=MinhaSenha

frameworkdemoiselle.persistence.conn2.driver.class=MinhaClasse
frameworkdemoiselle.persistence.conn2.url=MinhaURL
frameworkdemoiselle.persistence.conn2.username=MeuLogin
frameworkdemoiselle.persistence.conn2.password=MinhaSenha
```

```
frameworkdemoiselle.persistence.conn1.jndi.name=MeuJndiName1
frameworkdemoiselle.persistence.conn2.jndi.name=MeuJndiName2
```

Caso várias conexões sejam configuradas, é possível determinal a conexão padrão - aquela que será utilizada quando o desenvolvedor não especificar qual deseja utilizar.

| Propriedade | Descrição |
| --- | --- |
| frameworkdemoiselle.persistence.default.datasource.name | Caso múltiplas conexões sejam criadas, define a conexão padrão quando uma `Connection` é injetada no código sem utilizar a anotação `@Name`. |

## 6.2.2. Utilização

Para utilizar uma conexão JDBC em seu código, basta injetá-la. O Demoiselle se encarregará de produzir o tipo adequado de conexão.

```
public class ClasseDAO {

    @Inject
    private Connection conn1;

    @Transactional
    public void metodoPersistir(){
        conn1.prepareStatement("INSERT INTO TAB_1 VALUES (1,'JDBC')").execute();
    }
}
```

Caso multiplas conexões tenham sido definidas, é possível utilizar a anotação `@Name` para injetar uma conexão específica.

```
public class ClasseDAO {

    @Inject
    @Name("conn1")
    private Connection conn1;

    @Inject
    @Name("conn2")
    private Connection conn2;

    @Transactional
    public void metodoPersistirEmConn1(){
        conn1.prepareStatement("INSERT INTO TAB_1 VALUES (1,'JDBC')").execute();
    }

    @Transactional
    public void metodoPersistirEmConn2(){
        conn2.prepareStatement("INSERT INTO TAB_2 VALUES (1,'JDBC')").execute();
    }
}
```

### Cuidado

Caso a propriedade `frameworkdemoiselle.persistence.default.datasource.name` seja utilizada para especificar uma conexão padrão, a anotação `@Name` só é necessária para utilizar conexões

diferentes da padrão. Caso essa propriedade não seja utilizada e existam múltiplas conexões configuradas, torna-se obrigatório o uso da anotação `@Name` em todos os pontos de injeção.

# Transação

Esta funcionalidade utiliza os recursos do CDI para interceptar e delegar adequadamente o tratamento das transações. Em outras palavras, não reinventamos a roda. Criamos algumas estratégias de delegação e controle de transação com base no que está sendo mais utilizado no mercado, algumas mais simples de configurar, outras mais completas para utilizar.

Além de plugar e usar as estratégias prontas que fizemos para você, é possível também criar a sua. Vai que você precise de algo que não pensamos ainda. O importante é que você tenha opções, e uma das opções também é não utilizar a nossa solução. Caso você esteja utilizando o *Demoiselle Framework* em conjunto com outro framework (tais como o JBoss Seam, Spring ou Google Guice) que ofereça o controle de transação, você pode usá-lo também. Viva a liberdade de escolha!

Neste capítulo apresentaremos para você como usar a nossa solução de controle de transação, as estratégias prontas que oferecemos e a criação de sua própria estratégia.

## 7.1. Configurando

Para um correto funcionamento do Demoiselle é necessário inserir o interceptador de transação no arquivo `src/main/WEB-INF/beans.xml`.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                           http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">

       <interceptors>
            <class>br.gov.frameworkdemoiselle.transaction.TransactionalInterceptor</class>
       </interceptors>
</beans>
```

## 7.2. Métodos transacionais

Vamos começar pelo mais importante: como declarar os métodos como transacionais? Como informar ao *Demoiselle Framework* que o método deve participar da sessão transacional? A resposta é muito simples: anote seu método com `@Transactional`.

```
@Transactional
public void inserir() { }
```

Se você desejar que todos os métodos de sua classe sejam transacionais, anote diretamente a classe:

```
@Transactional
public class Simples {
    public void inserir() { }
    public void alterar() { }
    public void excluir() { }
}
```

Neste exemplo, os métodos `inserir()`, `alterar()` e `excluir()` da classe `Simples` participarão do contexto transacional.

## 7.3. E se acontecer uma Exception?

Caso ocorra uma exceção na execução de um método transacional, o mecanismo fará rollback na transação automaticamente. É possível mudar este comportamento utilizando exceções de aplicação (para maiores detalhes ver *Exceção*).

```
@ApplicationException(rollback = false)
public class AbacaxiException {
}
```

## 7.4. O objeto Transaction

Para ter acesso à instância da transação corrente, basta injetar `TransactionContext` em sua classe e obter a transação corrente.

```
public class Simples {
    @Inject
    private TransactionContext transactionContext;

    public void experimento() {
        Transaction transaction = transactionContext.getCurrentTransaction();
    }
}
```

## 7.5. A estratégia mais adequada

Para o controle transacional funcionar corretamente é preciso escolher a estratégia mais adequada para o seu caso. Não existe a bala de prata, você tem que avaliar a melhor estratégia para o seu projeto.

Você também pode optar por não utilizar controle de transação. Neste caso, basta não utilizar a anotação `@Transactional`. Contudo, caso você a utilize, você poderá escolher entre as estratégias JPA, JDBC, JTA (ambas fornecidas pelo `Framework`) e uma estratégia que você pode criar ou importar para seu projeto.

A forma de selecionar cada uma dessas estratégias é descrita abaixo. Caso tente utilizar o controle de transação e não selecione nenhuma estratégia, o framework lançará uma exceção lhe avisando sobre isto!

## 7.6. Estratégia JDBC

Esta estratégia, que está disponível na extensão `demoiselle-jdbc`, delega o controle das transações para o `java.sql.Connection` da especificação JDBC. Você deve escolher esta estratégia quando estiver persistindo dados com JDBC e utilizando apenas uma base de dados em sua aplicação. Como um `Connection` acessa apenas uma base de dados, não há como fazer o controle transacional de base de dados distintas.

A transação JDBC é simples de configurar e não exige nenhum recurso externo à sua aplicação. Para utilizá-la basta que seu projeto adicione no arquivo pom.xml dependência à extensão `demoiselle-jdbc`, que o Demoiselle fará a seleção por essa estratégia de forma automática.

### Dica

Para utilizar a estratégia de transação JDBC, inclua a dependência para extensão JDBC no arquivo pom.xml.

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle</groupId>
    <artifactId>demoiselle-jdbc</artifactId>
    <scope>compile</scope>
</dependency>
```

## 7.7. Estratégia JPA

Esta estratégia, que está disponível na extensão `demoiselle-jpa`, delega o controle das transações para o `javax.persistence.EntityManager` da especificação JPA. Você deve escolher esta estratégia quando estiver persistindo dados com JPA e utilizando apenas uma base de dados em sua aplicação. Como um `EntityManager` acessa apenas uma unidade de persistência, não há como fazer o controle transacional de unidades distintas.

A transação JPA é simples de configurar e não exige nenhum recurso externo à sua aplicação. Para utilizá-la basta que seu projeto adicione no arquivo pom.xml dependência à extensão `demoiselle-jpa`, que o Demoiselle fará a seleção por essa estratégia de forma automática.

### Dica

Caso não esteja utilizando o arquétipo JSF-JPA fornecidos pelo Demoiselle, confira se a dependência para a extensão está indicada corretamente no arquivo pom.xml.

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle</groupId>
    <artifactId>demoiselle-jpa</artifactId>
    <scope>compile</scope>
</dependency>
```

## 7.8. Estratégia JTA

Esta estratégia, também disponível através de uma extensão (`demoiselle-jta`), é responsável por delegar o controle de transação para um container JEE. Com a `JTATransaction` é possível incluir várias unidades de persistência de uma mesma aplicação no mesmo contexto transacional. Isso mesmo, o famoso *Two-Phase Commit (2PC)*.

A estratégia JTA não serve apenas para persistência em banco de dados, serve também para integrar com tecnologias que façam acesso ao contexto JTA, como é o caso do EJB. Para ativar esta estratégia basta que seu projeto adicione no arquivo pom.xml a dependência à extensão `demoiselle-jta`, que o Demoiselle fará a seleção por essa estratégia de forma automática, pois essa estratégia tem prioridade em relação à estratégia JPA e JDBC.

Feito isto, o controle transacional será delegado para a transação acessível via JNDI com o nome `UserTransaction`. A estratégia acessa o objeto da seguinte maneira: `Beans.getReference(UserTransaction.class)`. Portanto, para você utilizar esta estratégia, você precisa de um container JEE ou de um servidor JTA qualquer.

Caso você esteja persistindo os dados com JPA, é preciso também informar no arquivo `persistence.xml` o endereço da conexão JTA gerenciada. Veja um exemplo utilizando o servidor de aplicações JBoss AS7 e com o provider Hibernate (embutido no JBoss AS) como implementação JPA:

```
<?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="bookmark-ds" transaction-type="JTA">
    <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
    <properties>
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="false" />
        <property name="hibernate.hbm2ddl.auto" value="update" />
        <property name="hibernate.transaction.jta.platform"
            value="org.hibernate.service.jta.platform.internal.JBossAppServerJtaPlatform" />
    </properties>
    </persistence-unit>
</persistence>
```

### Dica

Caso não esteja utilizando o arquétipo JSF-JPA fornecidos pelo Demoiselle, confira se a dependência para a extensão está indicada corretamente, no arquivo pom.xml.

```
<dependency>
    <groupId>br.gov.frameworkdemoiselle</groupId>
    <artifactId>demoiselle-jta</artifactId>
    <scope>compile</scope>
</dependency>
```

Caso você esteja persistindo os dados com JDBC, é preciso informar no arquivo `demoiselle.properties` o endereço da conexão JTA gerenciada. Veja um exemplo utilizando o servidor de aplicações JBoss AS7:

```
frameworkdemoiselle.persistence.jndi.name=java:jboss/datasources/ExampleDS
```

## 7.9. Criando sua própria estratégia

Caso nenhuma das estratégias oferecidas sirva para você, crie a sua. Basta escrever uma classe não-final que implemente a interface `Transaction` do pacote `br.gov.frameworkdemoiselle.transaction`. É

preciso que sua classe não possua construtores explícitos ou que possua um construtor público sem parâmetros. É possível fazer injeções nesta classe.

```
package projeto;

import br.gov.frameworkdemoiselle.transaction.Transaction;

public class MyTransaction implements Transaction {
    public void begin() { }
    public void commit() { }
    public void rollback() { }
    public void setRollbackOnly() { }
    public int getStatus() { }
    public void setTransactionTimeout(int seconds) { }
    public boolean isActive() { }
    public boolean isMarkedRollback() { }
}
```

Pronto, é só isso! Agora, os métodos anotados com `@Transactional` irão utilizar a estratégia criada em seu projeto de forma automática, mesmo que as extensões `demoiselle-jdbc`, `demoiselle-jpa` e `demoiselle-jta` sejam adicionadas ao projeto, pois o framework dará prioridade máxima à estratégia criada no projeto.

## 7.10. Escolhendo a estratégia manualmente

Existem alguns casos nos quais você vai ter que definir a estratégia manualmente. Um exemplo é quando seu projeto implementa mais do que uma estratégia de transação. Outra situação pode acontecer em casos de teste, nos quais você queira utilizar estratégia diferente. Nesses casos você deve definir no arquivo `demoiselle.properties` qual estratégia será utilizada. Veja alguns exemplos de definição de estratégias própria, `JDBCTransaction`, `JPATransaction` e `JTATransaction`. É importante notar que apenas uma estratégia pode estar ativa por vez:

```
frameworkdemoiselle.transaction.class=projeto.MyTransaction
frameworkdemoiselle.transaction.class=br.gov.frameworkdemoiselle.transaction.JDBCTransaction
frameworkdemoiselle.transaction.class=br.gov.frameworkdemoiselle.transaction.JPATransaction
frameworkdemoiselle.transaction.class=br.gov.frameworkdemoiselle.transaction.JTATransaction
```

# Exceções

Esta funcionalidade foi feita para você que acha muito verboso encher o código de `try/catch`. E o que dizer de repetir o tratamento de exceções em vários métodos da mesma classe na base do copiar/colar? Oferecemos a você uma alternativa para resolver estes problemas, mas você estará livre para usá-la: isoladamente, misturando com a forma verbosa ou até mesmo não usá-la.

## 8.1. Configurando

Para um correto funcionamento do Demoiselle é necessário inserir o interceptador de exceção no arquivo `src/main/WEB-INF/beans.xml`.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <interceptors>
        <class>br.gov.frameworkdemoiselle.exception.ExceptionHandlerInterceptor</class>
    </interceptors>
</beans>
```

## 8.2. Tratadores de exceção

Para definir um tratador de exceção, basta anotar o método com `@ExceptionHandler`. O tratamento de exceções só é possível em classes anotadas com `@Controller` ou os derivados desta anotação. Para ver mais detalhes sobre controladores no Demoiselle, leia o capítulo *Controladores*.

```
@Controller
public class Simples {

    @ExceptionHandler
    public void tratador(NullPointerException cause) { }
    public void inserir() { }
    public void alterar() { }
    public void excluir() { }
}
```

Neste exemplo, qualquer exceção do tipo `NullPointerException` que ocorrer nos métodos da classe `Simples` terá o tratamento delegado para o método `tratador()`. Para as exceções não tratadas, o comportamento seguirá o padrão da linguagem.

## 8.3. Múltiplos tratadores

Não se limite a apenas um tratador por classe, você pode ter vários.

```
@Controller
public class Simples {
```

```
    @ExceptionHandler
    public void tratador(NullPointerException cause) { }
    @ExceptionHandler
    public void tratador(AbacaxiException cause) { }
    public void inserir() { }
    public void alterar() { }
    public void excluir() { }
}
```

Caso as exceções `NullPointerException` ou `AbacaxiException` ocorram nos métodos da classe `Simples`, o tratamento será delegado para o seu tratador.

## 8.4. Misturando os dois mundos

É possível que, em determindas situações no seu projeto, você precise misturar o tratamento sofisticado com a tradicional forma verbosa. Como fazer isso?

Suponha que no método `inserir()` você precise dar um tratamento exclusivo para uma exceção. Para isso, misture as duas abordagens para atingir o seu objetivo.

```
@Controller
public class Simples {

    @ExceptionHandler
    public void tratador(NullPointerException cause) { }
    public void inserir() {
        try {
            // tenta algo
        } catch (AbacaxiException cause ) {
            // trata o problema
        }
    }
    public void alterar() { }
    public void excluir() { }
}
```

Neste caso a exceção `AbacaxiException` só será tratada no método `inserir()`.

## 8.5. Exceção de Aplicação

Imagine que você precise informar que, caso um determinado tipo de exceção seja lançado através do seu método, a transação atual sofrerá um *rollback*. Ou, então, que haja necessidade de informar o grau de severidade da exceção, de forma que uma camada de apresentação específica a trate de forma diferenciada. Estas duas opções são possíveis através do uso da anotação `@ApplicationException`. Utilize-a em suas exceções e informe os atributos `rollback` e `severity` para alcançar os objetivos acima.

```
@ApplicationException(rollback=true, severity=SeverityType.INFO)
public class MinhaException extends Exception {
}
```

No exemplo citado acima, estamos informando que caso esta exceção seja lançada por um método anotado com `@Transactional`, este método sofrerá um rollback. Ao mesmo tempo, sua camada de exibição apresentará uma mensagem automática para o usuário, conforme o nível de severidade. O comportamento padrão para o `rollback` é realizar o rollback sempre que a exceção for lançada. O grau de severidade padrão é informativo (`INFO`).

## 8.6. Tratamento Padrão

As exceções lançadas a partir da camada de negócio, ou de persistência, não causam a interrupção de sua aplicação, muito menos apresentam a tela padrão de erro do JSF ou de outra tecnologia de visão. Qualquer exceção lançada que chega até a camada de apresentação recebe um tratamento especial. Inicialmente, ela é encapsulada de forma que possa ser exibida de forma elegante para o usuário. No caso do JSF, é utilizado o mecanismo de Messages próprio desta tecnologia.

No caso do Vaadin, o tratamento é bem semelhante, contudo a exceção é tratada de forma que possa ser exibida adotando os mecanismos próprios da tecnologia. No caso de exceções que não usam a anotação `@ApplicationException`, um *rollback* é realizado de forma automática. Por último, sem o uso desta anotação, toda exceção é vista como tendo nível de gravidade igual a `ERROR`.

# Configuração

## 9.1. Configurações em uma aplicação

Muitas vezes, por motivos diversos, é necessário parametrizar a aplicação à partir de algum mecanismo de configuração. E em java é comum se utilizar as seguintes abordagens para armazenas as configurações:

- *arquivo de propriedades*: tratam-se de simples arquivos de texto nomeados com a extensão `.properties`, os quais são escritos com a sintaxe `chave=valor`, armazenando uma única chave por linha;
- *arquivo XML*: são arquivos de texto altamente estruturados com a sintaxe de tags e que permitem uma maior validação dos seus valores, sendo geralmente nomeados com a extensão `.xml`;
- *variáveis de ambiente*: valores definidos no sistema operacional, independente de plataforma (Windows, Linux, Mac OS, etc) e que podem ser recuperados durante a execução da aplicação.

Esse capítulo mostra de que maneira o *Demoiselle Framework* pode facilitar a utilização dessas formas de configuração, oferencendo vários recursos interessantes para a sua aplicação.

## 9.2. As classes de configuração

O primeiro passo para a utilização do mecanismo de configuração em uma aplicação consiste em criar uma classe específica para armazenar os parâmetros desejados e anotá-la com `@Configuration`. O código abaixo mostra um exemplo de classe de configuração:

```
@Configuration
public class BookmarkConfig {

    private String applicationTitle;

    private boolean loadInitialData;

    public String getApplicationTitle() {
        return applicationTitle;
    }

    public boolean isLoadInitialData() {
        return loadInitialData;
    }
}
```

### Nota

As classes anotadas com `@Configuration` são instanciadas uma única vez (seguindo o padrão de projeto *singleton*) e podem ser injetadas em qualquer ponto da aplicação. Seu ciclo de vida é gerenciado automaticamente pelo CDI. Os recursos (arquivo ou variável de ambiente) são lidos no primeiro acesso à respectiva classe de configuração, quando os seus atributos são preenchidos automaticamente.

### Nota

Recomenda-se usar o sufixo “Config” nas classes de configuração, e que sejam criados apenas os acessores para leitura (*getters*).

Esse é um exemplo bastante simples, no qual não são especificados nem nome nem tipo do arquivo de configuração. Nessa situação os parâmetros de nome *applicationTitle* e *loadInitialData* serão procurados em um arquivo de propriedades de nome *demoiselle.properties*. Ou seja, quando não especificados nome e tipo do arquivo, assume-se que o arquivo é do tipo *propriedades* e seu nome é *demoiselle*. Mas como fazer para não utilizar o valor padrão e definir nome e tipo do arquivo? Bastante simples. Basta adicionar esses parâmetros à anotação @Configuration, como mostra o exemplo a seguir:

```
@Configuration(resource="my-property-file", type=ConfigType.XML)
public class BookmarkConfig {

    private String applicationTitle;

    private boolean loadInitialData;

    public String getApplicationTitle() {
        return applicationTitle;
    }

    public boolean isLoadInitialData() {
        return loadInitialData;
    }
}
```

Devemos atribuir o nome do arquivo de configuração ao parâmetro *resource*, sem a extensão. Ao parâmetro *type* pode ser atribuída uma das três possibilidades: *ConfigType.PROPERTIES*, que é o valor padrão e indica que as configurações daquela classe estão em um arquivo do tipo *properties*; *ConfigType.XML*, que indica que as configurações daquela classe estão em um arquivo do tipo *xml*; e *ConfigType.SYSTEM*, que indica que as configurações daquela classe são valores definidos pelo Sistema Operacional. Nesse exemplo, ao definir *resource* e *type* os parâmetros de nome *applicationTitle* e *loadInitialData* serão procurados em um arquivo xml de nome *my-property-file* (my-property-file.xml).

Outro parâmetro que você pode ajustar nessa anotação é o prefixo. Ao definir um valor de prefixo você informa que o nome das propriedades definidas naquela classe devem ser concatenados com o prefixo, de forma que o nome dos atributos procurados no arquivo seja *prefixo.nomeatributo*. O exemplo abaixo mostra a utilização desse parâmetro. Nesse caso, os parâmetros de nome *info.applicationTitle* e *info.loadInitialData* serão procurados em um arquivo de propriedade de nome *my-property-file* (*my-property-file.properties*).

```
@Configuration(prefix="info", resource="my-property-file")
public class BookmarkConfig {

    private String applicationTitle;

    private boolean loadInitialData;

    public String getApplicationTitle() {
        return applicationTitle;
```

```
    }

    public boolean isLoadInitialData() {
        return loadInitialData;
    }
}
```

### Nota

O *Demoiselle Framework* adiciona automaticamente o ponto entre o prefixo e o nome do atributo, você não precisa se preocupar com isso.

### Dica

No arquivo xml o prefixo corresponde a uma *tag* acima das tag que correspondem aos atributos. O exemplo acima ficaria da seguinte forma em um arquivo xml:

```
<info>
    <applicationTitle>Demoiselle Application<\applicationTitle>
    <loadInitialData>true<\loadInitialData>
</info>
```

## 9.3. Especificando os parâmetros

Atualmente são suportados nativamente pelo *Demoiselle Framework* parâmetros de sete categorias diferentes. São eles: tipos primitivos (*int, float, boolean, etc*), classes *wrapper* (*Integer, Float, Boolean, etc.*) , *String*, *Class*, *Map*, *Array* e instâncias de *Enum*. A seguir vamos explicar e exemplificar como utilizar cada um desses tipos, e alertar para as possíveis exceções que poderão ser lançadas para sua aplicação.

### Cuidado

A partir da versão 2.4.0 não são mais reconhecidas as convenções de substituição de nomes. Os parâmetros serão procurados exatamente como foram definidos na classe de configuração.

**Primitivos**

A utilização dos tipos primitivos é bastante simples. Veja no exemplo abaixo uma classe de configuração um arquivo de configurações, que ilustram como é o procedimento para adicionar parâmetros do tipo primitivo.

```
@Configuration
    public class BookmarkConfig {

    private int pageSize;

    public String getPageSize() {
```

```
        return pageSize;
    }
}
```

Para essa classe, o arquivo de propriedade correspondente (*demoiselle.properties*) deveria apresentar o seguinte conteúdo:

```
pageSize=10
```

Bastante simples, não? Mesmo assim, é bom ficarmos atentos, pois alguns cenários diferentes podem acontecer. Vamos supor por exemplo que, por um motivo qualquer, a classe de configuração não esteja associada a um arquivo que contenha a chave de um de seus parâmetros. Nesse caso será atribuido o valor padrão da linguagem ao atributo, que para os tipos primitivos é 0, exceto para os tipos *boolean*, cujo valor padrão é *false*, e *char*, cujo valor padrão é o caracter nulo (*'\u0000'*). Outro cenário possível é a existência da chave, mas sem valor atribuído (*pageSize=*). Nesse caso o valor encontrado no arquivo é equivalente a uma *String* vazia, e a exceção *ConfigurationException*, cuja causa foi uma *ConversionException*, será lançada.

**Wrappers**

Os atributos do tipo *wrapper* devem ser utilizados da mesma forma que os atributos do tipo primitivo. A única diferença entre eles é que o valor padrão atribuído a um parâmetro, no caso da classe de configuração não estar associada a um arquivo que contenha sua chave, é nulo.

**Strings**

Por sua vez, as configurações do tipo *String* tem funcionalidade bastante similar às configurações do tipo Wrapper. A diferença fica por conta de que, ao deixar a chave sem valor atribuído, para atributo desse tipo, não será lançada exceção, pois que não haverá o problema de conversão, e à configuração será atribuido o valor de uma *String* vazia.

**Class**

A partir da versão 2.4.0 é possível ter atributos do tipo *Class* como parâmetro. O atributo pode ou não ser tipado, e no arquivo o valor atribuído à chave deve corresponder ao nome ( *Canonical Name*) de uma classe existente. Abaixo temos um exemplo de uma classe de configuração com dois atributos do tipo *Class*, um tipado e outro não tipado:

```
@Configuration
public class BookmarkConfig {

    private Class<MyClass> typedClass;

    private Class<?> untypedClass;

    public Class<MyClass> getTypedClass() {
        return typedClass;
    }

    public Class<?> getUntypedClass() {
        return untypedClass;
    }
}
```

```
```

O arquivo de propriedades teria o seguinte conteúdo:

```
typedClass=package.MyClass
untypedClass=package.MyOtherClass
```

Caso uma chave de uma configuração do tipo *Class* não tenha valor atribuído ou seja atribuído um nome de classe que não existe (ou não possa ser encontrada pela aplicação), no momento do seu carregamento sera lançada uma exceção do tipo *ConfigurationException*, cuja causa é uma *ClassNotFoundException*. Caso a classe de configuração não esteja associada a um arquivo que contenha a chave de um de seus parâmetros do tipo *Class*, este será carregado com valor nulo.

**Map**

Para utilizar parâmetros do tipo *Map*, o arquivo de configurações deve usar a seguinte estrutura na formação da chave: *prefixo+nomedoatributo+chavedomap*. Vejamos um exemplo. Se temos em nossa aplicação uma classe de configuração como a mostrada abaixo:

```
@Configuration
public class BookmarkConfig {

    private Map<String, String> connectionConfiguration;

    public Map<String, String> getConnectionConfiguration() {
        return connectionConfiguration;
    }
}
```

O arquivo de configuração deverá ser preenchido no seguinte formato (se for do tipo *properties*):

```
connectionConfiguration.ip=192.168.0.120
connectionConfiguration.gateway=192.168.0.1
connectionConfiguration.dns1=200.10.128.99
connectionConfiguration.dns2=200.10.128.88
```

Dessa forma, ao fazer a chamada *connectionConfiguration.get("gateway");* por exemplo, o valor retornado será *192.168.0.1*.

## Dica

Você pode utilizar a chave do Map com nome "default" para indicar que, no arquivo de configuração, a chave é formada apenas pela junção do prefixo com o atributo, sem utilizar a própria chave do Map. Por exemplo, se o seu arquivo de propriedades contiver uma chave:

```
prefix.myMap=Default Value
```

então seu código poderá ter um comando:

```
String value = myMap.get("default");
```

e o valor de *value* será *"Default Value"*.

Caso a classe de configuração não esteja associada a um arquivo que contenha a chave de um de seus parâmetros do tipo *Map*, este será carregado com valor nulo, e estará sujeito às exceções informadas anteriormente, conforme o tipo de variáveis que ele contenha.

**Array**

No caso do *Array*, a principal diferença em relação às demais formas de declarar configurações é a maneira de atribuir valores aos seus respectivos elementos no arquivo de configuração. Por exemplo, para que um *Array* de inteiros, de nome *integerArray* tenha o conteúdo *{-1, 0, 1}*, você deve criar um arquivo de propriedades que contenha as seguintes linhas:

```
integerArray=-1
integerArray=0
integerArray=1
```

Exceto a forma de atribuir os valores às configurações, se comporta de acordo com o tipo de variável que ele contém, conforme o espeficifcado para cada um.

**Enum**

É possível criar uma lista de constantes do tipo *Enum* e carregar um valor de constante através de um arquivo de configuração. Por exemplo, caso exista o seguinte *Enum*

```
public enum ConfigurationType {

    PROPERTIES , XML , SYSTEM;

}
```

e ele seja usado no seguinte arquivo de configuração

```
@Configuration
public class ConfigurationLoader {

    private ConfigurationType loadedConfigurationType;
```

```
    public ConfigurationType getLoadedConfigurationType(){
        return loadedConfigurationType;
    }
}
```

O arquivo do tipo *properties* pode ser criado assim:

```
loadedConfigurationType=SYSTEM
```

### Nota

O valor definido no arquivo de configuração para atributos do tipo *Enum* deve ser idêntico ao nome da constante definida no *Enum*, inclusive casando letras maiúsculas e minúsculas. De fato, o valor da propriedade deve casar com o valor retornado no código: *Enum.name()*.

Caso o valor definido no arquivo de configuração não case com nenhuma constante definida no *Enum*, uma exceção de tipo *ConfigurationException* de causa *ConversionException* será lançada. Já se à propriedade for atribuido um valor vazio, o atributo do tipo *Enum* receberá o valor *null*.

## 9.4. Mais Recursos

Além das possibilidades relacionadas acima, existem ainda algumas anotações e recursos extras que o *Demoiselle Framework* oferece para o desenvolvedor na utilização das configurações. A seguir listamos e explicamos como utilizar esses recursos em sua aplicação.

**Ignore**

Por padrão, todos os atributos existentes em uma classe anotada com `@Configuration` são tratados como parâmetros de configuração e serão automaticamente preenchidos durante a leitura do recurso. Porém, caso você não queira que determinado atributo seja tratado como parâmetro dentro desse tipo de classe, basta anotá-lo com a anotação `@Ignore`, que o atributo será ignorado (como indica a própria anotação) pelo carregador de configurações.

**Valor Padrão**

Muitas vezes é interessante que especifiquemos um valor padrão para o parâmetro, para o caso dele não estar presente no arquivo de configuração. Para isso, basta atribuir o valor desejado no momento da declaração do atributo, como exemplificado abaixo:

```
@Configuration
public class BookmarkConfig {

    private String applicationTitle = "My App";

    public String getApplicationTitle() {
        return applicationTitle;
    }
}
```

Com essa atribuição, se no arquivo de propriedades não existir uma chave com valor `applicationTitle` esse parametro será carregado com o valor `My App`.

**Bean Validation**

Fazer validação mantém a integridade dos dados e pode ser fator importante na lógica da aplicação. A partir da versão 2.4.0 o *Demoiselle* permite que os atributos das classes de configuração sejam anotados com todas as anotações definidas pela JSR 303 (Bean Validation). Com esse recurso você pode exigir que determinado parâmetro não seja nulo (*@NotNull*), limitar um valor máximo ou mínimo para ele (*@Max* e *@Min*, respectivamente), dentre outras restrições (que podem ser feitas simultâneamente). A lista completa das restrições que podem ser aplicadas nos atributos das classes de configuração pode ser conferida aqui: *http://docs.oracle.com/javaee/6/tutorial/doc/gircz.html* [http://docs.oracle.com/javaee/6/tutorial/doc/gircz.html].

Para utilizar esse recurso você deve ter como dependência de seu projeto alguma implementação da especificação Bean Validation. A implementação de referência é o *Hibernate Validator* [http://www.hibernate.org/subprojects/validator].

**Name**

Em alguns casos você pode querer que um determinado parâmetro tenha nomes diferentes na classe de configuração e no arquivo de propriedades. Para isso você pode utilizar a anotação *@Name*. Basta anotar o atributo passando como parâmetro o nome pelo qual você deseja que ele seja procurado no arquivo de propriedades, como mostra o exemplo abaixo:

```
@Configuration
public class BookmarkConfig {

    @Name("app.title")
    private String applicationTitle;

    public String getApplicationTitle() {
        return applicationTitle;
    }
}
```

Com essa anotação, ao invés de procurar pela chave *applicationTitle*, o *Demoiselle* irá buscar pela chave *app.title*.

**Escopo**

A partir da versão 2.3.3 do *Demoiselle Framework* as classes anotadas com *@Configuration* estarão por padrão no escopo estático (*@StaticScoped*).

**Extratores**

Você precisa de parâmetros de um tipo que ainda não é suportado pelo *Demoiselle*? Você pode implementar sua própria classe extratora. A partir da versão 2.4.0 as aplicações podem implementar a interface *ConfigurationValueExtractor*, e ter um extrator de configurações para atributos de qualquer tipo. Essa interface obriga a classe a implementar os métodos: *boolean isSupported(Field field)*, que retorna `true` caso a classe seja extratora daquele tipo de campo, e *Object getValue(String prefix, String key, Field field, Configuration configuration) throws Exception* que deve extrair o valor do campo da forma correta.

# Inicialização

## 10.1. Introdução ao mecanismo

Uma aplicação qualquer, seja do tipo Web ou desktop, geralmente necessita efetuar determinadas tarefas durante a sua inicialização e ou finalização. Eis alguns exemplos: abertura de conexão a um servidor de banco de dados, carregamento de parâmetros de configuração a partir de arquivos externos e execução de scripts específicos. Ou seja, normalmente essas ações são definidas como funções estruturais da aplicação.

Para que a execução dessas ações ocorra de forma concisa, faz-se necessário o uso de um mecanismo padronizado para inicialização do ambiente. O *Demoiselle Framework* fornece esse mecanismo através da especificação CDI introduzindo duas anotações: `@Startup` e `@Shutdown`.

## 10.2. Implementação na aplicação

A fim de utilizar o mecanismo inerente do *Demoiselle Framework*, é preciso simplesmente anotar os métodos contendo as instruções desejadas com `@Startup`, para a *inicialização*, ou `@Shutdown`, para a *finalização*.

### Dica

O mecanismo de inicialização do *Demoiselle Framework* é independente da natureza da aplicação Java, isto é, visa tanto aplicações do tipo Web quanto do tipo desktop (ex: Swing).

As instruções contidas em um método anotado com `@Startup` serão executadas automaticamente quando a aplicação Java for inicializada, seja ela hospedada em um contêiner Web ou executada através de um método `main()`. Nenhum outro arquivo ou classe precisa ser definido. A anotação `@Startup` pode ser utilizada em conjunto com a anotação `@Priority`, que recebe como parâmetro um número inteiro que serve para definir a prioridade de execução do respectivo método, na existência de mais de um inicializador para a aplicação.

De maneira análoga, um método anotado com `@Shutdown` será executado no momento de finalização de uma aplicação, obedecendo também à ordem de prioridade definida com a anotação `@Priority`.

Eis um exemplo de implementação de inicializador em uma aplicação:

```
public class BookmarkInitializer {

    @Startup
    @Priority(1)
    public void initialize() {
        ...
    }

    @Shutdown
    @Priority(5)
    public void finalize() {
        ...
    }
}
```

### Dica

Para a definição de prioridade de execução de um método na inicialização ou finalização, podem ser utilizadas as constantes `MIN_PRIORITY` ou `MAX_PRIORITY` presentes em `br.gov.frameworkdemoiselle.annotation.Priority`.

## 10.3. Um exemplo prático

Eis um interessante caso de uso de inicialização e finalização: rodar um servidor de modo standalone em paralelo à execução da aplicação principal. Eis o código referente a essa implementação:

```
import br.gov.frameworkdemoiselle.lifecycle.Shutdown;
import br.gov.frameworkdemoiselle.lifecycle.Startup;

import static br.gov.frameworkdemoiselle.annotation.Priority.MAX_PRIORITY;
import static br.gov.frameworkdemoiselle.annotation.Priority.MIN_PRIORITY;

public class DatabaseServer {

    private final org.hsqldb.Server server;

    public DatabaseServer() {
        server = new Server();
        server.setDatabaseName(0, "db");
        server.setDatabasePath(0, "database/db");
        server.setPort(9001);
        server.setSilent(true);
    }

    @Startup
    @Priority(MAX_PRIORITY)
    public void startup() {
        server.start();
    }

    @Shutdown
    @Priority (MIN_PRIORITY)
    public void shutdown() {
        server.stop();
    }
}
```

# Tratamento de Mensagens

## 11.1. Mensagens em uma aplicação

Uma aplicação bem estruturada, seja na plataforma Web ou Desktop, deve exibir mensagens informativas, de aviso, ou de erro para o usuário após efetuar determinadas tarefas. Por exemplo, após gravar um registro no banco de dados, é aconselhável que a aplicação exiba uma mensagem informativa. Se alguma exceção ocorreu, é preciso exibir uma mensagem de erro. Ou seja, a severidade da mensagem deve ser escolhida de acordo com o resultado da execução.

Veja na tabela a seguir a definição de cada um dos níveis de severidade da mensagem em uma aplicação e um exemplo de texto associado:

### Tabela 11.1. Níveis de severidade em mensagens

| Severidade | Utilização | Exemplo de texto |
|---|---|---|
| Informação | Usada quando da ocorrência normal ou esperada no fluxo, com o objetivo de confirmar ao usuário uma situação de sucesso. | “Aluno incluído com sucesso.” |
| Aviso | Usada quando uma regra de negócio ou validação qualquer da aplicação tenha desviado o fluxo normal de execução. | “A turma selecionada já está lotada. Matrícula do aluno não efetuada.” |
| Erro | Reservado para o caso de uma situação anormal, tal como exceções provenientes de ambiente (falha de conexão na rede, queda de um servidor de banco de dados, etc) tenha impedido a execução. | “Não foi possível efetuar a modificação do aluno.” |

Em uma aplicação construída usando-se arquitetura em camadas, as mensagens geralmente são originadas em uma determinada camada e precisam ser transmitidas às demais até chegar ao usuário. Por exemplo, se ocorre um erro de gravação no banco de dados, isto é, na camada de persistência, tal informação precisa ser repassada às camadas subsequentes, ou seja, as camadas de negócio e posteriormente de controle e visão. Este conceito é justamente chamado de *tratamento de mensagens*, o qual, ao lado do tratamento de exceções, auxilia o fluxo de execuções em uma aplicação bem arquiteturada. Em resumo, é preciso programar a troca de mensagens entre as diversas camadas de uma aplicação orientada a objetos.

Veremos na seção a seguir como o *Demoiselle Framework* pode ajudar o desenvolvedor na tarefa de troca de mensagens em uma aplicação.

## 11.2. Introdução ao mecanismo

Ortogonalmente às camadas verticais da aplicação (i.e., apresentação, negócio e persistência), a arquitetura proposta pelo *Demoiselle Framework* fornece os contextos de transação, segurança e mensagem. Este último é justamente responsável pela troca de mensagens entre as camadas.

Tecnicamente falando, o contexto de mensagens no *Demoiselle Framework* é representado pela interface `MessageContext`. Esta interface contém métodos destinados a inclusão, recuperação e limpeza de mensagens no contexto.

Para obter uma instância do contexto de mensagens, basta injetá-lo na classe:

```
@Inject
private MessageContext messageContext;
```

A maneira mais simples de se inserir uma mensagem no contexto é invocando o método `add()` de `MessageContext` passando como argumento o texto literal da mensagem:

```
messageContext.add("Aluno inserido com sucesso.");
```

Opcionalmente, pode-se indicar a severidade da mensagem:

```
messageContext.add("Deseja realmente excluir o aluno?", SeverityType.WARN);
```

### Dica

Quando a severidade não é informada (argumento de tipo `SeverityType`), será considerado o nível informativo, isto é, `INFO`.

Para a transmissão das mensagens no contexto é também fornecida a interface `Message`, a qual permite com que os textos sejam obtidos de catálogos especializados, tais como arquivos de propriedades ou banco de dados. Essa interface precisa ser implementada em uma classe customizada pelo desenvolvedor. Sua estrutura é bem simples:

```
public interface Message {

    String getText();
    SeverityType getSeverity();
}
```

A classe `DefaultMessage` fornecida pelo *Demoiselle Framework* é uma implementação da interface `Message` que faz uso dos arquivos de recurso da aplicação para obtenção das descrições das mensagens. Especificamente utiliza o arquivo `messages.properties`.

### Nota

A unidade básica de manipulação de mensagens no *Demoiselle Framework* é a interface `Message`. Basta que ela seja implementada na aplicação para que o contexto de mensagens possa manipulá-la. A classe `DefaultMessage` é oferecida como implementação padrão dessa interface.

Para incluir uma mensagem no contexto usando o tipo `Message` é preciso invocar o método `add()` de `MessageContext` passando o objeto como argumento. Eis um exemplo disso utilizando a classe `DefaultMessage`:

```
Message message = new DefaultMessage("Ocorreu um erro ao excluir o aluno!", SeverityType.ERROR);
messageContext.add(message);
```

A extensão para *demoiselle-jsf* transfere automaticamente as mensagens incluídas no `MessageContext` para o `FacesContext`.

O contexto de mensagens, representado pela interface `MessageContext`, é capaz de armazenar diversas mensagens em uma mesma requisição. Ele não é restrito a aplicações do tipo Web, isto é, pode ser usado também para aplicações do tipo desktop (i.e., Swing).

## 11.3. Parametrização das mensagens

Um recurso importante no tratamento de mensagens de uma aplicação consiste na parametrização dos textos que elas carregam. Isso é extremamente útil para indicar com detalhes uma situação específica, com o objetivo de melhor informar o usuário. Por exemplo, ao invés de simplesmente exibir "Exclusão de disciplina não permitida" é preferível mostrar o seguinte texto mais explicativo "Exclusão não permitida: disciplina está sendo usada pela turma 301". Para efetuar essa parametrização, é preciso usar a formatação de textos padronizada pela classe *`java.text.MessageFormat`* [http://download.oracle.com/javase/6/docs/api/java/text/MessageFormat.html] da API do Java, que basicamente considera a notação de chaves para os argumentos.

O *Demoiselle Framework* usa a `MessageFormat` para efetuar a parametrização e formatação dos valores. Para usar essa funcionalidade, basta incluir as chaves na string da mensagem e ao inseri-la no contexto especificar os parâmetros:

```
Message message = new DefaultMessage("Aluno {0} inserido com sucesso");
messageContext.add(message, aluno.getNome());
```

Eis um exemplo mais avançado do uso de parametrizações em mensagens:

```
Message message = new DefaultMessage("As {1,time} do dia {1,date,short} houve {2} na turma {0}");
messageContext.add(message, new Integer(502), new Date(), "falta de professor");
```

O resultado da execução deste código seria um texto como: "Às 13:45:05 do dia 03/01/11 houve falta do professor na turma 502". Ou seja, os argumentos `{0}`, `{1}` e `{2}` são substituídos por seus respectivos elementos na ordem em que são passados no método `add()`, sendo que ainda podemos formatar esses valores.

### Dica

É possível passar mais de um parâmetro nas mensagens adicionadas no contexto, usando para isso a sintaxe de formatação da classe `java.text.MessageFormat`. A ordem dos argumentos passados no método `add()` reflete nos elementos substituídos na string.

## 11.4. Internacionalização das mensagens

A `DefaultMessage` oferece a funcionalidade de internacionalização da aplicação, ou seja, a disponibilização dos textos em diversos idiomas. Numa aplicação do tipo Web, o mecanismo de internacionalização faz com que

as mensagens e textos sejam exibidos de acordo com o idioma selecionado pelo usuário na configuração do navegador.

Como já foi citado anteriormente, a implementação `DefaultMessage` busca os textos das mensagens no arquivo de recursos `messages.properties`, o qual possui entradas no conhecido formato `chave=valor`. Todavia, para fazer uso desses textos, é preciso passar a chave da mensagem desejada como argumento na invocação do construtor da classe. Este identificador precisa estar entre sinais de chaves, tal como no exemplo a seguir:

```
Message message = new DefaultMessage("{ALUNO_INSERIR_OK}");
messageContext.add(message, aluno.getNome());
```

Ou seja, ao invés de usar a string literal, passamos o identificador da chave presente no arquivo de propriedades.

**Nota**

O símbolo de chaves `{}` na string do construtor da classe `DefaultMessage` indica se o texto da mensagem será recuperado de um arquivo de recursos ou considerado o texto literal.

Eis um exemplo de conteúdo do arquivo `messages.properties` destinado a manter por padrão os textos no idioma português brasileiro (i.e., `pt-BR`). Veja que ele contém a chave `ALUNO_INSERIR_OK` usada no exemplo anterior:

```
ALUNO_INSERIR_OK=Aluno {0} inserido com sucesso
ALUNO_ALTERAR_OK=Aluno {0} alterado com sucesso
ALUNO_EXCLUIR_OK=Aluno {0} excluído com sucesso
```

A fim de prover internacionalização para o idioma inglês americano (i.e., `en-US`) no exemplo em questão, eis o conteúdo do arquivo `messages_en_US.properties`:

```
ALUNO_INSERIR_OK=Student {0} was successfully added
ALUNO_ALTERAR_OK=Student {0} was successfully modified
ALUNO_EXCLUIR_OK=Student {0} was successfully removed
```

Similarmente, o arquivo `messages_fr.properties` se destina a prover as mensagens no idioma francês (independente do país), contendo as linhas a seguir:

```
ALUNO_INSERIR_OK=L'étudiant {0} a été ajouté avec succès
ALUNO_ALTERAR_OK=L'étudiant {0} a été modifié avec succès
ALUNO_EXCLUIR_OK=L'étudiant {0} a été supprimé avec succès
```

### Nota

As chaves das mensagens (ex: `ALUNO_INSERIR_OK`) devem ser as mesmas em todos os arquivos de propriedades. São elas que identificam unicamente uma mensagem, que ao final do processo tem o seu texto automaticamente traduzido para o idioma preferido do usuário.

### Dica

É possível ainda configurar de modo genérico o idioma, especificando apenas a língua. Por exemplo, ao invés de inglês britânico (`en_GB`) e inglês americano (`en_US`), podemos simplesmente definir o idioma inglês (`en`). Neste caso, o arquivo deverá ser o `messages_en.properties`.

### Nota

Internamente o *Demoiselle Framework* usa a classe *java.util.Locale* [http://download.oracle.com/javase/6/docs/api/java/util/Locale.html] presente na API do Java para manipular os textos das mensages durante a internacionalização.

### Nota

A seleção de idioma na internacionalização de uma aplicação consiste na definição de:

- língua: códigos formados por duas letras em minúsculo listados na ISO-639 (ISO Language Code) - exemplos: pt, en, fr, es, de;
- país: códigos formados por duas letras em maiúsculo listados na ISO-3166 (ISO Country Code) - exemplos: BR, PT, US, GB.

## 11.5. Destino das mensagens

O Framework Demoiselle permite configurar o destino das mensagens enviadas. Por padrão, mensagens enviadas em um ambiente SE (Swing por exemplo) são exibidas como registros de log no console, já mensagens enviadas em um ambiente WEB usando JSF 2.0 são redirecionadas para a classe `FacesContext`. Caso esse comportamento padrão não seja suficiente para você, é possível personalizar o mecanismo de redirecionamento de mensagens, fazendo-o enviar as mensagens para um local de seu interesse.

Para isso existe a interface `MessageAppender`. Para toda mensagem enviada, o Framework Demoiselle vai determinar a implementação mais adequada de `MessageAppender` a utilizar e vai redirecionar qualquer mensagem para essa implementação.

```
public interface MessageAppender extends Serializable {

    /**
     * Method that must hold message in an appropriate way and in an appropriate local.
     * Demoiselle holds a message in a Logger or in a FacesContext, depending on the project.
```

```
     *
     * @param message
     *          message to be stored.
     */
    void append(Message message);
}
```

Para criar seu próprio `MessageAppender`, implemente essa interface e anote-a com a anotação `@Priority` - o Framework Demoiselle irá selecionar a implementação adequada paseada na maior prioridade. Não é necessário configurar mais nada, o Framework Demoiselle selecionará a implementação automaticamente. Cabe-lhe então a tarefa de implementar o método `append(Message message)` para tratar a mensagem da forma que melhor se adequar a seu projeto.

## 11.6. Exemplos de implementação

A fim de auxiliar a manutenção das descrições das mensagens em uma aplicação diversas soluções podem ser escolhidas pelos arquitetos e empregadas pelos desenvolvedores. Nesta seção serão mostradas duas sugestões para essa questão, usando interfaces e enumeradores.

A vantagem em se utilizar ambas as soluções é que diversas mensagens relacionadas podem ser agrupadas, reduzindo assim a quantidade de arquivos a serem mantidos e centralizando a configuração das mensagens.

Sendo assim, uma possível solução é criar uma interface destinada a armazenar todos os modelos de mensagens de uma determinada severidade a ser usada na aplicação. Nesta interface são declarados campos do tipo `Message` com o modificador `final` (implicitamente também será `public` e `static`). Veja o exemplo de código para a interface `InfoMessages`:

```
public interface InfoMessages {

    final Message BOOKMARK_DELETE_OK = new DefaultMessage("{BOOKMARK_DELETE_OK}");
    final Message BOOKMARK_INSERT_OK = new DefaultMessage("{BOOKMARK_INSERT_OK}");
    final Message BOOKMARK_UPDATE_OK = new DefaultMessage("{BOOKMARK_UPDATE_OK}");
}
```

No exemplo em questão, o texto das mensagens será recuperado do arquivo de recursos `messages.properties` presente no diretório `/src/main/resources/`. Eis o conteúdo desse arquivo:

```
BOOKMARK_DELETE_OK=Bookmark exclu\u00EDdo\: {0}
BOOKMARK_INSERT_OK=Bookmark inserido: {0}
BOOKMARK_UPDATE_OK=Bookmark atualizado: {0}
```

### Dica

Recomenda-se criar uma interface para cada tipo de severidade (ex: `InfoMessages`, `WarningMessages`, `ErrorMessages` e `FatalMessages`), agrupando nestas as mensagens usadas exclusivamente para o mesmo fim.

Já a segunda abordagem consiste no uso de enumerações para centralizar os modelos de mensagem. É utilizado o mesmo arquivo de recursos `messages.properties` ilustrado na abordagem anterior. Porém, neste caso cada enumerador da enumeração corresponderá a uma chave no arquivo de propriedades.

```
public enum InfoMessages implements Message {

    BOOKMARK_DELETE_OK,
    BOOKMARK_INSERT_OK,
    BOOKMARK_UPDATE_OK;

    private final DefaultMessage msg;

    private InfoMessages() {
        msg = new DefaultMessage("{" + this.name() + "}");
    }

    @Override
    public String getText() {
        return msg.getText();
    }

    @Override
    public SeverityType getSeverity() {
        return msg.getSeverity();
    }
}
```

A fim de adicionar mensagens ao contexto, eis um exemplo de código que faz uso do artefato `InfoMessages`, que funciona não importa qual abordagem escolhida (interface ou enumeração):

```
@BusinessController
public class BookmarkBC {

    @Inject
    private MessageContext messageContext;                          (1)

    public void insert(Bookmark bookmark) {
        ...
        messageContext.add(InfoMessages.BOOKMARK_INSERT_OK,         (2)
            bookmark.getDescription());
    }

    public void update(Bookmark bookmark) {
        ...
        messageContext.add(InfoMessages.BOOKMARK_UPDATE_OK,         (2)
            bookmark.getDescription());
    }

    public void delete(Bookmark bookmark) {
        ...
        messageContext.add(InfoMessages.BOOKMARK_DELETE_OK,         (2)
            bookmark.getDescription());
    }
```

```
}
```

1. No ponto contendo `@Inject` será injetado via CDI o contexto de mensagens presente na aplicação, ou seja, uma instância da interface `MessageContext` que poderá ser utilizada em qualquer método nessa classe.
2. Aqui os métodos `insert()`, `update()` e `delete()` na classe `BookmarkBC` manipulam o contexto de mensagens em cada invocação destes. O método `add()` de `MessageContext` faz com que a mensagem passada como parâmetro seja adicionada ao contexto, que ao final é exibida para o usuário na camada de apresentação.

# Resource Bundle

Um dos requisitos para se construir uma aplicação nos dias de hoje é o de que seja utilizada por pessoas em vários lugares no mundo e em diferentes línguas. Portanto, é preciso que as aplicações sejam facilmente internacionalizáveis. Para isso, existe um recurso no java chamado de *Resource Bundle*, que nada mais é do que um esquema de arquivos `properties`, onde cada arquivo representa uma língua e cada um desses arquivos possui um conjunto de chaves e valores, sendo que os valores são os textos que serão exibidos na aplicação e estão na língua correspondente à língua que o arquivo representa.

O arquivo `properties` que será utilizado para montar a aplicação é escolhido pelo próprio usuário, seja através da língua definida no browser ou no próprio sistema operacional. Caso o usuário escolha uma língua que não está disponível na aplicação, uma língua default será utilizada. Por exemplo: vamos imaginar que em uma aplicação existem dois arquivos `properties`, um em português e outro em inglês, e que o arquivo default é o inglês. Vamos imaginar também que a aplicação é Web, portanto a língua escolhida está definida no próprio browser. Caso esteja configurado no browser do usuário a língua alemã e como não existe nenhum arquivo de properties para alemão, a aplicação será exibida na língua inglesa, que é a língua configurada como default.

Todos os arquivos são criados praticamente com o mesmo nome. O que diferencia um arquivo do outro é o acréscimo da sigla que representa a língua daquele arquivo. O arquivo que representa a língua default não tem essa sigla ao fim do nome. Seguindo o exemplo citado acima e imaginando que o nome dos nossos arquivos é messages, ficaria da seguinte forma: `messages.properties` seria o arquivo default que representaria a língua inglesa e `messages_pt.properties` seria o arquivo da língua portuguesa. Veja abaixo um exemplo com esses dois arquivos.

`messages.properties`:

```
button.edit=Edit
button.new=New
button.save=Save
```

`messages_pt.properties`:

```
button.edit=Editar
button.new=Novo
button.save=Salvar
```

## 12.1. Utilizando Resource Bundle no Demoiselle

Na versão 2 do *Demoiselle Framework*, existe uma fábrica de Resource Bundle que fica no Core e permite seu uso através da injeção ou através de uma instanciação normal. O grande detalhe é que nessa fábrica é injetado um objeto do tipo `Locale`, isso quer dizer que é necessário criar também uma fábrica de `Locale`. Como a definição de `Locale` varia de acordo com a camada de apresentação, essas fábricas foram criadas nas extensões de apresentação: `demoiselle-servlet`, `demoiselle-jsf` e `demoiselle-se`. Na extensão `demoiselle-se` a definição do `Locale` é dada através do `Locale` definido na máquina do usuário, enquanto que nas extensões `demoiselle-servlet` e `demoiselle-jsf` essa definição acontece através do `Locale` do browser do usuário, por se tratarem de extensões para camada de apresentação Web. Por default, a fábrica de Resource Bundle vai injetar um bundle apontando para o arquivo `messages`, mas isso pode ser facilmente alterado através da anotação `@Name`. Veja abaixo como utilizar o Resource Bundle no Demoiselle.

Utilizando Resource Bundle através da injeção:

```
@Inject
@Name("messages-core")
private ResourceBundle bundle;

public String metodoQueRetornaOValorDaChavebuttonedit() {
    return bundle.getString("button.edit");
}
```

Utilizando Resource Bundle sem uso de injeção:

```
private ResourceBundleFactory bundleFactory = new ResourceBundleFactory(Locale.getDefault());
private ResourceBundle bundle;

public String metodoQueRetornaOValorDaChavebuttonedit() {
    bundle = bundleFactory.create("messages-core");
    return bundle.getString("button.edit");
}
```

# Parâmetro

É muito comum em uma aplicação web haver a necessidade de passar parâmetros através da URL. A passagem de parâmetros até que é algo fácil e tranquilo de fazer. O chato é a captura do parâmetro dentro do Page Bean, pois toda vez que quisermos fazê-lo, temos que acessar o `FacesContext`, para a partir daí pegar o `HttpServletRequest` e depois recuperar o valor passado. Veja abaixo como ficaria o código.

Passagem do parâmetro:

```
http://localhost:8080/aplicacao/pagina.jsf?parametro=valorParametro
```

Captura do parâmetro pelo Page Bean

```
public class Classe {

    public void metodo() {
        FacesContext context = FacesContext.getCurrentInstance();
       HttpServletRequest req = (HttpServletRequest) context.getExternalContext().getRequest();
        String param = req.getParameter("parametro");
    }
}
```

## 13.1. Passagem de parâmetros

Visando facilitar essa recuperação do parâmetro passado através da URL, foi disponibilizada na versão 2.X do *Demoiselle Framework* uma funcionalidade através da interface `Parameter` e de sua implementação `ParameterImpl`, que permite capturar esse valor através do uso de injeção. Para isso, basta criar no seu Page Bean um objeto do tipo `br.gov.frameworkdemoiselle.util.Parameter` e anotá-lo com `@Inject`. O nome desse objeto é o nome que será usado para buscar o valor do parâmetro. Caso o usuário queira dar um nome diferente ao objeto, ele pode anotá-lo com `@Name` e no valor dessa anotação, colocar o nome do parâmetro. Por default o objeto criado tem o escopo de request, mas é possível usar o escopo de sessão ou de visão, bastando anotar o objeto com `@SessionScoped` ou `@ViewScoped`, respectivamente. Veja abaixo como ficaria essa passagem de parâmetros na versão 2.X do Demoiselle.

Passagem do parâmetro:

```
http://localhost:8080/aplicacao/pagina.jsf?parametro=1
http://localhost:8080/aplicacao/pagina.jsf?parametroString=valorParametroString
```

Captura do parâmetro pelo Page Bean:

```
public class Classe {

    @ViewScoped
    @Inject
    private Parameter<Long> parametro;
```

```
    @Name("parametroString")
    @SessionScoped
    @Inject
    private Parameter<String> objetoComNomeDiferenteDoParametro;
}
```

## 13.2. As classes de parâmetro

A interface `Parameter` e sua implementação `ParameterImpl` disponibilizam alguns métodos, como `setValue(T value)`, `getKey()`, `getValue()` e `getConverter()`, que servem respectivamente para atribuir o valor do objeto, capturar o nome do parâmetro passado na URL, recuperar o valor passado para aquele parâmetro e capturar o conversor de tipo utilizado. Logo, para usar o valor daquele objeto, basta utilizar o método `getValue()`, tal como mostrado a seguir:

```
public class Classe {

    @ViewScoped
    @Inject
    private Parameter<Long> parametro;

    public void metodo() {
        System.out.println("Valor do parametro: " + parametro.getValue());
    }
}
```

# Logger

Uma API de logging é um recurso interessante para informar o usuário e o desenvolvedor sobre os erros, alertas, resultados de operações, os recursos acessados e outras informações a respeito do sistema. Por esse motivo foi implementada na versão 2.X do *Demoiselle Framework* uma fábrica de logger que fica no core e visa permitir a injeção desse recurso baseado na classe `org.slf4j.Logger`. Veja abaixo como usar o logger no *Demoiselle*:

```
public class MinhaClasse {

    @Inject
    private Logger logger;

    public void meuMetodo() {
        logger.debug("logando meu metodo");
        logger.warn("mensagem de alerta do meu metodo");
    }
}
```

# Templates

O *Demoiselle Framework* provê abstrações de classes para as camadas de apresentação, negócio e persistência. Tais classes podem ser encontradas no pacote `br.gov.frameworkdemoiselle.template` e auxiliam o desenvolvedor ao disponibilizar métodos comuns à maioria das aplicações. A seguir iremos exemplificar o uso de cada uma delas.

## 15.1. Camada de persistência

A classe abstrata `JPACrud` implementa as operações básicas de inclusão, remoção, atualização e recuperação de registros no banco de dados. Sendo assim, possibilita que o desenvolvedor concentre-se na criação de métodos específicos para atender as regras de negócio da sua aplicação. Esta classe recebe dois parâmetros em notação de genéricos:

- `T` representa a entidade que será tratada
- `I` representa o tipo do identificador da entidade

No exemplo abaixo demonstra-se a utilização da `JPACrud`, onde o desenvolvedor precisou implementar apenas um método específico.

```
@PersistenceController
public class SuaEntidadeDAO extends JPACrud<SuaEntidade, Long> {

    public List<SuaEntidade> findByAlgumCriterioEspecifico(final String criterio) {
          Query query = getEntityManager().createQuery("select se from SuaEntidade se where
 se.criterio = :criterio ");
        query.setParameter("criterio", criterio);
        return query.getResultList();
    }
}
```

## 15.2. Camada de negócio

De forma semelhante à classe `JPACrud`, a classe *DelegateCrud* foi criada com o intuito de dispensar o desenvolvedor de implementar métodos que serão comuns à maioria das entidades. Além disso, esta classe implementa a injeção de dependência entre as camadas de negócio e persistência. Para utilizá-la, três parâmetros devem ser passados em notação de genéricos:

- `T` representa a entidade
- `I` representa o tipo do identificador da entidade
- `C` representa uma classe que implemente a interface `CRUD`

Segue abaixo um exemplo da utilização do *DelegateCrud*. Neste caso, foi implementado um método para validar a entidade antes de proceder com a inclusão no banco de dados. Para isso, o método `insert()` fornecido pela classe foi sobrescrito.

```
@BusinessController
```

```
public class SuaEntidadeBC extends DelegateCrud<SuaEntidade, Long, SuaEntidadeDAO> {

    private void validateInsert(SuaEntidade se) {
        // valida os atributos da entidade
    }

    @Override
    @Transactional
    public void insert(SuaEntidade se) {
        validateInsert(se);
        super.insert(se);
    }
}
```

## 15.3. Camada de apresentação

Para a camada de apresentação, existem duas classes que implementam os comportamentos básicos de navegação para páginas de listagem e edição. Estas classes são `AbstractListPageBean` e `AbstractEditPageBean`, respectivamente. De forma semelhante à `DelegateCrud`, estas classes realizam a injeção de dependência da camada de negócio dentro do artefato da camada de apresentação. Ambas recebem dois parâmetros em notação de genéricos:

- `T` representa a entidade que será tratada
- `I` representa o tipo do identificador da entidade

Estendendo o `AbstractListPageBean`:

```
@ViewController
public class SuaEntidadeListMB extends AbstractListPageBean<SuaEntidade, Long> {
}
```

Estendendo o `AbstractEditPageBean`:

```
@ViewController
public class SuaEntidadeEditMB extends AbstractEditPageBean<SuaEntidade, Long> {
}
```

# Segurança

Neste capítulo será tratada uma questão de grande importância para a maioria das aplicações e motivo de infindáveis discussões nas equipes de desenvolvimento: controle de acesso. Assim como tudo relacionado ao framework, a implementação de segurança foi projetada de forma simples e flexível, independente de camada de apresentação ou tecnologia, te deixando livre para implementar sua própria solução ou utilizar as extensões existentes.

Para utilizar o modelo de segurança proposto basta utilizar o Framework Demoiselle, pois no núcleo do framework estão as interfaces e anotações que definem o comportamento básico da implementação.

## 16.1. Configurando

Para um correto funcionamento do Demoiselle é necessário inserir os interceptadores de segurança no arquivo `src/main/webapp/WEB-INF/beans.xml`.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <interceptors>
        <class>br.gov.frameworkdemoiselle.security.RequiredPermissionInterceptor</class>
        <class>br.gov.frameworkdemoiselle.security.RequiredRoleInterceptor</class>
    </interceptors>
</beans>
```

Opcionalmente é possível configurar o comportamento do módulo de segurança definindo propriedades no arquivo *demoiselle.properties* da sua aplicação.

### Tabela 16.1. Propriedades de segurança do Framework Demoiselle

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.security.enabled | Habilita ou desabilita o mecanismo de segurança | true |
| frameworkdemoiselle.security.authenticator.class | Define a classe que implementa o mecanismo de autenticação. (Detalhes na seção *Criando sua implementação*) | - |
| frameworkdemoiselle.security.authorizer.class | Define a classe que implementa o mecanismo de autorização. (Detalhes na seção *Criando sua implementação*) | - |

## 16.2. Autenticação

O mecanismo de autenticação busca verificar a identidade do usuário de um sistema. A forma mais conhecida - e comum - para executar essa verificação se dá por meio de um formulário de login, geralmente solicitando um nome de usuário e sua respectiva senha. No entanto, outras formas como reconhecimento biométrico e autenticação por token, para citar apenas duas, tem ganhado um grande número de adeptos.

O Framework Demoiselle deixa o desenvolvedor livre para definir qual forma usar, de acordo com a sua conveniência e necessidade. A peça chave para tornar isso possível é o contexto de segurança, representado

pela interface `SecurityContext`. Nessa estão definidos os métodos responsáveis por gerenciar os mecanismos de autenticação como, por exemplo, executar login/logout de usuários e verificar se os mesmos estão ou não autenticados.

Para utilizar o `SecurityContext`, basta injetá-lo em seu código. O método `login` ativa o mecanismo de autenticação e o método `logout` remove as credenciais atualmente autenticadas do sistema. A classe `SecurityContext` possui outros métodos que permitem verificar se há um usuário autenticado e acessar o objeto *gerente* (representado pela classe `javax.security.Principal`), um objeto que contém dados adicionais sobre o usuário atualmente autenticado. Consulte a documentação da classe `SecurityContext` para consultar as funcionalidades que ela oferece.

Um exemplo do uso do `SecurityContext` para autenticação segue abaixo:

```
public class ExemploAutenticacao {

    @Inject
    private SecurityContext securityContext;

    public void efetuarAutenticacao() {
        /*
      Obtem as credenciais do usuario, pode ser um login e senha ou um certificado digital. O mais
        comum e exibir uma tela HTML contendo um formulario que solicita as informacoes.
        */

        try{
            securityContext.login();

            //Executa codigo que requer autenticacao

            securityContext.logout();
        }
        catch(InvalidCredentialsException exception){
            //Trata credenciais invalidas
        }

    }
}
```

# 16.3. Autorização

Em certos sistemas é necessário não apenas autenticar um usuário, mas também proteger funcionalidades individuais e separar usuários em grupos que possuem diferentes autorizações de acesso. O mecanismo de autorização é responsável por garantir que apenas usuários autorizados tenham o acesso concedido a determinados recursos de um sistema.

No modelo de segurança do Framework Demoiselle, a autorização pode acontecer de duas formas:

- Permissão por funcionalidade, através da anotação @RequiredPermission
- Permissão por papel, através da anotação @RequiredRole

## 16.3.1. Protegendo o sistema com *@RequiredPermission*

A anotação `@RequiredPermission` permite marcar uma classe ou método e informar que acesso a esse recurso requer a permissão de executar uma *operação*. Operação nesse contexto é um nome definido pelo desenvolvedor

que representa uma funcionalidade do sistema. Por exemplo, determinada classe pode ter métodos responsáveis por criar, editar, listar e remover bookmarks, o desenvolvedor pode decidir agrupar esses métodos sobre a operação *gerenciar bookmark*.

```
class GerenciadorBookmark {

    @RequiredPermission(resource = "bookmark" , operation = "gerenciar")
    public void incluirBookmark(Bookmark bookmark) {
        //Ccdigo do metodo
    }

    @RequiredPermission(resource = "bookmark", operation = "gerenciar")
    public List<Bookmark> listarBookmarks() {
        //Codigo do metodo
    }

    @RequiredPermission
    public List<Bookmark> apagarBookmark(Long idBookmark) {
        public List<Bookmark> listarBookmarks() {
    }
}
```

### Dica

Perceba que a anotação `@RequiredPermission` sobre o método `apagarBookmark` não contém parâmetros. Quando não são passados parâmetros o valor padrão para o parâmetro `resource` é o nome da classe e o valor padrão para `operation` é o nome do método.

### Dica

É possível anotar a classe inteira com `@RequiredPermission`, isso protegerá o acesso a todos os métodos dessa classe.

```
@RequiredPermission(resource="bookmark" , operation="gerenciar")
class GerenciadorBookmark {

    public void incluirBookmark(Bookmark bookmark) {
        //Codigo do metodo
    }

    public List<Bookmark> listarBookmarks() {
        //Codigo do metodo
    }

    public List<Bookmark> apagarBookmark(Long idBookmark) {
        public List<Bookmark> listarBookmarks() {
    }
}
```

## 16.3.2. Protegendo o sistema com *@RequiredRole*

Diferente de `@RequiredPermission`, a anotação `@RequiredRole` utiliza o conceito de papéis - ou perfís - para proteger recursos. Uma classe ou método anotado com `@RequiredRole` exigirá que o usuário autenticado possua o papel indicado para acessar o recurso.

Voltando ao exemplo de nosso aplicativo de bookmarks, vamos supor que a função de listar os bookmarks existentes pode ser acessada por qualquer usuário autenticado, mas apenas administradores podem criar um novo bookmark. A classe responsável por tais funcionalidades pode ser criada da seguinte forma:

```
class GerenciadorBookmark {

    @RequiredRole("administrador")
    public void inserirBookmark(Bookmark bookmark) {
    }

    @RequiredRole({"convidado" , "administrador"})
    public List<Bookmark> listarBookmarks() {
    }

}
```

**Dica**

É possível informar mais de um papel para a anotação `@RequiredRole`, neste caso basta que o usuário autenticado possua um dos papéis listados para ter acesso ao recurso.

Da mesma forma que a anotação `@RequiredPermission`, a anotação `@RequiredRole` pode ser usada a nível de classe para proteger todos os métodos contidos nessa classe.

## 16.3.3. Protegendo porções do código

É possível proteger apenas parte de um código ao invés de todo o método ou toda a classe. Isso pode ser necessário em expressões condicionais, onde um trecho só deve ser executado caso o usuário possua a autorização necessária. Para isso voltamos a usar a interface `SecurityContext`, pois ela contém métodos que são funcionalmente equivalentes às anotações `@RequiredPermission` e `@RequiredRole`.

Como um exemplo, vamos supor que ao remover um bookmark um email seja enviado ao administrador, mas se o próprio administrador executou a operação não é necessário enviar o email.

```
class GerenciadorBookmark {

    @Inject
    private SecurityContext securityContext;

    public void removerBookmark(Long idBookmark) {

        //Codigo que remove o bookmark

        if  ( ! securityContext.hasRole("administrador") ){
            //Envia um email ao administrador
        }
```

```
    }

}
```

### 16.3.4. Protegendo porções de páginas `Java Server Faces`

As restrições de segurança podem ser utilizadas ainda em páginas web, com o auxílio de Expression Language. A interface `SecurityContext` está automaticamente disponível para páginas `Java Server Faces` como um *bean* de nome `securityContext`, bastando então acessar seus métodos a partir deste bean.

```
<p:commandButton value="#{messages['button.save']}" action="#{contactEditMB.insert}"
    ajax="false" disabled="#{!securityContext.hasPermission('contact', 'insert')}" />
```

Nesse caso, a habilitação de um botão está condicionada à existência de permissão para o usuário autenticado no momento executar a operação “insert” no recurso “contact”.

## 16.4. Redirecionando automaticamente para um formulário de acesso

Se sua aplicação usa a extensão *demoiselle-jsf* ou se você utilizou o arquétipo *demoiselle-jsf-jpa* durante a criação de seu projeto, então você pode definir uma página de login e o Framework Demoiselle vai automaticamente lhe redirecionar para essa página caso haja a tentativa de acessar um recurso protejido e nenhum usuário esteja autenticado no sistema.

**Dica**

Para acrescentar a extensão *demoiselle-jsf* em um projeto Maven, adicione a dependência abaixo no arquivo *pom.xml*.

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle</groupId>
  <artifactId>demoiselle-jsf</artifactId>
  <scope>compile</scope>
</dependency>
```

O arquétipo *demoiselle-jsf-jpa* já contém essa extensão, se você criou seu projeto baseado nesse arquétipo nada precisa ser feito.

Por padrão a página contendo o formulário de login deve se chamar *login.jsp* ou *login.xhtml* (a depender de como sua aplicação esteja configurada para mapear páginas JSF). Para mudar esse padrão, é possível editar o arquivo *demoiselle.properties* para configurar qual página deve ser utilizada.

### Tabela 16.2. Propriedades de segurança da extensão *demoiselle-jsf*

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.security.login.page | Define a página de login da aplicação. | "/login" |
| frameworkdemoiselle.security.redirect.after.login | Define a tela para qual o usuário será redirecionado após o processo de *login* bem sucedido. | "/index" |
| frameworkdemoiselle.security.redirect.after.logout | Define a tela para qual o usuário será redirecionado após o processo de *logout* bem sucedido. | "/login" |
| frameworkdemoiselle.security.redirect.enabled | Habilita ou desabilita o redirecionamento automático para a página de login após uma tentativa de acessar recurso protegido. | true |

## 16.5. Integrando o Framework Demoiselle com a especificação JAAS

Até agora vimos como criar código protegido em uma aplicação Demoiselle, mas nada foi dito sobre a tecnologia que implementa essa proteção. A verdade é que o Framework Demoiselle dá ao desenvolvedor a liberdade de implementar a solução que mais se adequa ao sistema desenvolvido, mas o framework também conta com suporte nativo à especificação JAAS (*Java Authentication and Authorization Service*).

O suporte a JAAS é fornecido para aplicações WEB e está implementado na extensão *demoiselle-servlet*, então é necessário declarar a dependência a essa extensão em sua aplicação.

**Dica**

Para acrescentar a extensão *demoiselle-servlet* em um projeto Maven, adicione a dependência abaixo no arquivo *pom.xml*.

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle</groupId>
  <artifactId>demoiselle-servlet</artifactId>
  <scope>compile</scope>
</dependency>
```

**Dica**

O arquétipo *demoiselle-jsf-jpa* já conta com a dependência à extensão *demoiselle-jsf*, que por sua vez depende da extensão *demoiselle-servlet*. Se sua aplicação é baseada no arquétipo *demoiselle-jsf-jpa* você já possui a extensão *demoiselle-servlet*.

Uma vez que sua aplicação contenha a extensão *demoiselle-servlet*, tudo que você precisa fazer é configurar o suporte a JAAS em seu servidor de aplicação e criar os usuários e papéis necessários. Esta configuração depende do servidor de aplicação utilizado e foge ao escopo deste documento.

Para autenticar um usuário presente no servidor de aplicação através do JAAS, a extensão *demoiselle-servlet* oferece a classe `Credentials`, que deve ser injetada em seu código. O código abaixo mostra como realizar a autenticação a partir de um servlet.

```
class LoginServlet extends HttpServlet {

    @Inject
    private SecurityContext securityContext;

    @Inject
    private Credentials credentials;

    public void doPost(HttpServletRequest req, HttpServletResponse resp) {

        credentials.setUsername( req.getParameter("username") );
        credentials.setPassword( req.getParameter("password") );

        securityContext.login();

    }

}
```

Uma vez autenticado o usuário, a anotação `@RequiredRole` passará a verificar se o usuário presente no JAAS possui o papel informado.

## Cuidado

A especificação JAAS não prevê o uso de permissões para proteger recursos, apenas papéis de usuários. Por isso ao utilizar a segurança da especificação JAAS o uso da anotação `@RequiredPermission` fica vetado. Utilizar essa anotação em um sistema que utilize JAAS para autorização causará uma exceção quando o recurso for acessado.

## Dica

É possível utilizar o JAAS para autenticar e autorizar papéis de usuários mas criar sua própria implementação para implementar a autorização de permissões. Para isso crie uma classe que herde a classe `br.gov.frameworkdemoiselle.security.ServletAuthorizer` e sobrescreva o método `hasPermission(String resource, String operation)` para implementar seu próprio mecanismo. Feito isso, basta definir sua classe no arquivo *demoiselle.properties* usando a propriedade `frameworkdemoiselle.security.authorizer.class`.

Mais detalhes sobre como criar sua própria implementação ou extender uma implementação existente podem ser vistos na seção *Criando sua implementação*.

## 16.6. Criando sua implementação

Para os mecanismos de autenticação não cobertos pelo Framework Demoiselle, é possível criar sua própria implementação e integra-la ao framework. Também é possível extender uma implementação existente e acrescentar funcionalidades inexistentes.

O ponto de extensão para o módulo de segurança são as interfaces `Authenticator` e `Authorizer`. Para criar um novo mecanismo de autenticação e autorização, é necessário apenas implementar essas duas interfaces em sua aplicação. Segue abaixo um exemplo de implementação.

```
public class MeuAuthenticator implements Authenticator {

    @Override
    public void authenticate() throws Exception {
        // Execute o procedimento necessario para autenticar o usuario
       // Apos isso, a chamada ao metodo getUser() deve returnar o usuario corrente autenticado,
        // e ira retornar nulo (null) se o processo de autenticacao falhar
    }

    @Override
    public User getUser(){
      // Retorna o usuario que esta autenticado, ou nulo (null) se nao houver usuario autenticado
    }

    @Override
    public void unauthenticate() throws Exception {
        // Execute o procedimento necessario para desautenticar um usuario
        // Apos isso,  a chamada ao metodo getUser() deve returnar nulo (null)
    }
}
```

```
public class MeuAuthorizer implements Authorizer {

    @Override
    public boolean hasRole(String role) throws Exception {
      // Verifique se o usuario autenticado tem o papel informado, retorne true em caso positivo
        return false;
    }

    @Override
    public boolean hasPermission(String resource, String operation) throws Exception {
        // Escreva aqui seu codigo de verificacao de permissao
        return false;
    }
}
```

### Nota

Fique atento! Note que para garantir que o usuário não esteja logado, não basta lançar uma exceção (InvalidCredencialsException) no método authenticator.autenticate(), mas é preciso que

o método authenticator.getUser() retorne null para que o método securityContext.isLoggedIn() retorne false.

Pronto! Sua aplicação já possui uma implementação de segurança definida.

## Dica

Você nunca deve chamar diretamente em sua aplicação as implementações das interfaces `Authenticator` e `Authorizer`, o Framework Demoiselle vai automaticamente chamar os métodos implementados quando for necessário.

Em um sistema que use as anotações `@RequiredRole` ou `@RequiredPermission`, deve haver pelo menos uma implementação dessas duas interfaces. Ao processar essas anotações, o Framework Demoiselle vai buscar uma implementação para essas interfaces e disparar uma exceção caso não encontre uma implementação adequada.

Se existe mais de uma implementação de `Authenticator` e/ou `Authorizer` (o que pode acontecer, por exemplo, quando seja necessário uma implementação na aplicação principal e outra para os testes), é possível definir no arquivo `demoiselle.properties` a classe que deve ser usada por padrão:

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.security.authenticator.class | Define a classe concreta utilizada como implementação da interface Authenticator | *nenhum, se houver apenas uma implementação o framework a detectará automaticamente sem necessidade de definir essa propriedade* |
| frameworkdemoiselle.security.authorizer.class | Define a classe concreta utilizada como implementação da interface Authorizer | *nenhum, se houver apenas uma implementação o framework a detectará automaticamente sem necessidade de definir essa propriedade* |

# Paginação

Neste capítulo serão considerados os seguintes assuntos:

- Motivação para o uso de um mecanismo padronizado para *paginação*;
- Funcionamento e uso da estrutura `Pagination` e do contexto de paginação (`PaginationContext`).

## 17.1. Introdução ao mecanismo

A apresentação de conjuntos de registros de médio a grande porte em formato de tabelas em aplicações Web geralmente requer um *mecanismo de paginação*, o qual permite ao cliente ver apenas um pedaço do resultado final, podendo este navegar para frente e para trás através dos registros. A menos que o conjunto de registros seja garantidamente pequeno, qualquer aplicação do tipo Web com funcionalidades de busca e/ou listagem de registros, precisa ser dotada de paginação.

O mecanismo de paginação para as aplicações fornecido pelo *Demoiselle Framework* consiste em um algoritmo direcionado ao banco de dados (i.e., Database-Driven Pagination). Essa abordagem, apesar de requerer instruções SQL específicas para obter porções determinadas de registros, é a mais utilizada por ser mais eficiente e produzir menos redundância de dados, diminuindo assim tráfego de rede, consumo de memória e reduzindo o tempo de resposta.

É fornecido em tempo de execução um *contexto de paginação*, o qual tem escopo de sessão e armazena a informação de paginação de cada entidade (i.e., bean) que necessite de tal mecanismo. Esse contexto é compartilhado entre as diversas camadas da aplicação, especificamente entre as camadas de visão e persistência. Dessa maneira, a paginação dos dados é transparente para a camada intermediária (i.e., negócio) e não interfere na modelagem das classes de um projeto.

## 17.2. Códigos de suporte

O *mecanismo de paginação* do *Demoiselle Framework* permite que os parâmetros para a consulta no banco sejam configurados de forma bastante prática. Por outro lado, a consulta paginada ao banco já é feita pela extensão `demoiselle-jpa`. Dessa forma, basta ajustar os parâmetros da paginação, e pedir as consultas normalmente. O resultado da consulta é então passado para algum componente de iteração de dados com suporte ao mecanismo conhecido como *Lazy Load* (ou *Lazy Loading*).

Farão parte do código de suporte para paginação:

- A classe `Pagination`: usada para manipular a paginação dos dados resultantes, contendo os campos `currentPage` (página atual, selecionada na camada de visão), `pageSize` (tamanho da página, a quantidade de registros que ela comportará) e `totalResults` (a quantidade de resultados existentes na base de dados);
- A classe `PaginationContext`: contexto usado para armazenar e fornecer estruturas do tipo `Pagination`;
- A classe `PaginationConfig`: armazenador de configurações referentes à paginação.

Códigos internos de suporte no Core:

```
public class Pagination {

    private int currentPage;
    private int pageSize;
    private Long totalResults;
    private Integer totalPages;
    // ...
```

```
}

@SessionScoped
public class PaginationContext {

    private final Map<Class<?>, Pagination> map;
    public Pagination getPagination(Class<?> clazz) { ... }
    public Pagination getPagination(Class<?> clazz, boolean create) { ... }
}

@Configuration
public class PaginationConfig {

    @Name("default_page_size")
    private int defaultPageSize = 10;

    @Name("max_page_links")
    private int maxPageLinks = 5;
}
```

Códigos internos de suporte em JPA:

```
public class JPACrud<T, I> implements Crud<T, I> {

    @Inject
    private PaginationContext paginationContext;
    // ...

    public List<T> findAll() {

        final String jpql = "select this from " + getBeanClass().getSimpleName() + " this";
        final Query query = getEntityManager().createQuery(jpql);
        final Pagination pagination = paginationContext.getPagination(getBeanClass());

        if (pagination != null) {
            if (pagination.getTotalPages() == null) {
                pagination.setTotalResults(this.countAll());
            }
            query.setFirstResult(pagination.getFirstResult());
            query.setMaxResults(pagination.getPageSize());
        }
        // ...
    }
}
```

Códigos internos de suporte em JSF:

```
public abstract class AbstractListPageBean<T, I> extends AbstractPage implements ListPageBean<T, I> {

    @Inject
    private PaginationContext paginationContext;

    @Inject
    private PaginationConfig paginationConfig;
```

```
    // ...

    public Pagination getPagination() {
        return paginationContext.getPagination(getBeanClass(), true);
    }

    public int getPageSize() {
        return paginationConfig.getDefaultPageSize();
    }

    public int getMaxPageLinks() {
       return paginationConfig.getMaxPageLinks();
    }
}
```

## 17.3. Implementação na aplicação

Veremos nessa seção como implementar a paginação em uma aplicação Java. Para esse exmplo tomamos como base a aplicação de Bookmarks fornecida pelo arquétipo *JSF com JPA* do *Demoiselle Framework* (para maiores detalhes ver *Arquétipos*). Iremos utilizar o componente `DataTable` do *PrimeFaces*, que oferece o mecanismo de *Lazy Loading* conhecido como `LazyDataModel`, muito útil para paginação e classificação de dados.

Primeiro é preciso configurar um objeto `LazyDataModel` no construtor do *Managed Bean* (*BookmarkList* nesse exemplo): instancia-lo e sobrescrever o método abstrado `load`, que recebe vários argumentos. Esses argumentos são recuperados na página `jsf` que carrega a instância do objeto `LazyDataModel`.

Dentro do método `load` iremos pegar do contexto de paginação uma instância da implementação da interface `Pagination` e ajustar alguns dos seus parâmetros para: indicar a partir de qual item a paginação deve iniciar, e o tamanho (quantidade de itens) de cada página. Esses dados são usados no método `findAll()`, da classe `JPACrud` (extensão JPA), que utiliza o contexto de paginação para pegar os parâmetros e fazer a consulta no banco buscando apenas os itens que estão dentro da página que o parâmetro `first` indicar. O resultado é passado para a instancia do `LazyDataModel`, que é responsável por exibir os dados de forma apropriada.

À classe *BookmarkList* devem ser adicionados os seguintes trechos de código:

```
// ...
import java.util.Map;
import br.gov.frameworkdemoiselle.pagination.Pagination;
// ...

public BookmarkListMB() {

    private LazyDataModel<Bookmark> lazyModel;
    lazyModel = new LazyDataModel<Bookmark>() {

        @Override
        public List<Bookmark> load (int first, int pageSize, String sortField,
                        SortOrder sortOrder, Map<String, String> filters){

            Pagination pagination = getPagination();
            pagination.setPageSize(pageSize);
            pagination.setFirstResult(first);

            List<Bookmark> itemsList = bc.findAll();
```

```
            lazyModel.setRowCount(pagination.getTotalResults());

            return itemsList;
        }
    };
    // ...
    public LazyDataModel<Bookmark> getLazyModel() {
        return lazyModel;
    }
    // ...
}
```

No arquivo `messages.properties` adicione as linhas:

```
page.first=0
page.rows=4
page.max.links=3
```

Na página JSF `bookmark_list.xhtml`, substitua a linha:

```
<p:dataTable id="list" var="bean" value="#{bookmarkListMB.resultList}">
```

por:

```
<p:dataTable id="list" var="bean"
value="#{bookmarkListMB.lazyModel}" lazy="true" paginator="true"
first="#{messages['page.first']}" rows="#{messages['page.rows']}"
pageLinks="#{messages['page.max.links']}">
```

Com essas alterações simples, a aplicação Bookmarks passa a utilizar o mecanismo de paginação oferecido pelo *Demoiselle Framework*.

## Dica

O método `getPagination()` do contexto `PaginationContext` é sobrecarregado, podendo aceitar os seguintes argumentos: `Class` ou `Class` e `boolean`.

## Nota

A JPA 2.0, através da Query API, suporta controle de paginação independente de fornecedor de banco de dados. Para controlar a paginação, a interface `Query` define os métodos `setFirstResult()` e `setMaxResults()` para especificar o primeiro resultado a ser recebido e o número máximo de resultados a serem retornados em relação àquele ponto. Internamente, são usadas instruções específicas do SGBD (ex: LIMIT e OFFSET no PostgreSQL).

# Monitoração e Gerenciamento de Recursos

## 18.1. Por que monitorar e gerenciar aplicações

Ao implantar um sistema para produção, muitas vezes é necessário monitorar aspectos sobre o funcionamento desse sistema. Quanta memória ele está utilizando? Qual o pico de MIPS utilizados? Quantas sessões estão autenticadas no momento?

Além de monitorar um sistema, as vezes é necessário gerenciá-lo alterando aspectos de seu comportamento. Se o sistema está implantado em um servidor alugado, talvez seja necessário ajustar o uso de MIPS para reduzir custos ou talvez deseje-se solicitar que o sistema limpe dados de sessão de autenticação abandonados por usuários que desligaram suas estações sem efetuar "logoff".

Para esse fim existem diversas tecnologias que permitem ao desenvolvedor expor aspectos monitoráveis e gerenciáves de seu sistema para clientes de gerenciamento. Exemplos dessas tecnologias incluem o *Simple Network Management Protocol* (SNMP) e o *Java Management Extension* (JMX).

O *Demoiselle Framework* dispõe de uma série de ferramentas para nivelar o conhecimento do desenvolvedor e facilitar o uso e integraçao de várias tecnologias de gerenciamento e monitoração. Através de seu uso o desenvolvedor pode facilmente integrar tais tecnologias, despreocupando-se com detalhes de implementação de cada uma delas.

## 18.2. Introdução ao mecanismo

Para expor aspectos monitoráveis da sua aplicação, o primeiro passo é criar uma interface contendo os atributos monitoráveis e as operações de gerenciamento que serão expostas para clientes de gerenciamento. Isso é feito através de uma simples classe Java (*ou POJO*) anotada com o estereótipo `@ManagementController`.

```
@ManagementController
public class GerenciadorUsuarios
```

Essa anotação é suficiente para o mecanismo de gerenciamento descobrir sua classe e disponibilizá-la para ser monitorada e gerenciada.

Contudo, a simples anotação acima não informa ao mecanismo quais aspectos da classe serão expostos. Por padrão, um *Management Controller* não expõe nenhum aspecto seu. Para selecionar quais aspectos serão expostos usamos as anotações *@ManagedProperty* e *@ManagedOperation*. Além disso outras anotações podem ser usadas para personalizar o funcionamento de classes anotadas com `@ManagementController`.

| Anotação | Descrição | Atributos |
|---|---|---|
| *@ManagedProperty* | Marca um atributo na classe como uma propriedade gerenciada, significando que clientes externos podem ler e/ou escrever valores nesses atributos.<br><br>Um atributo marcado pode estar disponível para leitura e/ou escrita. | • *description*: Um texto descritivo documentando o propósito da propriedade.<br><br>• *accessLevel*: Sobrescreve o nível padrão de acesso de uma propriedade. Os valores possíveis são READ_ONLY, WRITE_ONLY |

| Anotação | Descrição | Atributos |
|---|---|---|
| | Por padrão, o que determina a visibilidade de um atributo marcado é a presença dos métodos *getAtributo* e *setAtributo*, respectivamente disponibilizando o atributo para leitura e escrita.<br><br>Para sobrescrever esse comportamento existe na anotação *@ManagedProperty* o atributo *accessLevel*. Com ele é possível criar um atributo apenas para leitura, mas que contenha um método *set*. O contrário também é possível. | e DEFAULT, que significa que a presença de métodos *get* e *set* vai determinar o nível de acesso. |
| *@ManagedOperation* | Marca um método da classe gerenciada como uma operação, o que significa que clientes externos podem invocar esse método remotamente.<br><br>Operações gerenciadas normalmente são criadas para executar intervenções em um sistema já em execução. Por exemplo, é possível criar uma operação que, ao ser invocada, destrua todas as seções abertas no servidor e não utilizadas nos últimos 30 minutos. | • *description*: Um texto descritivo documentando o propósito da operação.<br><br>• *type*: Documenta o propósito da operação. *ACTION* informa que a operação modificará o sistema de alguma forma. *INFO* diz que a operação coletará e retornará informações sobre o sistema. *ACTION_INFO* informa que a operação modificará o sistema de alguma forma e retornará informações sobre o resultado. *UNKNOWN* é o padrão e significa que o resultado da execução da operação é desconhecido. |
| *@OperationParameter* | Esta anotação opcional pode ser usada para cada parâmetro de um método anotado com *@ManagedOperation*.<br><br>Ele permite detalhar melhor parâmetros em uma operação gerenciada. O efeito desta anotação é dependente da tecnologia utilizada para comunicação entre cliente e servidor. Na maioria das tecnologias, essa anotação meramente permite ao cliente exibir informações sobre cada parâmetro: nome, tipo e descrição. | • *name*: O nome do parâmetro quando exibido para clientes.<br><br>• *description*: Um texto descritivo documentando o propósito do parâmetro. |

## 18.3. Expondo aspectos de sua aplicação para monitoração

Uma vez que uma classe esteja anotada com *@ManagementController* e seus atributos e operações estejam expostos, a classe está pronta para ser monitorada.

Suponha que a aplicação deseje expor o número de usuários que efetuaram login. A operação de *login* será processada em uma classe de negócio *ControleAcesso*. Vamos supor também que existe uma classe chamada *MonitorLogin* responsável por expor o número de usuários que efetuaram login no sistema.

```
@BusinessController
public class ControleAcesso{

  @Inject
  private MonitorLogin monitorLogin;

  public boolean efetuarLogin(String usuario , String senha){
    // codigo de login
    monitorLogin.setContadorLogin( monitorLogin.getContadorLogin() + 1 );
  }
}
```

```
@ManagementController
public class MonitorLogin{

  @ManagedProperty
  private int contadorLogin;

  @ManagedOperation
  public void setContadorLogin(int qtdUsuarioLogados){
    contadorLogin = qtdUsuarioLogados;
  }

  @ManagedOperation
  public int getContatorLogin(){
    return contadorLogin;
  }
}
```

Como é possível ver, classes anotadas com *@ManagementController* podem ser injetadas em qualquer ponto do código. Valores definidos para seus atributos retêm seu estado, então um cliente que acesse remotamente o sistema e monitore o valor do atributo *contadorLogin* verá a quantidade de logins efetuados no momento da consulta.

### 18.3.1. Enviando notificações da aplicação

É comum que aplicações monitoradas permaneçam em estado de espera - é função do cliente de monitoração acessar a aplicação e obter as informações necessárias. No entanto existem casos onde é necessário que a aplicação comunique clientes de eventos ocorridos no sistema. Um exemplo é um monitor de espaço em disco que envia um alerta quando esse espaço for menor que 20% do total.

Para essa funcionalidade o *Demoiselle Framework* disponibiliza o utilitário *NotificationManager*, que pode ser injetado em sua aplicação a qualquer momento para notificar clientes externos de eventos importantes.

Considere o exemplo abaixo:

```
public class DiskWritter{

  @Inject
  private NotificationManager notificationManager;

  public void writeFile(byte[] data , File fileToWrite){
    // ... implementa##o da escrita de arquivo

    if (fileToWrite.getUsableSpace() / (float)fileToWrite.getTotalSpace() <= 0.2f){
          Notification notification = new DefaultNotification("O espa#o dispon#vel no disco #
 inferior a 20% do total");
        notificationManager.sendNotification( notification );
    }
  }
}
```

Como é possível ver no exemplo, o utilitário *NotificationManager* é usado para enviar notificações em decorrência de eventos ocorridos na sua aplicação. O uso mais comum é notificar avisos ou problemas para que ações sejam tomadas, mas é possível também usar essa técnica para tomar ações preventivas ou informativas - uma notificação que o backup noturno foi feito com sucesso por exemplo.

A interface *Notification* é a base das notificações enviadas e possui apenas um método: *Object getMessage()*. A API de monitoração não força o tipo específico da mensagem e usualmente essa mensagem será uma *String* (como no exemplo acima), mas algumas extensões podem utilizar tipos específicos de mensagem capazes de carregar mais informações.

O *demoiselle-core* disponibiliza por padrão o tipo concreto de notificação *DefaultNotification* - uma implementação direta da interface *Notification* que permite definir a mensagem a ser retornada por *getMessage()*. Além disso o *demoiselle-core* disponibiliza o tipo especial de mensagem *AttributeChangeMessage*, que permite especificar além do texto da mensagem enviada, dados sobre a mudança de valor de um atributo, como o nome do atributo alterado, o valor antigo e o novo.

Notificações contendo mensagens do tipo *AttributeChangeMessage* são automaticamente enviadas por padrão pelo framework quando um agente de monitoração altera o valor de uma propriedade anotada com *@ManagedProperty*, mas você também pode enviar mensagens desse tipo quando propriedades de sua aplicação são alteradas. Extensões e componentes compatíveis com a API de monitoração do *Demoiselle Framework* (por exemplo, a extensão *demoiselle-jmx*) podem fornecer implementações específicas da interface *Notification* e tipos especializados de mensagem. Consulte a documentação desses componentes para aprender sobre seus usos.

## 18.4. Conectando um cliente de monitoração

O *demoiselle-core* contém as funcionalidades necessárias para marcar aspectos monitoráveis de sua aplicação, mas não conta com nenhum mecanismo para estabelecer uma conexão com um cliente de monitoração. Para isso utiliza-se extensões do framework.

A extensão padrão do framework *Demoiselle* responsável pela tecnologia de monitoração é a *demoiselle-jmx*. Essa extensão utiliza a especificação JMX (JSR 3) e permite registrar as classes marcadas para monitoração como *MBeans*. Uma vez que as classes sejam registradas como *MBeans*, seus atributos e operações expostos para

monitoração podem ser acessados via JMX por um cliente adequado, como o *JConsole* que acompanha por padrão o JDK da Oracle.

### Dica

Para acrescentar a extensão *demoiselle-jmx* em um projeto Maven, adicione a dependência abaixo no arquivo *pom.xml*.

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle</groupId>
  <artifactId>demoiselle-jmx</artifactId>
  <scope>compile</scope>
</dependency>
```

### Dica

A API de monitoração é compatível com o uso de múltiplas extensões simultâneas. Adicione em seu projeto a dependência às extensões desejadas e configure-as individualmente, as classes monitoradas serão então expostas para todas as extensões escolhidas.

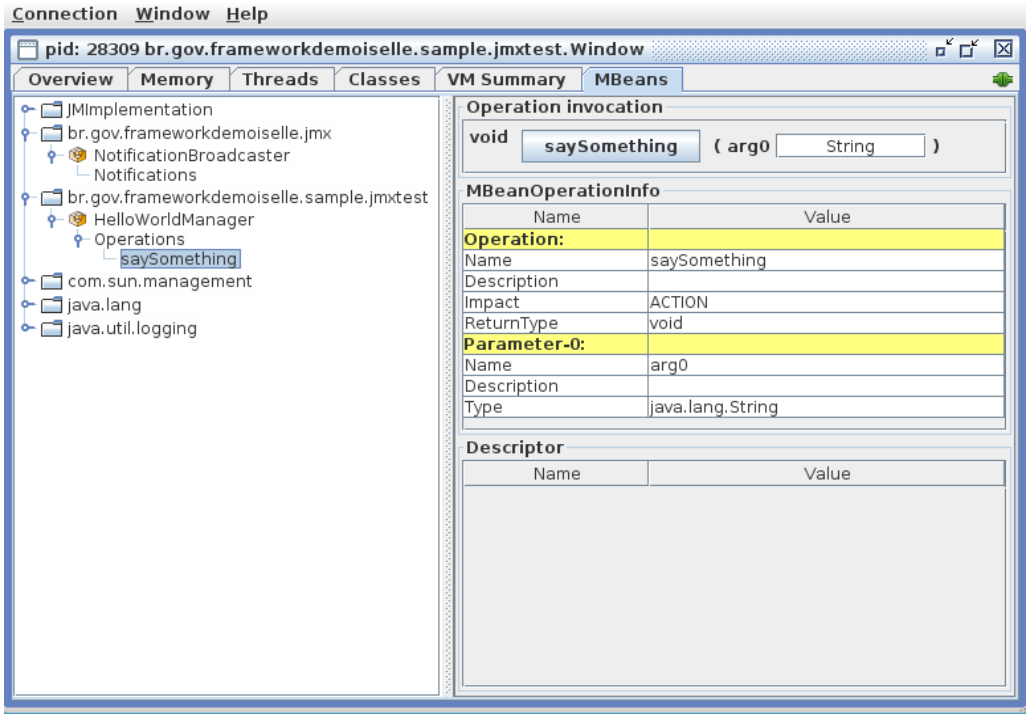
A figura 1 mostra como uma classe monitorada é exibida no *JConsole*.

```
@ManagementController
public class HelloWorldManager {

    @Inject
    private HelloWorld helloWorld;

    @ManagedOperation(type=OperationType.ACTION)
    public void saySomething(String whatToSay){
        helloWorld.say(whatToSay);
    }

}
```




## Figura 18.1. JConsole acessando uma aplicação monitorada

É possível perceber os seguintes elementos nessa imagem:

- Uma classe gerenciada *HelloWorldManager* com uma operação chamada *saySomething*
- Uma classe geradora de notificações *NotificationBroadcaster* responsável por converter as notificações para a API JMX

Através do *JConsole* é possível invocar comandos e acessar atributos em todas as classes anotadas com *@ManagementController* em aplicações que usam o *demoiselle-jmx*. Também é possível inscrever-se às notificações enviadas por *NotificationBroadcaster* e receber todas as notificações enviadas pela aplicação.

# Apêndice A. Instalação

## A.1. Pré-requisitos

| Software | Versão | Site |
|---|---|---|
| Java Development Kit (JDK) | 6.0 | http://openjdk.org/ |
| Apache Maven | 2.2 | http://maven.apache.org/ |
| Eclipse IDE | 3.6 | http://www.eclipse.org/ |
| m2eclipse plugin | 0.10 | http://m2eclipse.sonatype.org/ |
| JBoss Application Server | 6.0 | http://www.jboss.org/ |

## A.2. Demoiselle Infra

Para auxiliar no preparo do ambiente integrado de desenvolvimento utilizado na presente documentação, recomenda-se a utilização dos pacotes de software fornecidos pelo projeto *Demoiselle Infra* [http://demoiselle.sourceforge.net/infra/].

**Nota**

Atualmente são disponibilizados pacotes exclusivamente para a plataforma *GNU/Linux* e distribuições baseadas no *Debian*, tal como *Ubuntu*.

# Apêndice B. Atributos do demoiselle.properties

Em um projeto com o *Demoiselle Framework*, algumas propriedades e configurações do *Framework* podem ser ajustadas no arquivo `demoiselle.properties`. A seguir listamos as propriedades e configurações do *Demoiselle Framework* que o usuário pode modificar, acompanhados de alguns exemplos ilustrativos.

### Tabela B.1. Configurações do Core

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.security.enabled | Habilita o mecanismo de segurança. | true |
| frameworkdemoiselle.security. authenticator.class | Define a classe que implementa a estratégia de autenticação. | |
| frameworkdemoiselle.security.authorizer. class | Define a classe que implementa a estratégia de autorização. | |
| frameworkdemoiselle.transaction.class | Define a classe que implementa a estratégia de controle transacional. | |
| frameworkdemoiselle.pagination.page.size | Define o tamanho da página padrão do mecanismo de paginação. | 10 |

### Tabela B.2. Configurações da extensão JSF

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.security.login.page | Define a página de login da aplicação. | "/login" |
| frameworkdemoiselle.security.redirect. after.login | Define a tela para qual o usuário será redirecionado após o processo de *login* bem sucedido. | "/index" |
| frameworkdemoiselle.security.redirect. after.logout | Define a tela para qual o usuário será redirecionado após o processo de *logout* bem sucedido. | "/login" |
| frameworkdemoiselle.security.redirect. enabled | Habilita os redirecionamentos relacionados aos processos de login e logout. | true |
| frameworkdemoiselle.exception. application.handle | Habilita o tratamento automático das exceções da aplicação anotadas com @ApplicationException. | true |
| frameworkdemoiselle.exception.default. redirect.page | Define o redirecionamento das exceções da aplicação anotadas com @ApplicationException ocorridas durante a fase de renderização da página (PhaseId.RENDER_RESPONSE). | "/application_error" |
| frameworkdemoiselle.pagination.max. page.links | Configura a quantidade de links que será exibido em uma página. | 5 |

## Tabela B.3. Configurações da extensão JDBC

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.persistence.jndi.name | Define o nome JNDI onde o DataSource está disponível. | |
| frameworkdemoiselle.persistence.driver.class | Define a classe que implementa o Driver de conexão com a base de dados. | |
| frameworkdemoiselle.persistence.url | Define a URL de conexão com a base de dados. | |
| frameworkdemoiselle.persistence.username | Define o username para estabelecer a conexão com a base de dados. | |
| frameworkdemoiselle.persistence.password | Define o password para estabelecer a conexão com a base de dados. | |
| frameworkdemoiselle.persistence.default.datasource.name | Define a configuração de banco de dados padrão para aplicações que possuem mais de um datasource configurado. | |

## Tabela B.4. Configurações da extensão JPA

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.persistence.default.unit.name | Define o nome da unidade de persistência padrão (configurada em `persistence.xml`) que será injetada caso a anotação `@Name` não seja usada. Não é necessário se apenas uma unidade de persistência for configurada. | |
| frameworkdemoiselle.persistence.entitymanager.scope | Permite determinar o escopo de unidades de persistência injetadas. Dentro do escopo determinado, todos os pontos de injeção receberão a mesma instância de `EntityManager`.<br><br>Os valores possíveis são: request, session, view, conversation, application, noscope | request |

## Tabela B.5. Configurações da extensão JMX

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.management.mbean.domain | Define o domínio padrão onde classes anotadas com *@ManagementController* serão registradas no MBeanServer.<br><br>Na especificação JMX, um MBean é registrado no MBeanServer com um nome no formato *domain:name=MBeanName* (ex: *br.gov.frameworkdemoiselle:name=NotificationBroadcaster*). Esse parâmetro controla a porção *domain* desse formato. | O pacote da classe anotada com *@ManagementController* |

| Propriedade | Descrição | Valor padrão |
|---|---|---|
| frameworkdemoiselle.management.notification.domain | O mesmo que *frameworkdemoiselle.management.mbean.domain*, mas apenas para o domínio do MBean **br.gov.frameworkdemoiselle.internal.NotificationBroadcaster**. Esse MBean é automaticamente registrado para receber notificações enviadas usando a classe **br.gov.frameworkdemoiselle.management.NotificationManager** | br.gov.frameworkdemoiselle.jmx |
| frameworkdemoiselle.management.notification.name | O nome usado para registrar a classe **br.gov.frameworkdemoiselle.internal.NotificationBroadcaster** como MBean. | NotificationBroadcaster |