

Demoiselle Components Guide

Demoiselle Components

Marlon Carvalho

`<marlon.carvalho@gmail.com>`

Rodrigo Hjort

`<rodrigo.hjort@gmail.com>`

Sobre o Demoiselle Components	vii
I. Demoiselle Mail	1
1. Configuração do Mail	3
1.1. Configuração do Maven	3
1.2. Sessão	3
1.2.1. Configuração Manual	3
1.2.2. Sessão Provida	4
2. Utilização do Mail	5
2.1. Primeiro Exemplo	5
2.2. Definindo a Importância	6
2.3. Incluindo Anexos	6
2.4. Enviando E-mails em HTML	7
2.5. Enviando Cópias BCC e CC	8
2.6. Confirmações de Entrega e Leitura	8
II. Demoiselle Report	11
3. Configuração do Report	13
4. Utilização do Report	15
4.1. Iniciando	15
4.2. Injetando o Relatório	15
4.3. Exibindo Relatórios em JSF	16
4.4. Exibindo Relatórios com o JasperView	17
4.5. Acessando os objetos do JasperReports	17
4.5.1. Obtendo o objeto: JasperReport	17
4.5.2. Obtendo o objeto: JasperPrint	17
III. Demoiselle Validation	19
5. Configuração do Validation	21
6. Utilização do Validation	23
6.1. Implementação da Especificação	23
6.2. Aplicando Validações a Beans	23
6.3. Checando Restrições	23
6.4. Validações Disponibilizadas	24
IV. Demoiselle Authorization	25
7. Configuração do Authorization	27
7.1. Configuração do Maven	27
8. Arquitetura e Utilização do Authorization	29
8.1. Arquitetura	29
8.2. Utilização	29
V. Demoiselle Vaadin	31
9. Vaadin em um Projeto	33
9.1. Criando um Projeto com Vaadin	33
9.2. Estrutura do Projeto	34
9.3. Model View Presenter	34
10. Anotações do Vaadin	35
10.1. @TextField	35
10.2. @CheckBox	36
10.3. @ComboBox	36
10.4. @CpfField	36
10.5. @DateField	37
10.6. @FileField	37
10.7. @FormattedField	37
10.8. @OptionGroup	38
10.9. @PasswordField	38
10.10. @PhoneField	38

10.11. @RichText	39
10.12. @TextArea	39
11. Tópicos Avançados do Vaadin	41
11.1. Eventos	41
11.2. Criando Classes de View	41
11.3. Criando Classes de Presenter	42
11.4. Novos Componentes	42
11.5. Paginação	42
VI. Demoiselle Monitoring	45
12. Introdução ao Monitoring	47
12.1. Modalidades de monitoração: polling e trapping	47
12.2. Sobre o componente Demoiselle Monitoring	48
13. Configuração do Monitoring	49
13.1. Instalação do componente	49
13.2. Arquivos de configuração	49
14. Monitorando via JMX	51
14.1. Criando e exportando um MBean	51
14.2. Especificando um nome para o MBean	52
14.3. Atributos de leitura e escrita	53
15. Monitorando via SNMP (polling)	55
15.1. Configuração do Agente SNMP	55
15.2. Criação do MBean e da árvore MIB	56
15.3. Consumindo os recursos via cliente SNMP	56
15.4. Definindo o tipo de dados SNMP	57
15.5. Exportando um objeto para leitura e escrita via SNMP	59
15.6. Uma árvore MIB mais complexa	60
15.7. Definição de segurança (SNMPv3)	62
16. Monitorando via SNMP (trapping)	67
16.1. Configuração para os trappers SNMP	67
16.2. Criação e utilização do Trapper SNMP	67
16.3. Customizando OIDs nas traps SNMP	69
16.4. Definindo um código específico para a trap SNMP	70
16.5. Enviando traps SNMP com múltiplas variáveis	70
16.6. Especificando os tipos SNMP das variáveis	71
16.7. Usando o trapper SNMP simples	73
17. Monitorando via Zabbix (polling)	75
17.1. Configuração do Agente Zabbix	75
17.2. Criação do MBean para o Zabbix	75
17.3. Consultando os indicadores do Agente Zabbix	76
17.4. Uma monitoração mais complexa	77
17.5. Configurando a monitoração polling no servidor Zabbix	79
18. Monitorando via Zabbix (trapping)	81
18.1. Configuração para os trappers Zabbix	81
18.2. Criação e utilização do Trapper Zabbix	81
18.3. Definindo o host e a chave do indicador	83
18.4. Usando parâmetros curingas nas chaves	84
18.5. Consultando MBeans e usando a anotação @JMXQuery	85
18.6. Configurando a monitoração trapping no servidor Zabbix	86
18.7. Usando o trapper Zabbix simples	87
19. Monitorando com múltiplas abordagens	89
19.1. Um trapper multiuso: SNMP e Zabbix	89
20. Fazendo uso dos Checkers	93
20.1. Criando e utilizando um Checker	93

20.2. Injeção de Trappers em um Checker	94
A. Conceitos de Monitoração	95
A.1. NMS	95
A.2. SNMP	95
A.3. JMX	96

Sobre o Demoiselle Components

O *Demoiselle Components Guide* agrega em um único lugar toda a documentação referente a todos os componentes disponibilizados e que são compatíveis com a versão mais recente do *Demoiselle Framework*.

Parte I. Demoiselle Mail

Para o envio e recebimento de e-mails em aplicativos Java, a solução mais natural é usar a API [JavaMail](http://www.oracle.com/technetwork/java/javamail/index.html) [http://www.oracle.com/technetwork/java/javamail/index.html]. Ela provê um framework para trabalhar com e-mails e que é independente de protocolo e plataforma, facilitando bastante esta tarefa. Para ter esta funcionalidade em sua aplicação, basta ter um simples arquivo no formato JAR no seu `classpath` e que pode ser usado tanto em projetos Java SE como em Java EE.

Apesar de o JavaMail ser uma excelente API, ainda são necessárias algumas configurações extras e que necessitam de muita codificação por parte do programador. Existem outras tarefas que não são muito triviais de serem realizadas usando somente a API do JavaMail, como anexar arquivos, definir a prioridade do e-mail, adicionar pedido de retorno de leitura entre outros.

O componente *Demoiselle Mail* vem para preencher este vazio deixado pelo *JavaMail*, tornando estas tarefas extremamente fáceis. O componente fornece uma única interface para o programador interagir e apenas um arquivo de configuração. Totalmente baseada na ideia de DSL (Domain Specific Language). Definir a importância de um e-mail é tão simples quanto chamar um método `mail.importance().high()`. Neste manual, veremos em todos os detalhes este componente e como ele pode facilitar sensivelmente o envio de e-mail em suas aplicações.

Configuração do Mail

Vamos iniciar a apresentação do componente demonstrando os passos que são necessários para configurá-lo. Como será visto, são poucos e não tomarão muito tempo ou esforço.

1.1. Configuração do Maven

Primeiro, precisamos informar no seu projeto que há uma dependência para um componente do Demoiselle. A configuração é bem simples, basta adicionar uma sessão `dependency` igual ao da listagem abaixo.

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle.component</groupId>
  <artifactId>demoiselle-mail</artifactId>
  <version>2.0.0</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
```

Fique atento apenas para a versão do componente. No momento em que escrevíamos esta documentação, o Demoiselle Mail encontrava-se na versão 2.0.0. Verifique constantemente nossos repositórios e fique sempre atento a nossos informes sobre o lançamento de novas versões!

1.2. Sessão

Para realizar o envio de e-mails é necessário ter uma instância da classe `javax.mail.Session`. Ela contém todas as informações necessárias para que a API do JavaMail possa realizar o envio e o recebimento de e-mails. Como esta classe será instanciada, depende da estratégia que você queira adotar. Esta sessão pode ser configurada manualmente ou obtida através de um servidor, usando JNDI. O Demoiselle lhe permite as duas opções. Na primeira, você colocará todas as configurações necessárias para acessar o servidor de e-mail no arquivo `demoiselle.properties` e a conexão será feita diretamente. Na segunda opção, pode usar o mesmo arquivo apenas para informar se o servidor que provê esta instância é um Tomcat ou JBoss. Neste último caso, as configurações de acesso ao servidor de e-mail são de responsabilidade do servidor.

1.2.1. Configuração Manual

Este tipo de configuração requer que você edite o arquivo `demoiselle.properties` e defina as informações necessárias para se conectar em um servidor de e-mail. As configurações necessárias são basicamente: endereço do servidor de e-mail, usuário, senha do usuário e tipo de segurança. Caso ainda não exista, crie o arquivo `demoiselle.properties` na pasta `/src/main/resources` de seu projeto e selecione-o para edição. As opções que podem ser usadas, são:

Tabela 1.1. Configurações de e-mail no `demoiselle.properties`

Opção	Tipo de Dados	Descrição
<code>frameworkdemoiselle.mail.user</code>	Alfanumérico	O nome do usuário que será usado para realizar a conexão no servidor.
<code>frameworkdemoiselle.mail.password</code>	Alfanumérico	A senha do usuário para conexão ao servidor.

Opção	Tipo de Dados	Descrição
frameworkdemoiselle.mail.serverHost	Alfanumérico	Endereço do servidor de e-mail. O tipo mais comum é um servidor de SMTP.
frameworkdemoiselle.mail.serverPort	Numérico	A porta que deve ser usada para conectar no servidor.
frameworkdemoiselle.mail.enable.ssl	true false	Informa se a conexão com o servidor deve ser uma conexão segura usando SSL.
frameworkdemoiselle.mail.auth	true false	Informa se é necessária autenticação para se conectar no servidor.
frameworkdemoiselle.mail.type	local provided	Informar se a conexão deve ser configurada manualmente ou obtida de um servidor.
frameworkdemoiselle.mail.enable.tls	true false	Informa se a conexão com o servidor deve ser uma conexão segura usando STARTTLS.
frameworkdemoiselle.mail.require.tls	true false	Informa se a conexão com o servidor PRECISA ser uma conexão segura usando TLS.

Com estas configurações, você já está apto para realizar o envio de e-mails.

1.2.2. Sessão Provida

Quando sua aplicação está sendo executada dentro de um Tomcat, JBoss ou outro servidor qualquer, você pode obter uma sessão, já contendo todas as configurações, do próprio servidor. Neste caso, estas configurações já são providas pelo servidor e basta para você obter a sessão dele. Primeiro, edite o arquivo `demoiselle.properties` e adicione a opção `frameworkdemoiselle.mail.type=provided`. Depois, você deve colocar também a opção `frameworkdemoiselle.mail.lookup_class`. O valor vai depender de qual servidor você está usando.

O componente de E-mail do Demoiselle já lhe provê duas opções, sendo elas: `br.gov.frameworkdemoiselle.mail.internal.implementation.TomcatLookup` e `br.gov.frameworkdemoiselle.mail.internal.implementation.JBossLookup`. Feito isto, não é necessária mais nenhuma configuração.

Utilização do Mail

Vamos ver agora como usar o componente para realizar o envio de e-mails a partir de sua aplicação. Toda a API é baseada em uma DSL simples e fácil de usar. Veremos, neste capítulo, todas as opções disponíveis na API.



Dica

A sigla DSL significa Domain Specific Language. São linguagens criadas para um propósito específico, visando a solução de problemas com escopo bem definido e fechado.

2.1. Primeiro Exemplo

Uma vez que temos todas as configurações feitas, já podemos iniciar o envio de e-mail em nossa aplicação. Para isto, você precisa ter uma instância da interface *Mail* usando o mecanismo de injeção disponível no Demoiselle. Neste primeiro exemplo, vamos fazer um envio simples de e-mail, contendo apenas um assunto, corpo, de e para.

```
public class Teste {  
  
    @Inject  
    private Mail mail;  
  
    public void enviar() {  
        mail  
            .to("from@server.com")  
            .from("to@server.com")  
            .body().text("Meu Primeiro E-mail")  
            .subject("Assunto")  
            .send();  
    }  
}
```

A API é bem simples e auto-explicativa. A interface *Mail* contém métodos que denotam ações comuns no envio de e-mail, como `bcc()`, `cc()`, `from()`, `to()`, `attach()` e etc. Caso você use uma IDE que possui o recurso de autocomplete, seu esforço será ainda menor. Abaixo está a interface *Mail* e os métodos disponíveis.

```
public interface Mail {  
    public Mail from(String address);  
    public Mail from(String address, String name);  
    public Mail from(InternetAddress address);  
    public Mail replyTo(String address);  
    public Mail replyTo(String address, String name);  
    public Mail replyTo(InternetAddress emailAddress);  
    public Mail to(String address);  
    public Mail to(String address, String name);  
    public Mail to(InternetAddress address);  
    public Mail cc(String address);  
    public Mail cc(String address, String name);  
    public Mail cc(InternetAddress address);  
    public Mail bcc(String address);  
    public Mail bcc(String address, String name);  
    public Mail bcc(InternetAddress address);  
}
```

```
public Importance importance();  
public Mail deliveryReceipt(String address);  
public Mail readReceipt(String address);  
public Mail subject(String value);  
public Body body();  
public BaseMessage send();  
public Attach attach();  
}
```

2.2. Definindo a Importância

Caso haja necessidade de definir o nível de urgência, ou importância, do seu e-mail, basta usar conforme o exemplo abaixo.

```
public class Teste {  
  
    @Inject  
    private Mail mail;  
  
    public void enviar() {  
        mail  
            .to("from@server.com")  
            .from("to@server.com")  
            .body().text("Meu Primeiro E-mail")  
            .subject("Assunto")  
            .importance().high()  
            .send();  
    }  
}
```

Neste caso, você definiu a importância como alta. Também existem as opções NORMAL e LOW.



Dica

A importância de um e-mail depende bastante do cliente de e-mail que o destinatário do e-mail está usando. Alguns ignoram esta informação e nada exibem. Outros exibem ícones para mostrar que o remetente pede urgência em sua leitura. Infelizmente, nem todos clientes de e-mail fornecem meios para ver de forma fácil o nível de urgência dos e-mails que recebe.

2.3. Incluindo Anexos

A API também provê a possibilidade de se incluir arquivos em anexo ao e-mail. O anexo pode ser obtido da máquina local ou de uma URL. No caso de arquivos obtidos da máquina local, é possível passá-lo através de uma instância de File ou apenas informando o nome do arquivo. Neste primeiro exemplo, vamos anexar um arquivo de uma URL.

```
public class Teste {  
  
    @Inject  
    private Mail mail;
```

```
public void enviar() {  
    mail  
        .to("from@server.com")  
        .from("to@server.com")  
        .body().text("Meu Primeiro E-mail")  
        .subject("Assunto")  
        .attach().url("http://www.frameworkdemoiselle.gov.br/logo.jpg", "logo.jpg").inline()  
        .send();  
}
```

Estamos incluindo um arquivo de imagem e pedindo para que este arquivo seja exibido dentro da própria mensagem. Para isto, usamos a opção `inline()`. No próximo exemplo, vamos anexar um arquivo que está na própria máquina.

```
public class Teste {  
  
    @Inject  
    private Mail mail;  
  
    public void enviar() {  
        mail  
            .to("from@server.com")  
            .from("to@server.com")  
            .body().text("Meu Primeiro E-mail")  
            .subject("Assunto")  
            .importance().high()  
            .attach().file("documento.doc").attachment()  
            .send();  
    }  
}
```

2.4. Enviando E-mails em HTML

O envio de e-mails com conteúdo HTML é bem simples. Vamos utilizar os exemplos anteriores e apenas modificar uma chamada simples de método.

```
public class Teste {  
  
    @Inject  
    private Mail mail;  
  
    public void enviar() {  
        mail  
            .to("from@server.com")  
            .from("to@server.com")  
            .body().html("<strong>Meu</strong> Primeiro E-mail")  
            .subject("Assunto")  
            .importance().high()  
            .attach().file("documento.doc").attachment()  
            .send();  
    }  
}
```

```
}
```

A mudança está na linha onde tem `body()`. Agora, fizemos uma chamada para o método `html()` e não mais para `text()`. Apenas com isto, você já pode passar código HTML no lugar de apenas texto.

2.5. Enviando Cópias BCC e CC

O envio de cópias Carbon e Blind Carbon é tão simples quanto definir o remetente e o destinatário do e-mail. Veja o exemplo abaixo.

```
public class Teste {

    @Inject
    private Mail mail;

    public void enviar() {
        mail
            .to("from@server.com")
            .from("to@server.com")
            .bcc("bcc1@server.com")
            .bcc("bcc2@server.com")
            .cc("ccl@server.com")
            .cc("cc2@server.com.br")
            .body().html("<strong>Meu</strong> Primeiro E-mail")
            .subject("Assunto")
            .importance().high()
            .attach().file("documento.doc").attachment()
            .send();
    }
}
```

Observe que para enviar para mais de um e-mail, basta repetir a chamada ao método com um e-mail diferente. O mesmo vale para todas as demais opções, como `to`, `from`, `attach`, `readReceipt`, `deliveryReceipt` e `replyTo`.

2.6. Confirmações de Entrega e Leitura

Em alguns casos, você pode também incluir no seu e-mail, cabeçalhos que peçam ao destinatário do e-mail para confirmar o recebimento ou a leitura do e-mail. Com o componente do Demoiselle, esta ação é bastante simples. Para isto, você deve usar os métodos `deliveryReceipt` e `readReceipt`, passando como parâmetro o e-mail que deve receber a confirmação. Veja o exemplo abaixo.

```
public class Teste {

    @Inject
    private Mail mail;

    public void enviar() {
        mail
            .to("from@server.com")
            .from("to@server.com")
            .deliveryReceipt("from@server.com")
            .readReceipt("from@server.com")
    }
}
```



```
.body().html("<strong>Meu</strong> Primeiro E-mail")  
.subject("Assunto")  
.importance().high()  
.send();  
}  
}
```

Em ambos os casos, utilizamos o mesmo endereço de e-mail usado no FROM, o que é a prática mais comum. O método `deliveryReceipt` especifica para qual e-mail será enviada a confirmação de entrega do e-mail. Já o método `readReceipt` especifica para qual e-mail será enviada a confirmação de entrega do e-mail.



Dica

Igual a como acontece com a opção para incluir a importância da mensagem, a confirmação de leitura e recebimento dependem de configurações no cliente de e-mail do destinatário. O usuário pode desabilitar estas opções e o remetente nunca ficará sabendo se o e-mail foi realmente entregue ou lido.

Parte II. Demoiselle Report

A geração de relatórios é uma tarefa bastante comum em boa parte dos sistemas comerciais. Visando sempre facilitar a vida do desenvolvedor, diversas ferramentas foram surgindo, sempre tendo em mente criar mecanismos que desonerassem os responsáveis por este trabalho. Das diversas ferramentas existentes, o Crystal Reports talvez seja a mais conhecida e usada na plataforma Windows. Com a chegada da linguagem Java, contudo, foi necessário criar alternativas, uma vez que as ferramentas da época não eram compatíveis com esta linguagem. A primeira opção que surgiu no mercado foi o *Jasper Reports* [<http://jasperforge.org/projects/jasperreports>], que forma uma ótima dupla com o *iReport* [<http://jasperforge.org/project/ireport>]. O primeiro trata da geração do relatório em si, enquanto o segundo é um editor visual que facilita sobremaneira o trabalho de geração do layout.

Embora seja o mais conhecido dos programadores Java, o JasperReports não é a única opção disponível no mercado. Entre algumas opções livres e outras não-livres, temos atualmente o *BIRT* [<http://www.eclipse.org/birt/phoenix/>], da Eclipse, que tem ganhado cada dia mais adeptos. O BIRT apresenta características diferenciadas do JasperReports, mas o intuito é o mesmo: facilitar a criação de relatórios.

Tendo em vista que já existem ótimas ferramentas disponíveis e reconhecendo não haver necessidade de propor mais uma, o objetivo do componente de relatórios do Demoiselle é propor facilidades e tentar tornar o mais transparente possível, para o usuário do framework, qual a biblioteca que está sendo de fato utilizada. O componente já conta com o suporte para o *Jasper Reports* e o objetivo é torná-lo compatível com outras bibliotecas de geração de relatórios que venham a tornar-se padrão de mercado.

Nesta seção, vamos estudar como funciona e como utilizar este componente em seu projeto, passando por alguns conceitos importantes.

Configuração do Report

Vamos iniciar a apresentação do componente demonstrando os passos que são necessários para configurá-lo e torná-lo disponível para o seu projeto. Como será visto, são poucos e não tomarão muito tempo ou esforço. Inicialmente, precisamos informar que há uma dependência para um componente do Demoiselle. A configuração é bem simples, bastando adicionar uma sessão `dependency` igual ao da listagem abaixo.



Dica

O componente já traz consigo a dependência para o JasperReports, pois, até o momento, é a única ferramenta com suporte.

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle.component</groupId>
  <artifactId>demoiselle-report</artifactId>
  <version>2.0.0</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
```



Importante

Fique atento apenas para a versão do componente. No momento em que escrevíamos esta documentação, o *Demoiselle Report* encontrava-se na versão 2.0.0. Verifique constantemente nossos repositórios e fique sempre atento a nossos informes sobre o lançamento de novas versões!

Utilização do Report

Nesta seção, vamos abordar a utilização do componente de relatórios. Cabe destacar que o componente não está preso a uma tecnologia de visão específica. Ele é independente, podendo ser utilizado em uma aplicação JSF, Swing ou Vaadin. Ele limita-se a tratar da geração do relatório e não presume qual a tecnologia de visão será utilizada.

4.1. Iniciando

Primeiro, precisamos criar nosso relatório usando o Jasper. Para alcançar este objetivo, você pode usar a ferramenta iReport ou, caso tenha um bom conhecimento sobre o Jasper, criar o arquivo JRXML diretamente, escrevendo código XML. O componente trata tanto arquivos JRXML, compilando-os em *runtime*, ou arquivos já compilados, no formato *.jasper*. Seja qual for o formato ou método que você utilize para gerar os arquivos, para o componente do Demoiselle é importante que o arquivo esteja disponível no classpath da sua aplicação. Uma sugestão é colocar os relatórios na pasta `/src/main/resources/reports`.

Estrutura de diretórios



Atenção

Ao utilizar um arquivo **.jrxml* obrigará o componente a realizar a compilação do mesmo em tempo de execução. Para obter uma melhor performance use sempre o arquivo pré-compilado **.jasper*

4.2. Injetando o Relatório

Uma vez que temos o arquivo do relatório em nosso classpath, agora precisamos injetar uma instância de `br.gov.frameworkdemoiselle.report.Report` em nosso objeto. A interface `Report` prevê poucos métodos, sendo o mais importante, o método `export()`. Este método permite que você gere seu relatório com os dados informados, retornando um `Array` de bytes.

```
public class RelatorioExemplo {  
  
    @Inject  
    @Path("reports/Bookmarks.jrxml")  
    private Report relatorio;  
  
}
```

No exemplo acima, estamos injetando uma instância de `Report` e informando, através da anotação `@Path`, o caminho onde se encontra o nosso arquivo de relatório. Agora que já temos o relatório, podemos solicitar que ele seja gerado, conforme ilustra o exemplo a seguir.



Dica

A omissão da anotação `@Path` vai considerar o diretório `reports/` e o nome do arquivo será o nome da propriedade, neste caso: `relatorio.jasper`

```
public void getReport() {
    Map<String, Object> param = new HashMap<String, Object>();
    try {
        byte[] buffer = this.relatorio.export(getResultList(), param, Type.PDF);
    } catch (JRException e) {
        // Trata a exceção
    }
}
```

Neste exemplo, estamos passando os parâmetros contido no `HashMap param` e como *DataSource*, o retorno do método `getResultList()` que, de fato, retorna uma instância de `java.util.List`. Por último, estamos informando que queremos o relatório no formato PDF, da Adobe. Cabe agora ter uma forma para exibir o conteúdo deste relatório. Entretanto, o componente não entra neste mérito, cabendo a ele apenas gerar. De fato, a forma como o relatório será exibido é de responsabilidade da camada de visão. Em um sistema desenvolvido em JSF, esta apresentação ocorrerá através de um navegador. Caso seu sistema seja em linha de comando, ou Swing, precisará de outra forma para exibir.



Dica

A extensão JSF já provê mecanismos para a exibição de relatórios. Trataremos deste assunto na próxima seção.

4.3. Exibindo Relatórios em JSF

A extensão JSF já provê, desde as versões mais recentes, um utilitário que permite a exibição de arquivos, de forma genérica, através de um navegador. Aqui apresentaremos como você pode utilizá-lo para exibir o relatório que geramos na seção anterior. Reescreveremos o método `getReport()` para que ele faça este trabalho.

```
@Inject
private FileRenderer renderer;

public String showReport() {
    Map<String, Object> param = new HashMap<String, Object>();
    try {
        byte[] buffer = this.relatorio.export(getResultList(), param, Type.PDF);
        this.renderer.render(buffer, FileRenderer.ContentType.PDF, "relatorio.pdf");
    } catch (JRException e) {
        Faces.addMessage(e);
    }
    return getNextView();
}
```

Vamos considerar que o método acima está inserido em uma classe de *ManagedBean*, própria do Java Server Faces, e que este método será chamado assim que for realizado um clique em um botão na tela. Neste caso, teremos uma instância da classe `FileRenderer`, que é responsável pela exibição de arquivos, independente de onde ele veio. Neste exemplo, estamos apenas para que o conjunto de bytes que obtivemos do relatório seja impresso como um PDF e que para o navegador, este arquivo apareça com o nome *relatorio.pdf*.

4.4. Exibindo Relatórios com o JasperView

O *JasperReports* já disponibiliza um mecanismo para a visualização dos relatórios chamada *JasperView*. Para utilizá-la é preciso passar o objeto do relatório já preenchido, que é o *JasperPrint*. A interface *Report* possibilita acesso aos objetos do *JasperReports* através do método `getReportObject()` como pode ser visualizado no trecho de código abaixo.

```
public class ExibirRelatorio {

    @Inject
    private Report relatorio;

    @Inject
    private BookmarkBC bc;

    public void exibirNoJasperView() {
        relatorio.prepare(bc.findAll(), null);
        JasperViewer.viewReport((JasperPrint) relatorio.getReportObject());
    }

}
```

Perceba que para obter o *JasperPrint* é preciso primeiro invocar o método `prepare` do relatório para que o mesmo seja preenchido com os valores.

4.5. Acessando os objetos do JasperReports

Em alguns casos talvez seja necessário manipular diretamente os objetos do Jasper como o *JasperReport* e o *JasperPrint*. Neste caso devemos utilizar os métodos de acesso aos objetos internos e realizar o devido cast.

4.5.1. Obtendo o objeto: JasperReport

O objeto *JasperReport* pode ser acessado a qualquer momento através do método `getSource()` fazendo o cast diretamente.

```
JasperReport jasperReport = (JasperReport)relatorio.getSource();
```

4.5.2. Obtendo o objeto: JasperPrint

O objeto *JasperPrint* pode ser acessado após a preparação do relatório com seus devidos dados. Para isso é preciso utilizar o método `prepare` antes de acessar o *JasperPrint* pelo método `getReportObject()`

```
relatorio.prepare(bc.findAll(), null);
JasperPrint jasperPrint = (JasperPrint) relatorio.getReportObject();
```

Parte III. Demoiselle Validation

A tarefa de validação de dados sempre foi uma necessidade da grande maioria das aplicações. Contudo, não havia uma forma comum para realizar esta validação. Normalmente, esta regra era pulverizada em diversas camadas da aplicação. Ou, ainda pior, repetida tanto na visão como nas regras de negócio. Para solucionar este problema, foi lançada a JSR 303, chamada de Beans Validation.

Esta especificação provê uma forma padrão para a criação de validações através do uso de anotações que podem ser usadas nos beans da aplicação. Atualmente, as especificações JPA 2.0 e JSF 2.0 já se integram automaticamente à JSR 303, realizando automaticamente a validação e lançando a exceção correspondente caso haja uma violação. O componente Demoiselle Validation visa prover validações mais específicas, comuns em aplicações diversas, principalmente aquelas direcionadas para o público brasileiro, como CPF, CNPJ, CEP, Título de Eleitor, Inscrição Estadual e etc.

Configuração do Validation

Vamos iniciar a apresentação do componente demonstrando os passos que são necessários para que sua aplicação possa usá-lo. Como será visto, são poucos e não tomarão muito tempo ou esforço.

Primeiro, precisamos informar no seu projeto que há uma dependência para um componente do Demoiselle. A configuração é bem simples, basta adicionar uma sessão `dependency` igual ao da listagem abaixo.

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle.component</groupId>
  <artifactId>demoiselle-validation</artifactId>
  <version>2.0.0</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
```



Importante

Fique atento apenas para a versão do componente. No momento em que escrevíamos esta documentação, o *Demoiselle Validation* encontrava-se na versão 2.0.0. Verifique constantemente nossos repositórios e fique sempre atento a nossos informes sobre o lançamento de novas versões!

Utilização do Validation

Analisaremos agora como aplicar uma validação ao seu bean e quais são as validações que o componente fornece para sua aplicação.

6.1. Implementação da Especificação

Inicialmente, é necessário ter em seu *classpath* uma referência a alguma implementação da especificação. Atualmente, a mais utilizada é a Hibernate Validator. Contudo, existem outras e cabe analisar qual a melhor para seu projeto, pois cada implementação também fornece validações extras, além daquelas já descritas na própria especificação. Caso você opte pela utilização do Hibernate Validator, basta ter uma dependência no seu POM, conforme o exemplo abaixo.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.2.0-Final</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
```

6.2. Aplicando Validações a Beans

Veremos agora como você pode aplicar a um bean de sua aplicação alguma validação. Não há qualquer complicação, bastando anotar os atributos de sua classe com anotações específicas. O exemplo abaixo aplica uma validação que não permite que seja atribuído ao atributo *nome* um valor nulo.

```
public class MeuBean {

    @NotNull
    private String nome;

    // Gets e Sets
}
```

A anotação *@NotNull* já é provida por padrão pela especificação. Isto significa que toda implementação dela deve provê-la.

6.3. Checando Restrições

Conforme discutimos, o JSF 2.0 e o JPA 2.0 já realizam automaticamente a validação para você. Neste caso, quando você pede para um *EntityManager* persistir seu bean, ele não o fará caso haja alguma restrição. Da mesma forma, durante a fase de validação do JSF 2.0 ocorrerá esta verificação, já exibindo o erro para o usuário. Contudo, existem casos onde você queira verificar estas validações de forma manual, diretamente em sua classe de negócio, por exemplo. Para realizar esta tarefa, basta ter o código abaixo.

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
```

```
Validator validator = factory.getValidator();
Set<ConstraintViolation<?>> constraints = validator.validate(object, groups);

for (ConstraintViolation<?> violation:constraints) {
    violation.getMessage();
    violation.getInvalidValue();
    violation.getPropertyPath();
}
```

As primeiras linhas são apenas para obter a implementação da JSR 303 que está disponível no seu *classpath*. Depois, obtemos o validador padrão e solicitamos a validação de um determinado objeto. Teremos, assim, um conjunto de violações de restrições, nas quais você pode iterar e obter informações específicas sobre a violação.

6.4. Validações Disponibilizadas

O componente Demoiselle Validation provê outros tipos de validações, comuns em aplicações para o público brasileiro. As validações são de CPF, CNPJ, Inscrição Estadual, CEP, Título de Eleitor, PIS/PASEP. A utilização de cada uma é bem simples, bastando usar a anotação específica. Veja o exemplo abaixo.

```
public class MeuBean {

    @Cpf
    private String cpf;

    @Cnpj
    private String cnpj;

    @InscricaoEstadual(pattern=PatternInscricaoEstadual.BAHIA)
    private String inscricaoEstadual;

    @Cep
    private String cep;

    @TituloEleitor
    private String tituloEleitor;

    @PisPasep
    private String pis;

    // Gets e Sets
}
```

Das anotações apresentadas no exemplo, cabe uma explicação mais detalhada apenas a referente à Inscrição Estadual, pois cada estado tem seu próprio padrão. Desta forma, você deve informar para qual estado você quer realizar a validação. Esta validação ainda não provê uma forma universal de realizar a validação, de uma maneira na qual você não precisa informar qual o estado.

Parte IV. Demoiselle Authorization

O Guia de Referência do Demoiselle tratou de forma bem detalhada sobre um importante aspecto presente em grande parte dos sistemas: controle de acesso. Demonstrou as interfaces `Authorizer` e `Authenticator` e como implementá-las para que você crie seu próprio mecanismo de autenticação e autorização. Contudo, é bastante conhecida no mundo Java a especificação do JAAS, acrônimo para Java Authentication and Authorization Service, que provê meios para você realizar este mesmo trabalho em ambientes JEE.

O componente Demoiselle Authorization é uma iniciativa que visa permitir a utilização do JAAS, baseado nas interfaces explicitadas no Guia de Referência, na sua aplicação. O objetivo desta documentação é explicar como você pode usar este componente para usufruir do JAAS de forma mais simples. Cabe destacar que não entraremos em detalhes sobre a própria especificação do JAAS, uma vez que existe uma vasta documentação sobre o assunto.

Configuração do Authorization

Vamos iniciar a apresentação do componente demonstrando os passos que são necessários para configurá-lo. Como será visto, são poucos e não tomarão muito tempo ou esforço.

7.1. Configuração do Maven

Primeiro, precisamos informar no seu projeto que há uma dependência para um componente do Demoiselle. A configuração é bem simples, basta adicionar uma sessão `dependency` igual ao da listagem abaixo.

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle.component</groupId>
  <artifactId>demoiselle-authorization</artifactId>
  <version>2.0.0</version>
  <type>jar</type>
  <scope>compile</scope>
</dependency>
```



Importante

Fique atento apenas para a versão do componente. No momento em que escrevíamos esta documentação, o *Demoiselle Authorization* encontrava-se na versão 2.0.0. Verifique constantemente nossos repositórios e fique sempre atento a nossos informes sobre o lançamento de novas versões!

Arquitetura e Utilização do Authorization

A utilização do Demoiselle Authorization não tem qualquer complicação. De fato, a maior parte das tarefas encontra-se na configuração e criação das classes do próprio JAAS. Como não entraremos nestes detalhes na documentação do componente, resta muito pouco a se fazer.

8.1. Arquitetura

Conforme discutimos no Guia de Referência do Demoiselle, temos duas interfaces básicas para determinar como funciona o controle de acesso: `Authorizer` e `Authenticator`. O componente `Authorization` do Demoiselle, como você deve imaginar, implementa estas duas interfaces nas classes `JaasAuthorizer` e `JaasAuthenticator`.

Contudo, o trabalho das instâncias destas classes é bastante simples, pois todo o trabalho é realizado pelo próprio JAAS e que é tratado diretamente pelo container, como o JBoss ou Glassfish. As duas classes estão exemplificadas nas listagens abaixo. Primeiro, a classe `JaasAuthorizer`, que é simples e não implementa o método `hasPermission(resource, operation)` pois não há este mesmo conceito no JAAS. Segundo, temos o método `hasRole(role)`, simplesmente delegando.

```
public class JaasAuthorizer implements Authorizer {

    public boolean hasPermission(String resource, String operation) {
        throw new DemoiselleException(bundle.getString("permission-not-defined-for-jaas",
            RequiredRole.class.getSimpleName()));
    }

    @Override
    public boolean hasRole(String role) {
        HttpServletRequest request = Beans.getReference(HttpServletRequest.class);
        return request.isUserInRole(role);
    }

}
```

Da mesma forma, a classe `JaasAuthenticator` não implementa os métodos `unAuthenticate()` e `authenticate()`, pois isto é realizado diretamente pelo container. O uso destes métodos acarretará no lançamento de uma exceção do tipo `DemoiselleException`.

8.2. Utilização

A utilização do componente `Authorization` não altera a forma como você deveria usar normalmente a funcionalidade de segurança do framework Demoiselle. Você continuará anotando seus métodos que necessitam de controle de acesso com as anotações `@RequiredRole` e `@RequiredPermission`. Entretanto, lembre-se o que discutimos na seção anterior, quando vimos a implementação das interfaces `Authenticator` e `Authorizer`: existem métodos que não são implementados, pois já são feitos diretamente pelo container.

Relembrando, não temos a implementação de `hasPermission(resource, operation)` em `JaasAuthorizer`. Isto significa que não poderemos usar a anotação `@RequiredPermission`, pois acarretará

em uma exceção `DemoiselleException`. Veja abaixo um exemplo de uso e confira mais detalhes de como usar estas anotações na documentação de referência do Demoiselle.

```
public class ClasseExemplo {

    @RequiredRole("simpleRoleName")
    public void requiredRoleWithSingleRole() {
    }

    @RequiredRole({ "firstRole", "secondRole", "thirdRole", "fourthRole", "fifthRole" })
    public void requiredRoleWithArrayOfRoles() {
    }

}
```

Parte V. Demoiselle Vaadin

O *Vaadin* é um framework para construção de interfaces web que aplica ideias inovadoras. Apesar de usar o *GWT* como mecanismo para a exibição de seus componentes, eles tem características que os diferenciam. No *Vaadin*, assim como no *GWT*, a construção da interface ocorre de forma bem semelhante ao modelo já conhecido no *Swing*/*SWT*: através da composição de diversos componentes e usando apenas código Java. Nada mais!

O *Vaadin* não sofre de dois problemas comuns para os desenvolvedores que usam o *GWT*. O primeiro é a grande quantidade de classes de comunicação remota que precisamos criar. O segundo problema é a limitação no uso de bibliotecas externas na criação da interface, pois todo código Java da parte cliente é traduzido para Javascript. Com o *Vaadin*, todos os componentes possuem seu ciclo de vida no servidor e não no browser, o que elimina esta deficiência.

O objetivo deste componente é permitir o uso desta tecnologia com o *Demoiselle*, possibilitando o uso de Injeção de Dependência, além das demais funcionalidades já conhecidas do framework. Também são fornecidas funcionalidades extras que facilitam a criação de aplicações, voltadas a padrões de mercado bastante conhecidos.

Vaadin em um Projeto

9.1. Criando um Projeto com Vaadin

A construção de uma aplicação com Vaadin diferencia-se bastante de uma construída em JSF. Aqui não temos páginas XHTML com tags que definem os componentes de tela. Ao contrário disto, temos apenas código Java. Portanto, não espere encontrar qualquer arquivo com extensão XHTML, JSP ou HTML em um projeto Vaadin.

Tudo é feito com código puramente Java. Isto mesmo. Toda a tela é construída de forma semelhante a como é feito em Swing: criando painéis, janelas e agrupando componentes. Para facilitar o uso do Vaadin com o Demoiselle, o componente sugere uma estrutura padrão em seus arquétipos, muito parecida com a estrutura encontrada em aplicações JSF. Nesta seção, vamos discutir os principais aspectos que você pode se deparar ao tentar criar sua primeira aplicação Vaadin com o Demoiselle.

O primeiro passo na criação de uma aplicação é criar uma classe e colocá-la para estender de `br.gov.frameworkdemoiselle.template.VaadinApplication`. Seu arquivo `web.xml` também deve referenciar esta classe, conforme o exemplo abaixo:

```
<web-app>
  <display-name>Bookmark</display-name>
  <servlet>
    <servlet-name>Bookmark</servlet-name>
    <servlet-class>
      br.gov.frameworkdemoiselle.servlet.VaadinApplicationServlet
    </servlet-class>
    <init-param>
      <param-name>application</param-name>
      <param-value>com.suaempresa.Aplicacao</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>Bookmark</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Relembramos que a melhor forma de ter uma aplicação Vaadin com Demoiselle pronta para o uso, é adotar o nosso arquétipo `demoiselle-vaadin-jpa`. Este arquétipo já lhe fornece todas estas configurações, basta a você escrever o código da aplicação. Segue abaixo um exemplo de classe que estende de `VaadinApplication`.

```
@SessionScoped
public class Application extends VaadinApplication {

    @Inject
    ResourceBundle bundle;

    @Inject
    private MainPresenter mainPresenter;

    private Window mainWindow;

    public void init() {
```

```
mainWindow = new Window(bundle.getString("login.titulo"), mainPresenter.getView());
setMainWindow(mainWindow);
}
}
```

9.2. Estrutura do Projeto

A estrutura de pacotes de um projeto Vaadin não é muito diferente daquela de um projeto com JSF. Contudo, aqui não temos páginas com tags. Ao contrário disto, temos apenas classes Java que definem o layout e o comportamento de suas telas.

Uma vez que toda a tela é construída com código Java, precisamos de um pacote na nossa estrutura para armazenar estas classes. O componente sugere a criação do pacote `com.seuprojeto.ui.view` para isto. Também é sugerida a criação do pacote `com.seuprojeto.ui.presenter`, para o armazenamento das classes controladores da View, conforme o padrão MVP, que discutiremos logo a seguir. Observe que os demais pacotes seguem a estrutura padrão de projetos Demoiselle.

9.3. Model View Presenter

O componente Vaadin não prende o desenvolvedor a um determinado padrão ou modelo de desenvolvimento, entretanto, sugere uma estruturação de seu projeto, visando melhor legibilidade e facilidade nas futuras manutenções. O padrão sugerido pelo componente é o Model View Presenter (MVP). Neste sentido, algumas classes, anotações e mecanismos foram criados visando facilitar a aplicação deste padrão.

Este padrão é bastante semelhante ao já conhecido Model View Controller (MVC), contudo, existem algumas diferenças. O importante é entender que você deve separar suas classes que possuem a lógica de apresentação, daquelas que são responsáveis em tratar a interação do usuário e respostas a estas interações. A primeira é chamada de View, enquanto a segunda é conhecida como Presenter.

Um exemplo rápido da utilização deste padrão, seria uma tela na qual existe apenas um botão que quando clicado exibe uma mensagem em um Label. O botão e o label estariam na classe de View. Quando o usuário clica no botão, o Presenter captura este evento e informa para a View exibir a mensagem para o usuário.

Anotações do Vaadin

Para facilitar a criação de interfaces, o componente fornece algumas anotações que podem ser usadas nas classes de domínio da aplicação. Uma vez que um campo de sua classe é anotado, o componente pode usar esta informação para criar automaticamente um componente correspondente na tela. Suas classes de domínio, caso você tenha criado seu projeto a partir de um arquétipo Demoiselle, estarão no pacote `com.suaempresa.domain`. A principal anotação é a `@Field`. Com ela, você informa ao Demoiselle que este campo refere-se a um componente de tela. Veja um exemplo de utilização abaixo, retirado do arquétipo `demoiselle-vaadin.jpa`.

```
@Entity
public class Bookmark implements Serializable {

    @Column
    @Field(prompt = "{bookmark.prompt.description}", label = "{bookmark.label.description}")
    private String description;

}
```

Neste exemplo, estamos informando para o componente que o campo `description`, da entidade `Bookmark`, deve ser associado a um componente de tela. Os parâmetros `label` e `prompt` desta anotação informam, respectivamente, qual será o label do campo na tela e qual a mensagem descritiva do campo (`prompt`). Por padrão, este campo será exibido como uma caixa de texto comum, pois trata-se de uma `String`. Caso seja necessário personalizar, será necessário usar outras anotações, que discutiremos agora.



Nota

Observe que os valores para `prompt` e `label` estão entre chaves. Isto informa que não se trata de um valor literal, mas uma chave que deve ser usada para obter o valor real em um arquivo de `resource`, como o `messages.properties`.

10.1. @TextField

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de texto comum. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @Column
    @TextField
    @Field(prompt = "{bookmark.prompt.description}", label = "{bookmark.label.description}")
    private String description;

}
```

10.2. @CheckBox

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de seleção, comumente conhecida como CheckBox. É importante que o campo anotado seja do tipo Boolean. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @CheckBox
    @Field(prompt = "{bookmark.prompt.visited}", label = "{bookmark.label.visited}")
    private boolean visited;

}
```

10.3. @ComboBox

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma lista de seleção, comumente conhecida como ComboBox ou SelectList. Esta anotação é muito usada quando o campo se refere a um relacionamento entre entidades. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @ManyToOne
    @ComboBox(fieldLabel = "description")
    @Field(prompt = "{bookmark.prompt.category}", label = "{bookmark.label.category}")
    private Category category;

}
```

O parâmetro `fieldLabel` informa qual atributo da entidade `Category` será usado para exibir como descritivo do componente. Caso este parâmetro não seja usado, você deve sobrescrever o método `toString()`.

10.4. @CpfField

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de texto comum, entretanto, com uma formatação de CPF (Cadastro de Pessoa Física). Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @Column
    @Field(prompt = "CPF", label = "CPF")
    @CpfField
    private String cpf;

}
```

10.5. @DateField

Usando esta anotação, você informa que um atributo da classe deve ser exibido como um campo de data, no qual um calendário será exibido para selecionar um valor. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @Field(prompt = "{bookmark.prompt.data}", label = "{bookmark.label.data}")
    @DateField(format="dd/MM/yyyy")
    private Date data;

}
```

É importante que o atributo seja do tipo `java.util.Date`. O parâmetro `format` informa como a data deve ser exibida, quando selecionada.

10.6. @FileField

Usando esta anotação, você informa que um atributo da classe terá como conteúdo o valor binário de um determinado arquivo que será enviado pelo usuário. Será exibido na tela um campo de Upload de arquivos e seu conteúdo será gravado neste atributo. É importante que o atributo seja um array de bytes. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @Lob
    @Basic(fetch=FetchType.LAZY)
    @FileField
    @Field(prompt = "{clube.prompt.escudo}", label = "{clube.label.escudo}")
    private byte[] escudo;

}
```

10.7. @FormattedField

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de texto comum, entretanto, com uma formatação específica, informada através do parâmetro `format`. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @Column
    @Field(prompt = "Formatado", label = "Formatado")
    @FormattedField(format = "999.999.999-99", straight = true)
    private String formatado;

}
```

10.8. @OptionGroup

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma lista de botões de seleção única, comumente conhecida como `RadioButton`. Esta anotação é muito usada quando o campo se refere a um relacionamento entre entidades. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @ManyToOne
    @OptionGroup(fieldLabel = "description")
    @Field(prompt = "{bookmark.prompt.category}", label = "{bookmark.label.category}")
    private Category category;

}
```

O parâmetro `fieldLabel` informa qual atributo da entidade `Category` será usado para exibir como descritivo do componente. Caso este parâmetro não seja usado, você deve sobrescrever o método `toString()`.

10.9. @PasswordField

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de texto para que sejam informados valores secretos, como senhas. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @Column
    @Field(prompt = "Senha", label = "Senha")
    @PasswordField
    private String senha;

}
```

10.10. @PhoneField

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de texto comum, entretanto, com uma formatação de Telefone (9999-9999). Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {

    @Column
    @Field(prompt = "Telefone", label = "Telefone")
    @PhoneField
    private String phone;

}
```

10.11. @RichText

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de texto especial, no qual será permitido incluir formatação de texto. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {  
  
    @Column  
    @Field(prompt = "Texto", label = "Texto")  
    @RichText  
    private String texto;  
  
}
```

10.12. @TextArea

Usando esta anotação, você informa que um atributo da classe deve ser exibido como uma caixa de texto de múltiplas linhas, conhecida em HTML como TextArea. Veja um exemplo abaixo.

```
public class Bookmark implements Serializable {  
  
    @Column  
    @Field(prompt = "Texto", label = "Texto")  
    @TextArea  
    private String texto;  
  
}
```

Nas seções seguintes veremos como o componente trata estas anotações para que os atributos sejam exibidos como componentes de interface.

Tópicos Avançados do Vaadin

11.1. Eventos

Conforme discutido na seção que trata o padrão MVP, as classes de View não deveriam tratar diretamente os eventos de interesse do usuário. Por exemplo, quando o usuário clica em um botão “Salvar”, não deve ser responsabilidade da View chamar a classe de BusinessController diretamente. Esta tarefa cabe ao Presenter. É ele que deve tratar este evento de clique do botão, salvar os dados chamando o BusinessController e depois solicitar à View que seja exibida uma mensagem de sucesso para o usuário.

Então, como deve ocorrer a comunicação entre View e Presenter? Como o Presenter captura os eventos do usuário com a View? Existem diversas formas. É possível ter uma instância do Presenter dentro da View e fazer uma chamada direta. Ou, de uma forma mais desacoplada, a View pode lançar eventos que serão capturados por algum Presenter. E é neste ponto onde o mecanismo padrão de tratamento de eventos do CDI/Weld auxilia substancialmente. O componente faz forte uso deste mecanismo e já fornece alguns eventos bastante comuns para operações de CRUD.

Caso a classe de Presenter tenha interesse em saber quando uma determinada View é inicializada pela primeira vez, basta ter um método com a seguinte assinatura:

```
public void afterViewInitialization(@Observes @AfterViewInitialization MinhaView view) { }
```

A anotação `@Observes` é do CDI e informa que este método está observando eventos qualificados com `AfterViewInitialization` e que lançam o objeto `MinhaView`. De forma simples, sempre que uma View do tipo `MinhaView` for inicializada, este evento será lançado e o Presenter que tiver um método com a assinatura acima será chamado.

O componente Vaadin fornece diversos outros qualificadores de evento, como: `AfterViewRepainted`, `BeforeNavigateToView`, `ProcessClear`, `ProcessDelete`, `ProcessInvalidValue`, `ProcessItemSelection`, `ProcessMenuSelection` e `ProcessSave`. Muitas delas tratam eventos específicos de CRUD. A anotação `ProcessSave`, por exemplo, é lançada quando ocorre um clique em um botão de Salvar.

11.2. Criando Classes de View

Classes de visão são responsáveis pela criação de telas, através da composição de componentes que são fornecidos pelo Vaadin. O componente do Demoiselle fornece algumas classes abstratas que podem ser estendidas. Desta forma, sua classe já herdará alguns comportamentos que facilitarão a utilização do padrão MVP.

A classe mais simples é a `BaseVaadinView`. Estendendo esta classe, alguns eventos já serão lançados por padrão. Por exemplo, logo após a View ser inicializada, o evento `AfterViewInitialization` será lançado. Outra classe disponível é a `AbstractCrudView`. Como seu nome sugere, esta classe já fornece o comportamento padrão para tratar telas de CRUD. Esta classe já fornece dois componentes, por padrão: um `Table` e um `CrudForm`. Contudo, o posicionamento destes componentes na tela deve ser feito pelo próprio programador.

Uma vez estendida, o método `initializeComponent()` deve ser sobrescrito e usado para a criação dos componentes da interface. Caso a classe estendida seja `AbstractCrudView`, é neste momento que os componentes `Table` e `CrudForm` devem ser posicionados. Outros elementos podem ser adicionados, conforme a necessidade. Dentro de qualquer View o mecanismo de injeção de dependência está ativado por padrão. Desta forma, é possível injetar, por exemplo, o `ResourceBundle` para tratar a internacionalização de sua aplicação. É necessário também que sua classe de View seja anotada com `@View`. Trata-se de um qualificador que tem significado especial para o Demoiselle.

11.3. Criando Classes de Presenter

Uma vez criada a classe de `View`, cabe agora criar a classe de `Presenter` que é responsável por ela. No padrão MVP, normalmente, existe o relacionamento de um para um entre `View` e `Presenter`. Uma classe de `Presenter` típica é demonstrada no exemplo abaixo.

```
@Presenter
public class MainPresenter extends AbstractPresenter<MainView> {
}
```

É importante estereotipar sua classe com a anotação `@Presenter`. Ela possui significado especial para o `Demoiselle`. Após isto, é possível estender a classe `AbstractPresenter` e herdar alguns comportamentos que facilitarão a aplicação do padrão MVP. A partir daqui, pode-se ter este `Presenter` capturando os eventos lançados por `MainView`. Não há nenhum método a ser sobrescrito, como pode ser observado. Outra opção é estender a classe `AbstractCrudPresenter`, que já fornece operações de `CRUD`. O exemplo abaixo exemplifica seu uso.

```
@Presenter
public class ClubePresenter extends AbstractCrudPresenter<ClubeView, Clube, Long, ClubeBC> {

    public void processSave(@Observes @ProcessSave Clube clube) {
        doSave(clube);
    }

}
```

Podemos observar que são necessários alguns parâmetros extras. Primeiro, informamos qual `View` (`ClubeView`), depois qual a entidade que será tratada por este `Presenter` (`Clube`). Em seguida, é preciso informar o tipo do identificador da entidade, neste caso, `Long`. Por último, qual a classe de `BusinessController` para esta entidade (`ClubeBC`). Agora, é possível capturar os eventos de `CRUD` e repassar para métodos da classe-pai, como `doSave()`, `doDelete()` e etc.

11.4. Novos Componentes

O componente `Vaadin` fornece um componente chamado `CrudForm`. Este componente é, basicamente, uma extensão da classe `Form` do `Vaadin`, mas que adiciona comportamentos de `CRUD`. Por padrão, este `Form` já terá três botões. Além disto, estes botões já lançarão automaticamente os eventos `ProcessSave`, `ProcessDelete` e `ProcessClear`. Ao usar `CrudForm`, as anotações de atributos, já discutidas, serão usadas para construir automaticamente alguns campos relacionados aos atributos.

O `CrudForm` é capaz de construir automaticamente os campos, pois ele estende de `AutoForm`. Esta classe é que, de fato, trata as anotações colocadas nos atributos da entidade e transforma em campos equivalentes.

11.5. Paginação

A paginação também foi lembrada neste componente. Contudo, você verá uma paginação um pouco diferente daquela que normalmente vemos em aplicações web. Não teremos links para páginas ou setas para clicar. Toda a paginação é transparente, feita on-demand. Conforme a barra de rolagem sobe ou desce, a paginação ocorre automaticamente.

Para ter disponível o recurso de paginação nativo do `Demoiselle` com o `Vaadin`, basta usar a classe `PagedContainer` e seu método `create()`, que recebe dois parâmetros. O primeiro é a classe do tipo do bean

que será paginado. O segundo parâmetro é uma instância da interface `ResultListHandler`. Desta interface, deve ser implementado o método `handleResultList()`. Ele será chamado sempre que o objeto `Table` necessitar de uma nova consulta no banco para obter dados.

Parte VI. Demoiselle Monitoring

O *Demoiselle Monitoring* é um componente para monitoração de aplicações em Java desenvolvidas com o *Demoiselle Framework*.

Como objetivo geral, o componente fornece mecanismos que possibilitam a uma aplicação Java:

- responder a requisições provenientes de um servidor (modalidade *polling*)
- enviar automaticamente notificações a um servidor (modalidade *trapping*)



Nota

Atualmente o *Demoiselle Monitoring* considera as seguintes tecnologias e padrões de monitoramento: JMX, SNMP e Zabbix.

Introdução ao Monitoring

Num ambiente onde aplicações são disponibilizadas em inúmeros servidores heterogêneos, um dos maiores desafios técnicos é prover - de forma padronizada e confiável - a monitoração constante dos recursos que elas oferecem.

A monitoração de uma aplicação, serviço ou servidor pode ser motivada para fins como: disponibilidade, performance, segurança ou mesmo requisitos de negócio. Por exemplo, uma determinada aplicação crítica deve estar sempre no ar e com um tempo de resposta aceitável, além de estar protegida contra tentativas de invasão.

Alguns indicadores, tais como a quantidade de documentos emitidos por hora ou o número de transações efetuadas por minuto, podem ser coletados em intervalos regulares a pedido do cliente ou ainda para fins de tarifação (billing).

Em se tratando de uma solução robusta e profissional, existem tecnologias altamente avançadas e direcionadas para monitoração: os ditos NMS (Network Management Systems), sendo o Zabbix uma dessas opções. Entretanto, para fazer uso destes servidores de monitoração, é preciso dispor de um esforço adicional da equipe de desenvolvimento para codificar rotinas complexas e transversais à aplicação final. Estas rotinas poderiam não garantir o melhor desempenho para a monitoração e fatalmente seriam construídas novamente a cada necessidade de monitoração de uma nova aplicação.

Baseando-se no SNMP, o protocolo padrão para o gerenciamento de serviços de rede, e na especificação JMX para lidar com gerenciamento de modo genérico em Java, foi criado o componente *Demoiselle Monitoring*. Ele traz mecanismos que possibilitam a qualquer aplicação desenvolvida na linguagem Java a monitoração de seus indicadores nas duas modalidades existentes: polling e trapping.

12.1. Modalidades de monitoração: polling e trapping

Um NMS ¹ é capaz de monitorar hosts e serviços de duas maneiras: via polling ou trapping.

O método *polling* (também conhecido como *pull* ou *active check*) é o mais utilizado da monitoração. Tem a vantagem de manter a configuração centralizada no NMS e é executado por este em intervalos regulares ou sob demanda quando necessário. No polling a requisição parte do servidor NMS em direção ao host monitorado.

Por outro lado, o método *trapping* (também chamado de *push* ou *passive check*) são eventos iniciados e efetuados por processos ou aplicações externas, tendo seus resultados submetidos ao NMS para processamento. Ou seja, no trapping a requisição parte do host monitorado em direção ao servidor NMS.

O modo trapping é útil na monitoração de serviços que:

- sejam assíncronos em sua natureza e que não podem ser monitorados efetivamente através de polling de forma agendada e regular;
- estejam localizados atrás de um firewall e por isso não podem ser verificados de forma ativa pelo servidor de monitoração.

Exemplos de serviços assíncronos que são monitorados de forma passiva incluem traps SNMP e alertas de segurança - não podemos saber quantos (se algum) alertas ou traps serão recebidos durante um intervalo de tempo, portanto não é factível monitorar seu estado, por exemplo, a cada alguns minutos.

¹ *Network Management System*: sistema que monitora e controla os dispositivos gerenciados em uma rede.



Dica

Método *polling* (ou pull ou active check):

- Coleta iniciada pelo servidor em intervalos regulares
- Configuração centralizada
- Resposta enviada por um agente



Dica

Método *trapping* (ou push ou passive check):

- Evento iniciado pelo host monitorado
- Notificações e mensagens tempestivas
- Requisição proveniente de um trapper

12.2. Sobre o componente Demoiselle Monitoring

O componente *Demoiselle Monitoring* fornece duas abordagens de monitoração do tipo *polling* e *trapping* para as aplicações desenvolvidas em Java: através de *agente* e *trapper SNMP* ou de *agente* e *trapper ZABBIX*.



Importante

Os agentes e trappers são codificados inteiramente em Java e não fazem uso de binários externos (ex: `snmptrap` ou `zabbix_sender`).

Do ponto de vista fundamental, a abordagem usando o protocolo SNMP é a mais adequada, especialmente por oferecer a possibilidade de os indicadores serem coletados por qualquer cliente desse protocolo, além do que SNMP é o protocolo padrão para o gerenciamento de recursos de rede e serviços. A ideia de hierarquizar os recursos na árvore de serviços do SNMP (i.e. MIB Tree) torna-se obrigatória, o que pode contribuir para uma melhor organização dessa monitoração.

Ao se optar pela abordagem Zabbix, apenas servidores deste tipo poderão consumir os indicadores expostos pelo agente. Por outro lado, a máquina virtual (JVM) pode ser configurada para exportar via interface JMX os MBeans registrados nela, os quais podem ser reutilizados em outras suítes.



Dica

É possível utilizar na mesma aplicação todas abordagens de monitoração disponibilizadas pelo componente: JMX, SNMP e Zabbix.

Configuração do Monitoring

13.1. Instalação do componente

Para instalar o componente *Demoiselle Monitoring* na aplicação, basta adicionar a sua dependência no arquivo `pom.xml` do projeto gerenciado pelo Maven:

```
<dependency>
  <groupId>br.gov.frameworkdemoiselle.component</groupId>
  <artifactId>demoiselle-monitoring</artifactId>
  <version>2.0.0</version>
</dependency>
```

13.2. Arquivos de configuração

Os parâmetros utilizados pelo componente são concentrados no arquivo `demoiselle.properties`. Maiores informações sobre eles são detalhadas de forma distribuída nos capítulos seguintes.

Informações sobre a segurança no protocolo SNMP (especificamente para a versão 3) são armazenadas no arquivo `snmp-security.xml`.

Monitorando via JMX

A *Java Management Extensions (JMX)* é uma API padrão para gerenciamento e monitoração de recursos tais como aplicações, dispositivos, serviços e a máquina virtual Java (JVM). A tecnologia JMX foi desenvolvida através de um processo *Java Community Process (JCP)* que envolveu a *JSR-003 (Java Management Extensions)* e a *JSR-160 (JMX Remote API)*.

A tecnologia JMX é utilizada geralmente nesses casos:

- consulta e mudança de parâmetros de configuração em uma aplicação
- acúmulo de dados estatísticos sobre o comportamento de uma aplicação e disponibilização destes
- notificação de mudanças de estado e condições de erro

A API JMX inclui acesso remoto, o que permite que um programa de gerenciamento remoto possa interagir com uma aplicação em execução para esses propósitos.



Dica

Para saber mais sobre a *Tecnologia JMX* e a *API JMX*, acesse o link: <http://download.oracle.com/javase/6/docs/technotes/guides/jmx/>.

Nas seções a seguir veremos como usar os recursos da *API JMX* de uma forma bem prática e simples através do componente *Demoiselle Monitoring*.

14.1. Criando e exportando um MBean

Os recursos disponíveis na *JMX* são representados por objetos chamados *MBeans (Managed Beans)*. MBeans são classes Java dinamicamente carregadas e instanciadas e que podem ser usadas para se obter e ou alterar configurações das aplicações, coleta de estatísticas e notificação de eventos.

O primeiro passo para a criação de um MBean consiste na criação de uma interface com o sufixo “MBean” que expõe os métodos referentes aos atributos a serem exportados. Eis um exemplo:

```
public interface ExemploMBean {  
  
    String getAtributo();  
  
}
```

Em seguida, é preciso criar uma classe que implemente a referida interface, preenchendo com códigos os métodos. Além disso, a classe deve ser estereotipada com a anotação `@MBean`. Veja:

```
@MBean  
public class Exemplo implements ExemploMBean {  
  
    public String getAtributo() {  
        return "abc";  
    }  
}
```

}



Importante

Para ser um *Standard MBean* compatível com a API JMX, é necessário que a classe em questão implemente uma interface de mesmo nome acrescida do sufixo “MBean”.



Dica

A anotação `@MBean` é fornecida pelo *Demoiselle Monitoring* e encontra-se no pacote `br.gov.frameworkdemoiselle.monitoring.stereotype`.

Uma vez em execução a aplicação, você pode usar o programa `jconsole`¹ para verificar se o MBean foi devidamente registrado e exportado no servidor JMX. Veja o resultado:

Figura 14.1. Consultando MBeans exportados no servidor JMX.

14.2. Especificando um nome para o MBean

Cada MBean exportado pelo servidor JMX possui um nome único no qual é registrado. Ao usar a anotação `@MBean`, o seguinte padrão é adotado: “`nome.do.pacote:name=NomeDaClasse`”.

No exemplo anterior, o MBean foi registrado com o nome “`br.gov.frameworkdemoiselle.monitoring.mbean:name=Exemplo`”, pois a classe de nome `Exemplo` pertencia ao pacote `br.gov.frameworkdemoiselle.monitoring.mbean`.

É possível sobrescrever esse padrão ao utilizar a anotação `@Name` em conjunto com `@MBean` na declaração da classe, tal como ilustrado abaixo:

```
@MBean
@Name ( "br.gov.frameworkdemoiselle:name=OutroNome" )
```



Dica

A anotação `@Name` está no pacote `br.gov.frameworkdemoiselle.annotation`.

Dessa forma, o objeto MBean será registrado com o nome escolhido no servidor JMX. Veja:

Figura 14.2. Consultando MBean registrado com nome customizado.

¹ `jconsole` é uma ferramenta gráfica compatível com JMX para monitorar máquinas virtuais Java locais ou remotas. Para maiores informações, acesse o link <http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html>.

14.3. Atributos de leitura e escrita

O MBean criado como exemplo possuía apenas um atributo, de nome `Atributo`, o qual era somente de leitura (*read only*). Para que o valor de um atributo de MBean possa ser alterado, e não apenas consultado, basta implementar um método do tipo *setter* na classe. Primeiro altere a interface:

```
public interface ExemploMBean {  
  
    String getAtributo();  
  
    void setAtributo(String atributo);  
  
}
```

E em seguida implemente todos os métodos na classe:

```
@MBean  
public class Exemplo implements ExemploMBean {  
  
    private String atributo;  
  
    public String getAtributo() {  
        return atributo;  
    }  
  
    public void setAtributo(String atributo) {  
        this.atributo = atributo;  
    }  
  
}
```

Desta forma, o valor do atributo `Exemplo` do MBean poderá ser consultado e ou modificado via JMX.

Monitorando via SNMP (polling)

Para a abordagem SNMP, a arquitetura envolve a criação e exportação de MBeans. No mesmo servidor Java será levantado um *Agente SNMP* na porta 1161 em UDP.



Nota

O *Agente SNMP* não é um servidor SNMP convencional, tal como [Net-SNMP](http://www.net-snmp.org/) [http://www.net-snmp.org/], mas um programa que entende o protocolo SNMP e responde a requisições referentes a uma árvore de serviços restrita.

Internamente existe um mapeamento entre atributos de um MBean exportado no servidor JMX e os objetos disponibilizados na árvore SNMP.

15.1. Configuração do Agente SNMP

Alguns parâmetros devem ser configurados na aplicação para o levantamento do serviço SNMP pelo componente *Demoiselle Monitoring*. Tais configurações são feitas no arquivo `demoiselle.properties`:

```
frameworkdemoiselle.monitoring.snmp.agent.enabled = true
frameworkdemoiselle.monitoring.snmp.agent.protocol = udp
frameworkdemoiselle.monitoring.snmp.agent.port = 1161
frameworkdemoiselle.monitoring.snmp.agent.address = *
frameworkdemoiselle.monitoring.snmp.agent.mib.root = 1.3.6.1.4.1.35437.1
```

O acesso aos objetos disponibilizados nas árvores de serviço pelo agente SNMP pode ser restringido, mas por enquanto consideremos apenas SNMP v1 e v2. Crie o arquivo `snmp-security.xml` com o conteúdo abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<snmp-security>
  <access-control>
    <views>
      <view name="full">
        <include subtree="1.3" />
      </view>
    </views>
    <groups>
      <group name="v1v2group">
        <security name="public" model="SNMPv1"/>
        <security name="public" model="SNMPv2c"/>
        <access context="" model="ANY" auth="false" priv="false" exact="true">
          <read view="full" />
          <write view="full" />
          <notify view="full" />
        </access>
      </group>
    </groups>
  </access-control>
</snmp-security>
```

Tendo definidos esses arquivos, ao executar a aplicação o *Agente SNMP* será automaticamente inicializado.

15.2. Criação do MBean e da árvore MIB

Uma vez devidamente configurado o *Agente SNMP*, as árvores MIB a serem disponibilizadas por ele devem ser criadas pelo desenvolvedor. Veremos como fazer isso nessa seção. Primeiramente é preciso criar uma interface simples contendo os métodos a serem expostos, tal como no exemplo:

```
public interface ExemploMBean {  
  
    String getAtributo();  
  
}
```

Em seguida, uma classe anotada com os estereótipos `@MBean` e `@MIB` deve implementar os métodos dessa interface:

```
@MBean  
@MIB( ".2" )  
public class Exemplo implements ExemploMBean {  
  
    @OID( ".1" )  
    public String getAtributo() {  
        return "abc";  
    }  
  
}
```



Importante

Lembre-se de que a classe deve implementar uma interface de mesmo nome acrescida do sufixo "MBean".

Opcionalmente os objetos a serem disponibilizados na árvore podem ser marcados com a anotação `@OID`.



Dica

As anotações `@MBean` e `@MIB` estão no pacote `br.gov.frameworkdemoiselle.monitoring.stereotype`, enquanto que `@OID` se localiza em `br.gov.frameworkdemoiselle.monitoring.annotation.snmp`.

15.3. Consumindo os recursos via cliente SNMP

Tendo o serviço SNMP levantado pelo agente, podemos usar ferramentas tais como um cliente SNMP qualquer para efetuar operações. Eis um exemplo de utilização do `snmpget`¹ para obter o valor do objeto exportado na árvore:

¹ Para maiores informações sobre o `snmpget`, acesse: <http://www.net-snmp.org/docs/man/snmpget.html>


```
$ snmpget -v 1 -c public localhost:1161 1.3.6.1.4.1.35437.1.2.1.0
SNMPv2-SMI::enterprises.35437.1.2.1.0 = STRING: "abc"
```

Note que o *OID* (i.e., identificador de objeto) usado foi o “1.3.6.1.4.1.35437.1.2.1.0”. Ele é resultante da concatenação de “1.3.6.1.4.1.35437.1” (proveniente do arquivo `demoiselle.properties`) com “.2” (declarado em `@MIB`) e “.1” (declarado em `@OID`). O sufixo “.0” é um indicador de valor escalar em SNMP e sempre será incluído automaticamente.



Dica

No *Debian GNU/Linux* é disponibilizado o pacote `snmp` (<http://packages.debian.org/snmp/>): uma coleção de clientes de linha de comando destinados a efetuar requisições SNMP aos agentes. Para instalar esse pacote, basta executar `apt-get install snmp` como super-usuário.

15.4. Definindo o tipo de dados SNMP

O protocolo SNMP reconhece tipos de dados próprios. O componente *Demoiselle Monitoring* internamente faz um mapeamento automático dos tipos de dados do Java (primitivos e de referência) para os tipos de dados SNMP. Entretanto, existem situações em que o tipo SNMP pode ser melhor especificado. Por exemplo, ao declarar o tipo `int` no retorno do método a seguir, este será convertido para o tipo `integer` em SNMP:

```
@OID( ".2" )
public int getContador() {
    return 1500;
}
```

```
$ snmpget -v 1 -c public localhost:1161 enterprises.35437.1.2.2.0
SNMPv2-SMI::enterprises.35437.1.2.2.0 = INTEGER: 1500
```

Os tipos de dados SNMP v1 e v2 estão disponíveis na forma de anotações Java. Para exportar o objeto do exemplo acima sob o tipo específico *counter*, implemente o método usando a anotação `@Counter32`:

```
@OID( ".2" )
@Counter32
public int getContador() {
    return 1500;
}
```

Ao consultar esse objeto usando `snmpget`, perceba que agora o tipo de dados *counter* está sendo devidamente especificado pelo agente ao cliente:

```
$ snmpget -v 1 -c public localhost:1161 enterprises.35437.1.2.2.0
SNMPv2-SMI::enterprises.35437.1.2.2.0 = Counter32: 1500
```

Veja na tabela a seguir cada um dos tipos de dados SNMP e as anotações correspondentes.

Tabela 15.1. Tipos de dados SNMP

Tipo de dado	Anotação	Descrição
Counter	@Counter32	Um contador é um número inteiro não-negativo incrementado até que um valor máximo seja alcançado, voltando assim ao zero. A SMIv1 (RFC-1155 [http://tools.ietf.org/html/rfc1155]) especifica o tamanho de 32 bits para o tipo <i>counter</i> .
Counter64	@Counter64	Trata-se de um <i>counter</i> definido na SMIv2 (RFC-2578 [http://tools.ietf.org/html/rfc2578]) capaz de armazenar valores com 64 bits.
Gauge	@Gauge32	O <i>gauge</i> representa um inteiro não-negativo, que pode oscilar entre zero e um valor máximo. Este valor máximo é $2^{32}-1$ (i.e., o decimal 4.294.967.295).
Integer	@Integer32	O tipo de dados <i>integer</i> é um número inteiro negativo ou positivo na faixa de -2^{31} a $2^{31}-1$.
IP Address	@IPAddress	Representa um endereço de Internet de 32 bits através de uma string de 4 octetos, ou seja, um endereço IPv4. O valor é codificado como 4 bytes na ordem de rede de computadores.
Object Identifier (OID)	@ObjectIdentifier	Os <i>Object IDs</i> são provenientes do conjunto de todos os identificadores de objeto alocados de acordo com as regras definidas na especificação ASN.1 (http://www.itu.int/ITU-T/asn1/introduction/index.htm). Uma instância desse tipo pode ter até 128 sub-identificadores. Além disso, cada sub-identificador não deve exceder o valor $2^{32}-1$.
Octet String	@OctetString	Representa dados arbitrários binários ou textuais, ou seja, sequências ordenadas de 0 a 65.535 octetos.
Time Ticks	@TimeTicks	Representa um número não-negativo que especifica o tempo passado entre dois eventos, em unidades de centésimos de segundo. Compreende a faixa de valores de 0 a $2^{32}-1$.



Nota

O tipo de dados `String` do Java é implicitamente convertido para `@OctetString`. Já os tipos de dados numéricos, sejam eles primitivos (ex: `int`, `short`, `long`) ou de referência (ex: `Integer`, `Short`, `Long`), são convertidos para `Integer32`.



Dica

As anotações de tipos de dados SNMP fornecidas pelo componente estão no pacote `br.gov.frameworkdemoiselle.monitoring.annotation.snmp.type`.

15.5. Exportando um objeto para leitura e escrita via SNMP

Os exemplos de objetos de árvores de serviços que vimos até o momento eram somente de leitura. É possível também com o componente *Demoiselle Monitoring* disponibilizar objetos que possam ser lidos ou modificados por um cliente SNMP.

A maneira mais fácil de implementar isso é criando um campo na classe do MBean, ou seja, uma variável que possua métodos do tipo *getter* e *setter*. A anotação `@OID` para o SNMP deve ser declarada na variável, e não nos métodos. Além disso, é preciso indicar que o campo será modificável através da anotação `@ReadWrite`. Veja um exemplo:

```
@OID( ".3" )
@ReadWrite
private long campo;

public long getCampo() {
    return campo;
}

public void setCampo(long campo) {
    this.campo = campo;
}
```

Feito isso, o valor exportado pela árvore MIB agora pode ser consultado e alterado por um cliente SNMP. Veja como fica o uso do comando `snmpget`:

```
$ snmpget -v 1 -c public localhost:1161 enterprises.35437.1.2.3.0
SNMPv2-SMI::enterprises.35437.1.2.3.0 = INTEGER: 0
```

Este valor de objeto pode ser alterado usando o comando `snmpset`², tal como exemplificado:

```
$ snmpset -v 1 -c public localhost:1161 enterprises.35437.1.2.3.0 i 256
SNMPv2-SMI::enterprises.35437.1.2.3.0 = INTEGER: 256
```

É possível ainda restringir os valores aceitados para um campo numérico. Para isso, basta usar a anotação `@AllowedValues` na sua declaração. No exemplo a seguir, apenas os valores 1, 2 ou 3 poderão ser atribuídos ao campo:

```
@OID( ".3" )
@ReadWrite
@AllowedValues({1, 2, 3})
private long campo;
```

² Para maiores informações sobre o `snmpset`, acesse: <http://www.net-snmp.org/docs/man/snmpset.html>

15.6. Uma árvore MIB mais complexa

Veremos agora um caso prático, onde uma aplicação de gestão escolar precisa fornecer monitoração sobre indicadores de negócio e de sistema. A proposta é criar uma árvore de serviços (i.e., uma MIB tree) que exponha esses valores na forma de objetos SNMP. O endereço "1.3.6.1.4.1.35437.1.75048.2011.1" será a raiz dessa árvore específica (sob o ramo `enterprises.serpro.apps`).

O primeiro passo é criar a interface contendo todos os métodos referentes aos objetos a serem exportados. Lembre-se de usar o sufixo "MBean":

```
public interface EscolaMonitoringMBean {

    String getVersaoAplicacao();

    int getQtdTurmasIncluidas();

    long getQtdAlunosMatriculados();

    String getUltimoUsuarioLogado();

    int getNivelLog();

    void setNivelLog(int nivelLog);

}
```

Em seguida, crie a classe que implementa a interface anterior. É necessário que ela seja anotada com `@MBean` e `@MIB`. Veja:

```
@MBean
@Name("br.gov.frameworkdemoiselle:name=Escola")
@MIB("1.3.6.1.4.1.35437.1.75048.2011.1")
public class EscolaMonitoring implements EscolaMonitoringMBean {

    private static final String VERSAO = "2.4.1-BETA";
    private static final String[] USUARIOS = { "Fulano", "Sicrano", "Beltrano" };

    private int qtdTurmas = 0;
    private long qtdAlunos = 0;

    @OID(".1")
    @Override
    public String getVersaoAplicacao() {
        return VERSAO;
    }

    @OID(".2")
    @Counter32
    @Override
    public int getQtdTurmasIncluidas() {
        this.qtdTurmas += (int) (Math.random() * 10);
        return this.qtdTurmas;
    }

}
```

```

@OID(".3")
@Gauge32
@Override
public long getQtdAlunosMatriculados() {
    this.qtdAlunos = (int) (Math.random() * 100) + 100;
    return this.qtdAlunos;
}

@OID(".4")
@Override
public String getUltimoUsuarioLogado() {
    int pos = (int) (Math.random() * USUARIOS.length);
    return USUARIOS[pos];
}

@OID(".5")
@ReadWrite
@AllowedValues({1, 2, 3, 4})
private int nivelLog = 1;

@Override
public int getNivelLog() {
    return this.nivelLog;
}

@Override
public void setNivelLog(int nivelLog) {
    this.nivelLog = nivelLog;
}
}

```

Neste caso de exemplo, a árvore montada a partir de `EscolaMonitoring` disponibilizará cinco objetos via SNMP. O objeto `VersaoAplicacao` retornará sempre o mesmo valor de *string*. Já `QtdTurmasIncluidas` e `QtdAlunosMatriculados` retornarão números inteiros (respectivamente do tipo *counter* e *gauge*) que serão incrementados aleatoriamente a cada chamada. O objeto `UltimoUsuarioLogado` retorna randomicamente um dos três nomes possíveis de usuário. E `NivelLog` armazena um número que pode ser consultado ou modificado (com restrições de valores).

Note que chamadas consecutivas ao serviço SNMP com o comando `snmpwalk`³ retornarão valores distintos para alguns dos objetos, tal como esperado:

```

$ snmpwalk -v 1 -c public localhost:1161 enterprises.35437.1.75048
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.1.0 = STRING: "2.4.1-BETA"
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.2.0 = Counter32: 115
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.3.0 = Gauge32: 129
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.4.0 = STRING: "Fulano"
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.5.0 = INTEGER: 1
End of MIB

```

```

$ snmpwalk -v 1 -c public localhost:1161 enterprises.35437.1.75048

```

³ Para maiores informações sobre o `snmpset`, acesse: <http://www.net-snmp.org/docs/man/snmpwalk.html>

```
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.1.0 = STRING: "2.4.1-BETA"
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.2.0 = Counter32: 121
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.3.0 = Gauge32: 166
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.4.0 = STRING: "Beltrano"
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.5.0 = INTEGER: 1
End of MIB
```

```
$ snmpwalk -v 1 -c public localhost:1161 enterprises.35437.1.75048
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.1.0 = STRING: "2.4.1-BETA"
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.2.0 = Counter32: 128
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.3.0 = Gauge32: 189
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.4.0 = STRING: "Beltrano"
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.5.0 = INTEGER: 1
End of MIB
```

Veja na figura a seguir as árvores de serviço SNMP sendo consultadas através de `SNMP walk` com a ferramenta gráfica *iReasoning MIB Browser* [<http://ireasoning.com/mibbrowser.shtml>].

Figura 15.1. Consultando as árvores de serviços SNMP.

15.7. Definição de segurança (SNMPv3)

Nos exemplos que vimos até agora, só fizemos uso da versão 1 do protocolo SNMP. Todavia, existem situações em que as outras versões existentes, 2c e 3, sejam consideradas. A versão 2 implementa basicamente tipos de dados adicionais à versão 1, particularmente os de 64 bits (ex: *counter64*). Já a versão 3 traz como principal feature o controle de acesso, isto é, parâmetros mínimos de segurança. Este é justamente o tema desta seção.

Para com que a versão 3 do protocolo SNMP seja utilizada, é preciso configurar devidamente os seguintes mecanismos existentes no agente:

- Secret-Key Authentication
- Privacy Using Conventional Encryption
- User-Based Security Model (USM)
- View-Based Access Control Model (VACM)

No componente *Demoiselle Monitoring* essa parametrização é feita em um único arquivo, o `snmp-security.xml`, contendo a estrutura indicada no exemplo a seguir:

```
<?xml version="1.0" encoding="UTF-8"?>
<snmp-security>
  <security>
    <user name="escola">
      <auth protocol="SHA" pass="senha123" />
      <priv protocol="AES" pass="senha456" />
    </user>
    <user name="zabbix">
      <auth protocol="MD5" pass="senha321" />
      <priv protocol="DES" pass="senha654" />
    </user>
  </security>
</snmp-security>
```

```

</security>
<access-control>
  <views>
    <view name="escolaReadView">
      <include subtree="1.3.6.1.4.1.35437.1.75048" />
    </view>
    <view name="escolaWriteView">
      <include subtree="1.3.6.1.4.1.35437.1.75048" />
    </view>
    <view name="escolaNotifyView">
      <include subtree="1.3.6.1.4.1.35437.1.75048" />
    </view>
  </views>
  <groups>
    <group name="v3escola">
      <security name="escola" model="USM" />
      <security name="zabbix" model="USM" />
      <access model="USM" auth="true" priv="true" exact="true">
        <read view="escolaReadView" />
        <write view="escolaWriteView" />
        <notify view="escolaNotifyView" />
      </access>
    </group>
  </groups>
</access-control>
</snmp-security>

```

Ou seja, os seguintes itens precisam ser detalhados neste arquivo:

- nomes de *usuários* para a autenticação
- senha para *autenticação* do usuário, usando um dos seguintes protocolos de criptografia: SHA ou MD5
- senha para *privacidade* do usuário, usando um dos seguintes protocolos de criptografia: DES, 3DES ou AES (128, 192 e 256 bits)
- *visões* destinadas a leitura, escrita e notificação nas árvores de serviço (MIBs)
- *grupos* de acesso para vincular usuários, modelos de segurança, restrições de autenticação e privacidade às visões



Importante

A configuração anterior sugerida para o arquivo `snmp-security.xml` considerava apenas as versões 1 e 2c do protocolo SNMP. Ou seja, para a segurança bastava indicar o nome da comunidade (i.e., `public`).

Considerando a nova configuração de segurança sugerida, ao utilizar os comandos das ferramentas clientes com o protocolo SNMPv3, é preciso passar argumentos adicionais, tais como o nome do usuário, a forma escolhida para autenticação e privacidade e as respectivas senhas e algoritmos de criptografia.

A instrução `snmpget` fica assim para o usuário `escola` usando SHA na autenticação e AES na privacidade:

```
$ snmpget -v 3 -l authPriv -u escola -a SHA -A senha123 -x AES -X senha456 \
```

```
localhost:1161 enterprises.35437.1.75048.2011.1.3.0
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.3.0 = Gauge32: 134
```

Nos aplicativos clientes será preciso informar esses parâmetros de segurança do SNMPv3. Veja como isso é apresentado no *iReasoning MIB Browser*.

Figura 15.2. Configuração de segurança SNMPv3.

Para testar o controle de acesso (VACM) do SNMPv3, ao requisitar um objeto pertencente a uma visão não permitida ao grupo do usuário, o resultado é exibido abaixo:

```
$ snmpget -v 3 -l authPriv -u escola -a SHA -A senha123 -x AES -X senha456 \
localhost:1161 enterprises.35437.1.2.2.0
SNMPv2-SMI::enterprises.35437.1.2.2.0 = No Such Object available on this agent at this OID
```

Além disso, ao executar o comando `snmpwalk`, apenas os ramos da árvore disponíveis para o grupo do usuário são exibidos:

```
$ snmpwalk -O n -v 3 -l authPriv -u escola -a SHA -A senha123 -x AES -X senha456 localhost:1161 .
.1.3.6.1.4.1.35437.1.75048.2011.1.1.0 = STRING: "2.4.1-BETA"
.1.3.6.1.4.1.35437.1.75048.2011.1.2.0 = Counter32: 19
.1.3.6.1.4.1.35437.1.75048.2011.1.3.0 = Gauge32: 184
.1.3.6.1.4.1.35437.1.75048.2011.1.4.0 = STRING: "Sicrano"
.1.3.6.1.4.1.35437.1.75048.2011.1.5.0 = INTEGER: 1
.1.3.6.1.4.1.35437.1.75048.2011.1.5.0 = No more variables left in this MIB View (It is past
the end of the MIB tree)
```

De maneira semelhante, ao utilizar o usuário `zabbix` com os protocolos MD5 para autenticação e DES para privacidade, além das respectivas senhas, a instrução para o `snmpget` é a seguinte:

```
$ snmpget -v 3 -l authPriv -u zabbix -a MD5 -A senha321 -x DES -X senha654 \
localhost:1161 enterprises.35437.1.75048.2011.1.3.0
SNMPv2-SMI::enterprises.35437.1.75048.2011.1.3.0 = Gauge32: 138
```

Caso a senha de autenticação estiver incorreta, o resultado é o seguinte:

```
$ snmpget -v 3 -l authPriv -u zabbix -a MD5 -A senha_errada -x DES -X senha654 \
localhost:1161 enterprises.35437.1.75048.2011.1.3.0
snmpget: Authentication failure (incorrect password, community or key)
```

Entretanto, se a senha de privacidade for inválida, ocorre um timeout na requisição:

```
$ snmpget -v 3 -l authPriv -u zabbix -a MD5 -A senha321 -x DES -X senha_errada \
localhost:1161 enterprises.35437.1.75048.2011.1.3.0
```



```
Timeout: No Response from localhost:1161.
```

E por fim, caso as formas de autenticação e privacidade especificadas no cliente forem diferentes das configuradas no agente, a resposta é a seguinte:

```
$ snmpget -v 3 -l authNoPriv -u zabbix -a MD5 -A senha321 \  
    localhost:1161 enterprises.35437.1.75048.2011.1.3.0  
Error in packet  
Reason: authorizationError (access denied to that object)  
Failed object: SNMPv2-SMI::enterprises.35437.1.75048.2011.1.3.0
```

Monitorando via SNMP (trapping)

O componente *Demoiselle Monitoring* fornece um mecanismo simples e transparente para uma aplicação Java enviar notificações do tipo trap a um servidor SNMP. Trata-se de um código escrito inteiramente em linguagem Java que faz o papel do utilitário `snmptrap`¹.

16.1. Configuração para os trappers SNMP

Alguns parâmetros devem ser configurados na aplicação para o envio de traps SNMP pelo componente *Demoiselle Monitoring*. Tais configurações são feitas no arquivo `demoiselle.properties`:

```
frameworkdemoiselle.monitoring.snmp.trapper.protocol = udp
frameworkdemoiselle.monitoring.snmp.trapper.server = 127.0.0.1
frameworkdemoiselle.monitoring.snmp.trapper.port = 162
frameworkdemoiselle.monitoring.snmp.trapper.community = public
frameworkdemoiselle.monitoring.snmp.trapper.version = v1
frameworkdemoiselle.monitoring.snmp.trapper.enterprise = 1.3.6.1.4.1.35437.1.1.2425.2011
```

Esses parâmetros definem o endereço e a forma de comunicação com o servidor SNMP que receberá as traps, além de especificar alguns campos do cabeçalho das mensagens.

16.2. Criação e utilização do Trapper SNMP

Ao utilizar o *Demoiselle Monitoring*, o desenvolvedor poderá implementar facilmente suas próprias *trappers*. Para isso, ele pode inicialmente criar uma interface contendo os métodos que enviarão as mensagens. Eis um exemplo:

```
public interface MyTrapper {

    void sendFailure(String message);

}
```

Em seguida ele criará a respectiva classe, anotando com o estereótipo `@Trapper` e o qualificador `@SNMP` (ficará implícito que se trata de um “trapper para SNMP”). Veja como é simples:

```
@Trapper
@SNMP
public class MyTrapperSNMP implements MyTrapper {

    public void sendFailure(String message) { }

}
```

¹ Para maiores informações sobre o `snmptrap`, acesse: <http://www.net-snmp.org/docs/man/snmptrap.html>



Nota

Lembre-se de que criar uma interface para o trapper e em seguida implementá-la não é obrigatório, mas apenas um direcionamento. Se preferir, você pode construir a classe trapper diretamente (isto é, sem usar o `implements`). Entretanto, a aplicação será mais flexível ao considerar o uso de interfaces na arquitetura de seus componentes.



Importante

Note que os métodos da classe trapper não precisam ser codificados, ou seja, podem ficar vazios. Isso porque neles atuarão interceptadores do CDI. Na versão 1.x do componente isso era realizado através de aspectos com o AspectJ.

Finalmente, para utilizar esse trapper em qualquer parte da aplicação, basta declarar um campo com o tipo da interface e anotá-lo com `@Inject`:

```
public class TrapperTest {

    @Inject
    @SNMP
    private MyTrapper trapper;

    public void sendFailureMessage() {
        trapper.sendFailure("Hello, snmpd...");
    }

}
```

Uma vez invocado o método `sendFailure()`, o interceptador agirá sobre ele e fará o envio, de forma assíncrona, da mensagem ao servidor SNMP na forma de um trap.

Para testar essa funcionalidade, utilize um servidor SNMP, tal como *Net-SNMP*². Uma sugestão é iniciá-lo com o comando `snmptrapd -f -Le`. Inspecionando a LOG do servidor SNMP, eis os registros referentes aos comandos executados pelo exemplo em Java:

```
2011-09-14 14:26:22 0.0.0.0(via UDP: [127.0.0.1]:10728) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011 Enterprise Specific Trap (0)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.0 = STRING: "Hello, snmpd..."
```

Observe os valores que foram repassados na mensagem da trap SNMP enviada ao servidor. O OID da empresa (`enterprises.35437.1.1.2425.2011`) foi obtido através do parâmetro `frameworkdemoiselle.monitoring.snmp.trapper.enterprise` do arquivo `demoiselle.properties`. A versão do protocolo SNMP, o endereço e a porta do servidor e a comunidade também vieram desse arquivo.

² *Net-SNMP* é um serviço daemon que escuta e responde requisições SNMP dos clientes. Para maiores informações, acesse o site <http://net-snmp.sourceforge.net/>.

A única variável da mensagem SNMP foi definida automaticamente com o sufixo `.1.0` e o tipo de dados `STRING`. O valor da string `"Hello, snmpd..."` provém da chamada do método `trapper.sendFailure()` no código Java.

16.3. Customizando OIDs nas traps SNMP

Num caso prático é interessante e recomendável que os códigos de identificação de objetos (OIDs) da trap SNMP sejam customizados para a aplicação, especialmente quando mensagens diferentes são disparadas pelo trapper.

O *Demoiselle Monitoring* considera uma classe trapper como sendo uma árvore de serviços (i.e., MIB Tree) e cada um dos seus métodos pode se tornar um ramo dela. Para customizar essas OIDs, basta declarar a classe trapper com a anotação `@MIB` e anotar cada um dos seus métodos com `@OID`.

Veja um exemplo:

```
@Trapper
@SNMP
@MIB(".1.1")
public class MyTrapperSNMP implements MyTrapper {

    @OID(".5")
    public void sendFailure(String message) { }

    @OID(".7")
    public void sendInformation(String message) { }

}
```



Dica

Para ambas as anotações `@MIB` e `@OID` vale a seguinte regra: se a string iniciar com `"."`, haverá concatenação da OID hierarquicamente superior (código relativo), caso contrário será considerada integralmente a OID especificada (código absoluto).

Ao executar o código Java invocando os métodos `trapper.sendFailure("Fatal Failure")` e `trapper.sendInformation("Simple Info")`, veja como fica o resultado no servidor SNMP:

```
2011-09-15 09:32:27 0.0.0.0(via UDP: [127.0.0.1]:-12829) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.5 Enterprise Specific Trap (0)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.5.1.0 = STRING: "Fatal Failure"

2011-09-15 09:32:35 0.0.0.0(via UDP: [127.0.0.1]:-26520) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.7 Enterprise Specific Trap (0)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.7.1.0 = STRING: "Simple Info"
```

16.4. Definindo um código específico para a trap SNMP

Na verdade uma trap SNMPv1 contém três elementos além de suas variáveis ³: a identificação do objeto gerenciado (*enterprise*) e os campos tipo de trap genérico (*generic-trap*) e código específico de trap (*specific-trap*). Para traps customizadas, o que é o caso do componente *Demoiselle Monitoring*, o tipo de trap genérico será `enterpriseSpecific(6)`, restando a possibilidade de definir o código específico, cujo valor default é 0.

Na classe trapper, para definir o código específico de trap, basta usar a anotação `@SpecificTrap` no método desejado. Segue um exemplo de classe com essa customização:

```
@Trapper
@SNMP
public class MyTrapperSNMP implements MyTrapper {

    @SpecificTrap(4)
    public void sendFailure(String message) { }

    @SpecificTrap(6)
    public void sendInformation(String message) { }

}
```

Ao executar o código Java, observe os valores do campo `Enterprise Specific Trap` nas traps enviadas ao servidor SNMP:

```
2011-09-15 10:32:15 0.0.0.0(via UDP: [127.0.0.1]:-20449) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011 Enterprise Specific Trap (4)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.0 = STRING: "Fatal Failure"

2011-09-15 10:32:21 0.0.0.0(via UDP: [127.0.0.1]:-11555) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011 Enterprise Specific Trap (6)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.0 = STRING: "Simple Info"
```

16.5. Enviando traps SNMP com múltiplas variáveis

Nas seções anteriores, vimos como implementar classes trapper com o componente *Demoiselle Monitoring* definindo OIDs e tipos específicos nas mensagens enviadas ao servidor SNMP. Entretanto, existe a possibilidade de serem enviadas múltiplas variáveis na mesma mensagem SNMP. Até o momento nos exemplos utilizamos uma única variável do tipo string.

Enviar múltiplas variáveis em uma trap SNMP é muito simples: basta o método em questão possuir diversos argumentos. Os tipos das variáveis nos argumentos podem ser primitivos ou de referência. A conversão dos tipos de dados do Java para os respectivos tipos do SNMP ocorre de forma automática.

Veja um exemplo de trapper contendo dois métodos sobrecarregados de nome `sendVars()`:

```
@Trapper
@SNMP
```

³ Para maiores informações sobre o formato das traps SNMP, consulte a [RFC-1157](http://www.ietf.org/rfc/rfc1157.txt) [http://www.ietf.org/rfc/rfc1157.txt].

```
@MIB(".1.1")
public class MyTrapperSNMP implements MyTrapper {

    @OID(".2")
    public void sendVars(String s, int i, long l, boolean b) { }

    @OID(".4")
    public void sendVars(Object o, Integer i, Long l, Boolean b) { }

}
```

Eis um exemplo de invocação dos dois métodos em um trecho qualquer da aplicação:

```
trapper.sendVars("abc", 123, 456789L, true);

trapper.sendVars(new Date(), new Integer(234), new Long(567890L), Boolean.FALSE);
```

O resultado final enviado ao servidor SNMP será o seguinte:

```
2011-09-15 11:30:09 0.0.0.0(via UDP: [127.0.0.1]:-7159) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.2 Enterprise Specific Trap (0)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.2.1.0 = STRING: "abc"
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.2.2.0 = INTEGER: 123
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.2.3.0 = INTEGER: 456789
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.2.4.0 = STRING: "true"

2011-09-15 11:30:14 0.0.0.0(via UDP: [127.0.0.1]:-28941) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.4 Enterprise Specific Trap (0)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.4.1.0 = STRING: "Thu Sep 15 11:30:14 2011"
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.4.2.0 = INTEGER: 234
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.4.3.0 = INTEGER: 567890
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.4.4.0 = STRING: "false"
```

Ou seja, cada um dos argumentos dos métodos é transformado em uma variável na mensagem da trap SNMP. Note que o OID das variáveis é sequencial e automático, utilizando o prefixo definido nas anotações @MIB e @OID e finalizando com ". 0", pois são escalares.



Nota

Quando o tipo de dados do Java não tem relação direta com um tipo do SNMP, é utilizado *Octect String*, ou seja, é considerado o formato textual da variável. Ex: `boolean`, `Date` ou `Object` são todos convertidos para `STRING`.

16.6. Especificando os tipos SNMP das variáveis

Nas seções anteriores todos os exemplos que vimos utilizavam somente os tipos de dados `INTEGER` ou `STRING` para as variáveis. Todavia, a especificação SNMP prevê diversos tipos de dados específicos que fornecem alguma semântica. O *Demoiselle Monitoring* fornece suporte para a definição destes tipos específicos, a qual veremos a seguir.



Dica

Para maiores informações sobre os tipos de dados do SNMP, consulte as especificações [RFC-1155](http://tools.ietf.org/html/rfc1155) [http://tools.ietf.org/html/rfc1155] e [RFC-2578](http://tools.ietf.org/html/rfc2578) [http://tools.ietf.org/html/rfc2578].

Para definir um tipo de dado específico do SNMP, basta declarar o argumento do método precedido da anotação correspondente ao tipo (ex: @Counter32, @Gauge32 ou @Integer32). A lista completa de tipos de dados SNMP e as anotações providas pelo *Demoiselle Monitoring* podem ser conferidas em [Tipos de dados SNMP](#).

Veja no exemplo a seguir uma classe trapper que possui o método `sendNumbers()` com cinco argumentos do tipo `int`. Para que sejam considerados mais do que apenas `INTEGER`, os parâmetros foram devidamente anotados com os tipos numéricos. Na mesma classe, o método `sendObjects()` faz uso de tipos avançados, como `IP Address` e `OID`.

```
@Trapper
@SNMP
@MIB(".1.1")
public class MyTrapperSNMP implements MyTrapper {

    @OID(".6")
    public void sendNumbers(int a, @Integer32 int b,
        @Counter32 int c, @Gauge32 int d, @TimeTicks int e) { }

    @OID(".8")
    public void sendObjects(@IPAddress String ip,
        @ObjectIdentifier String oid, @OctetString String str) { }

}
```



Dica

As anotações de tipos de dados SNMP fornecidas pelo componente estão no pacote `br.gov.frameworkdemoiselle.monitoring.annotation.snmp.type`.

Eis um exemplo de invocação dos dois métodos da trapper em qualquer trecho da aplicação:

```
trapper.sendNumbers(100, 200, 300, 400, 500);

trapper.sendObjects("192.168.0.1", "1.3.6.1.4.1", "um texto");
```

O resultado recebido por um servidor SNMP será o seguinte:

```
2011-09-15 15:12:45 0.0.0.0(via UDP: [127.0.0.1]:-20608) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.6 Enterprise Specific Trap (0)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.6.1.0 = INTEGER: 100
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.6.2.0 = INTEGER: 200
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.6.3.0 = Counter32: 300
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.6.4.0 = Gauge32: 400
```



```
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.6.5.0 = Timeticks: (500) 0:00:05.00

2011-09-15 15:12:50 0.0.0.0(via UDP: [127.0.0.1]:-29915) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.8 Enterprise Specific Trap (0)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.8.1.0 = IPAddress: 192.168.0.1
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.8.2.0 = OID: SNMPv2-SMI::enterprises
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.8.3.0 = STRING: "um texto"
```

Perceba que a grande diferença é que os seguintes tipos de dados SNMP, os quais carregam significado, são transmitidos na mensagem SNMP: Counter32, Gauge32, TimeTicks, IPAddress e OID.

16.7. Usando o trapper SNMP simples

Para que o mecanismo de monitoração seja rapidamente implantado em uma aplicação, o *Demoiselle Monitoring* fornece os *Simple Trappers*. Tratam-se de classes do tipo trapper prontas para serem utilizadas pelo desenvolvedor, ou seja, sem a necessidade de se programar trappers customizadas.

Basicamente o programador fará uso da interface `SimpleTrapper`, presente no pacote `br.gov.frameworkdemoiselle.monitoring.trapping`:

```
public interface SimpleTrapper {
    void send(String key, String message);
    void send(String message);
}
```



Nota

O método `send()` da interface `SimpleTrapper` é sobrecarregado: aquele que possui os parâmetros `key` e `message` recebe a chave (i.e., a OID da trap) em tempo de execução. Já o método que possui apenas o parâmetro `message` faz uso de uma chave default armazenada nas configurações da aplicação.

Junto com a interface é disponibilizada a respectiva implementação `SimpleTrapperSNMP`, destinada ao envio de traps SNMP.

Para utilizar essa funcionalidade na aplicação, basta declarar uma variável do tipo `SimpleTrapper` e usar a anotação `@Inject` com o qualificador `@SNMP`. Em seguida, qualquer um dos métodos `SimpleTrapper.send()` podem ser invocados na classe em questão.

Eis um exemplo de código:

```
public class SimpleTrapperTest {

    @Inject
    @SNMP
    private SimpleTrapper snmp;

    public void simpleTrappingSpecifiedKey() {
        snmp.send("1.3.6.1.4.1.35437.1.1.2425.2011.1.1", "You say yes. I say no!");
    }
}
```

```
public void simpleTrappingDefaultKey() {  
    snmp.send("Let it be. Let it be!");  
}  
  
}
```

O método de exemplo `simpleTrappingSpecifiedKey()` envia a trap SNMP considerando a OID passada no seu primeiro argumento (i.e., 1.3.6.1.4.1.35437.1.1.2425.2011.1.1). Já o método `simpleTrappingDefaultKey()` utiliza a OID padrão configurada no arquivo `demoiselle.properties`:

```
frameworkdemoiselle.monitoring.snmp.trapper.enterprise = 1.3.6.1.4.1.35437.1.1.2425.2011
```

O resultado da execução do código de exemplo pode ser observado na LOG do servidor SNMP:

```
2011-09-16 11:13:10 0.0.0.0(via UDP: [127.0.0.1]:-12961) TRAP, SNMP v1, community public  
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1 Enterprise Specific Trap (0)  
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.1.1.0 = STRING: "You say yes. I say no!"  
  
2011-09-16 11:13:15 0.0.0.0(via UDP: [127.0.0.1]:-23825) TRAP, SNMP v1, community public  
SNMPv2-SMI::enterprises.35437.1.1.2425.2011 Enterprise Specific Trap (0)  
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.1.0 = STRING: "Let it be. Let it be!"
```

Monitorando via Zabbix (polling)

A abordagem específica Zabbix no *Demoiselle Monitoring* envolve a criação e disponibilização de componentes básicos em Java, os MBeans, tal como na monitoração com SNMP. No próprio servidor Java será levantado um *Agente Zabbix* na porta 10052 em TCP.



Nota

O *Agente Zabbix* não se trata de um agente Zabbix convencional, como o daemon *zabbix_agentd* [http://www.zabbix.com/documentation/1.8/manual/processes/zabbix_agentd]. É apenas um programa que entende o protocolo do Zabbix e responde às requisições enviadas pelo servidor.

17.1. Configuração do Agente Zabbix

Para habilitar o *Agente Zabbix* e definir seus parâmetros, altere o arquivo `demoiselle.properties` incluindo as linhas a seguir:

```
frameworkdemoiselle.monitoring.zabbix.agent.enabled = true
frameworkdemoiselle.monitoring.zabbix.agent.port = 10052
frameworkdemoiselle.monitoring.zabbix.agent.address = *
frameworkdemoiselle.monitoring.zabbix.agent.protocol = 1.4
```

Tendo definidos esses parâmetros, ao executar a aplicação o *Agente Zabbix* será automaticamente inicializado.

17.2. Criação do MBean para o Zabbix

Uma vez devidamente configurado o *Agente Zabbix*, os indicadores do host monitorado a serem disponibilizados por ele devem ser criadas pelo desenvolvedor. Veremos como fazer isso nessa seção. Primeiramente é preciso criar uma interface simples contendo os métodos a serem expostos, tal como no exemplo:

```
public interface ExemploMBean {

    String getAtributo();

}
```

Em seguida, uma classe anotada com o estereótipo `@MBean` deve implementar os métodos dessa interface:

```
@MBean
public class Exemplo implements ExemploMBean {

    public String getAtributo() {
        return "valor do atributo";
    }

}
```

}



Importante

Lembre-se de que a classe deve implementar uma interface de mesmo nome acrescida do sufixo “MBean”.



Dica

A anotação `@MBean` é fornecida pelo componente e encontra-se no pacote `br.gov.frameworkdemoiselle.monitoring.stereotype`.

As regras e recomendações para a criação de um MBean para o Agente Zabbix são exatamente as mesmas aplicadas na seção [Monitorando via JMX](#).

17.3. Consultando os indicadores do Agente Zabbix

Tendo o *Agente Zabbix* levantado e MBeans devidamente exportados no servidor Java, podemos consultar os indicadores do host através do protocolo Zabbix. Na prática isso será feito pelo servidor Zabbix (especificamente pelo daemon [zabbix_server](#) [http://www.zabbix.com/documentation/1.8/manual/processes/zabbix_server]). Entretanto, podemos utilizar o comando `telnet`¹ para testar o funcionamento do agente e obter o valor do indicador no host. Veja:

Primeiro execute a instrução `telnet localhost 10052`. Em seguida digite a linha `jmx[br.gov.frameworkdemoiselle.monitoring.mbean:name=Exemplo][Atributo]` e pressione **ENTER**. A comunicação será encerrada, porém o valor do atributo `Atributo` do MBean de nome `Exemplo` será exibido no terminal. Veja:

```
$ telnet localhost 10052
Trying ::1...
Connected to localhost.
Escape character is '^]'.
jmx[br.gov.frameworkdemoiselle.monitoring.mbean:name=Exemplo][Atributo]
ZBXD^Avalor do atributoConnection closed by foreign host.
```

O protocolo do Zabbix permite que múltiplas consultas a indicadores sejam efetuadas numa mesma requisição. A string “ZBXD” faz parte do protocolo.



Dica

No *Debian GNU/Linux* é disponibilizado o pacote `telnet` (<http://packages.debian.org/telnet>), o qual fornece o comando de mesmo nome a ser usado para se comunicar com outro host através do protocolo TELNET. Para instalar esse pacote, basta executar `apt-get install telnet` como super-usuário.

¹ Para maiores informações sobre o `telnet`, acesse: <http://en.wikipedia.org/wiki/Telnet>

Quando o *Agente Zabbix* recebe uma requisição vinda de um host com o formato “jmx[nome-do-mbean][atributo]”, ele busca dentro da mesma máquina virtual (JVM) um MBean pré-registrado no respectivo contêiner JMX com o nome especificado. Se ele o encontrar, fará a requisição JMX do atributo informado, provocando desta maneira a execução do seu respectivo método *getter*.

Ao configurar o host monitorado na interface do Zabbix, utilize o formato citado acima no campo *Key* do indicador (i.e., *item*).



Importante

Para o *Agente Zabbix* um indicador de host (i.e., *host item*) é internamente um atributo de MBean registrado na mesma JVM em que ele roda. O agente fornece a consulta a esse atributo por meio do protocolo Zabbix obedecendo ao seguinte formato: “jmx[nome-do-mbean][atributo]”.

17.4. Uma monitoração mais complexa

Veremos agora um caso prático, onde uma aplicação de gestão escolar precisa fornecer monitoração sobre indicadores de negócio e de sistema. A proposta é fornecer diversos indicadores de negócio da aplicação através do *Agente Zabbix* provido pelo componente *Demoiselle Monitoring*. Posteriormente esses mesmos indicadores (i.e., *Items*) poderão ser configurados para um determinado dispositivo (i.e., *Host*) no servidor Zabbix.

O primeiro passo é criar a interface em Java contendo todos os métodos referentes aos indicadores a serem exportados. Lembre-se de usar o sufixo “MBean”:

```
public interface EscolaMonitoringMBean {

    String getVersaoAplicacao();

    int getQtdTurmasIncluidas();

    long getQtdAlunosMatriculados();

    String getUltimoUsuarioLogado();

    int getNivelLog();

    void setNivelLog(int nivelLog);

}
```

Em seguida, crie a classe que implementa a interface anterior. É necessário que ela seja anotada com `@MBean`. Veja:

```
@MBean
@Name("br.gov.frameworkdemoiselle:name=Escola")
public class EscolaMonitoring implements EscolaMonitoringMBean {

    private static final String VERSAO = "2.4.1-BETA";
    private static final String[] USUARIOS = { "Fulano", "Sicrano", "Beltrano" };

    private int qtdTurmas = 0;
```

```

private long qtdAlunos = 0;

@Override
public String getVersaoAplicacao() {
    return VERSAO;
}

@Override
public int getQtdTurmasIncluidas() {
    this.qtdTurmas += (int) (Math.random() * 10);
    return this.qtdTurmas;
}

@Override
public long getQtdAlunosMatriculados() {
    this.qtdAlunos = (int) (Math.random() * 100) + 100;
    return this.qtdAlunos;
}

@Override
public String getUltimoUsuarioLogado() {
    int pos = (int) (Math.random() * USUARIOS.length);
    return USUARIOS[pos];
}

private int nivelLog = 1;

@Override
public int getNivelLog() {
    return this.nivelLog;
}

@Override
public void setNivelLog(int nivelLog) {
    this.nivelLog = nivelLog;
}
}

```

Neste caso de exemplo, a classe `EscolaMonitoring` fará com que cinco indicadores sejam disponibilizados via JMX, possibilitando que eles sejam consultados pelo servidor Zabbix. O indicador `VersaoAplicacao` retornará sempre o mesmo valor de *string*. Já `QtdTurmasIncluidas` e `QtdAlunosMatriculados` retornarão números inteiros que serão incrementados aleatoriamente a cada chamada. O indicador `UltimoUsuarioLogado` retorna randomicamente um dos três nomes possíveis de usuário. E `NivelLog` armazena um número que pode ser apenas consultado pelo Zabbix.



Nota

Na abordagem com o Zabbix os valores dos indicadores somente podem ser consultados, ao contrário do SNMP, que fornece o comando de escrita `SET`.

Para testar a classe `EscolaMonitoring` e o *Agente Zabbix*, execute a aplicação e abra uma sessão TCP com o comando `telnet localhost 10052`. Em seguida, cole as linhas a seguir e pressione **ENTER**:

```
jmx[br.gov.frameworkdemoiselle:name=Escola][VersaoAplicacao]
jmx[br.gov.frameworkdemoiselle:name=Escola][QtdAlunosMatriculados]
jmx[br.gov.frameworkdemoiselle:name=Escola][QtdTurmasIncluidas]
jmx[br.gov.frameworkdemoiselle:name=Escola][UltimoUsuarioLogado]
jmx[br.gov.frameworkdemoiselle:name=Escola][NivelLog]
```

A sessão TELNET será finalizada. Note que chamadas consecutivas ao serviço retornarão valores distintos para alguns dos indicadores, tal como esperado:

```
$ telnet localhost 10052
Trying ::1...
Connected to localhost.
Escape character is '^'.
jmx[br.gov.frameworkdemoiselle:name=Escola][VersaoAplicacao]
jmx[br.gov.frameworkdemoiselle:name=Escola][QtdAlunosMatriculados]
jmx[br.gov.frameworkdemoiselle:name=Escola][QtdTurmasIncluidas]
jmx[br.gov.frameworkdemoiselle:name=Escola][UltimoUsuarioLogado]
jmx[br.gov.frameworkdemoiselle:name=Escola][NivelLog]
ZBXD^A2.4.1-RC2ZBXD^A^C191ZBXD^A^A8ZBXDBeltranoZBXD^A^AlConnection closed by foreign host.
```

```
$ telnet localhost 10052
Trying ::1...
Connected to localhost.
Escape character is '^'.
jmx[br.gov.frameworkdemoiselle:name=Escola][VersaoAplicacao]
jmx[br.gov.frameworkdemoiselle:name=Escola][QtdAlunosMatriculados]
jmx[br.gov.frameworkdemoiselle:name=Escola][QtdTurmasIncluidas]
jmx[br.gov.frameworkdemoiselle:name=Escola][UltimoUsuarioLogado]
jmx[br.gov.frameworkdemoiselle:name=Escola][NivelLog]
ZBXD^A2.4.1-RC2ZBXD^A^C122ZBXD^A^B13ZBXD^ASicranoZBXD^A^AlConnection closed by foreign host.
```

Os valores dos indicadores consultados via protocolo Zabbix ² são retornados de uma só vez, contendo a string “ZBXD” como separador, além de caracteres especiais de controle.

17.5. Configurando a monitoração polling no servidor Zabbix

Uma vez criada a classe MBean de monitoração, tendo o Agente Zabbix e a aplicação Java em execução, podemos efetivamente monitorar os indicadores no servidor Zabbix.


Crie o *Host* no Zabbix (no exemplo foi chamado “Armenia Server”), e em seguida crie os *Items* para ele. O tipo do indicador (campo *Type*) deve ser “Zabbix agent” e a chave (campo *Key*) deve corresponder ao formato “jmx[nome-do-mbean][atributo]”. Veja:

Figura 17.1. Configuração de indicador no Zabbix

² Para maiores informações sobre o protocolo Zabbix para agentes, acesse o link <http://www.zabbix.com/wiki/doc/tech/proto/zabbixagentprotocol>.

No exemplo, todos os indicadores contidos na classe `EscolaMonitoring` foram incluídos para monitoração pelo servidor Zabbix. Note que eles possuem intervalos de requisição distintos. Por exemplo, a versão da aplicação (`VersaoAplicacao`) é obtida a cada 8640 segundos, ou seja, uma vez ao dia. Já `QtdAlunosMatriculados` e `QtdTurmasIncluidas` são consultadas a cada minuto. Veja:

Figura 17.2. Configuração de indicadores do host no Zabbix



Importante

Na tela de configuração do host, é preciso definir a porta TCP correspondente ao *Agente Zabbix* fornecido pelo componente *Demoiselle Monitoring*, que deve diferir do default 10050.

Assim que os indicadores começam a ser monitorados e armazenados pelo servidor Zabbix, podemos criar gráficos customizados usando esses valores coletados. Veja um exemplo de plotagem usando os indicadores `QtdAlunosMatriculados` e `QtdTurmasIncluidas` da classe `EscolaMonitoring`:

Figura 17.3. Geração de gráficos customizados no Zabbix

Monitorando via Zabbix (trapping)

O componente *Demoiselle Monitoring* fornece um mecanismo simples e transparente para uma aplicação Java enviar mensagens a um servidor Zabbix. Trata-se de um código escrito inteiramente em linguagem Java que faz o papel do utilitário `zabbix_sender`¹.

O envio de mensagens a um servidor Zabbix é muito simples, basta informar três dados ao servidor:

- *host*: pré-cadastrado no Zabbix como nome de Host
- *chave*: pré-cadastrada no Zabbix em um indicador do host (i.e., campo *Key* de um *Item*)
- *valor*: a mensagem a ser transmitida, podendo ser um texto, número ou valor lógico (booleano)

Veremos nas próximas seções de que forma o componente *Demoiselle Monitoring* pode auxiliar o desenvolvedor a preparar uma aplicação Java para ser monitorada via polling por um servidor Zabbix.

18.1. Configuração para os trappers Zabbix

Alguns parâmetros devem ser configurados na aplicação para o envio de mensagens ao servidor Zabbix pelo componente *Demoiselle Monitoring*. Tais configurações são feitas no arquivo `demoiselle.properties`:

```
frameworkdemoiselle.monitoring.zabbix.trapper.server = localhost
frameworkdemoiselle.monitoring.zabbix.trapper.port = 10051
frameworkdemoiselle.monitoring.zabbix.trapper.host = Armenia Server
frameworkdemoiselle.monitoring.zabbix.trapper.default_key = app.message
```

Esses parâmetros definem o endereço (*server*) e a porta (*port*) de comunicação com o servidor Zabbix que receberá as mensagens, além de especificar o nome do host (*host*) e uma chave padrão (*default_key*) para estas.

18.2. Criação e utilização do Trapper Zabbix

Ao utilizar o *Demoiselle Monitoring*, o desenvolvedor poderá implementar facilmente suas próprias *trappers*. Para isso, ele pode inicialmente criar uma interface contendo os métodos que enviarão as mensagens. Eis um exemplo:

```
public interface MyTrapper {

    void sendFailure(String message);

}
```

Em seguida ele criará a respectiva classe, anotando com o estereótipo `@Trapper` e o qualificador `@Zabbix` (ficará implícito que se trata de um “trapper para Zabbix”).

Veja como é simples:

¹ Para maiores informações sobre o `zabbix_sender`, acesse: http://www.zabbix.com/documentation/1.8/manual/processes/zabbix_sender

```
@Trapper
@Zabbix
public class MyTrapperZabbix implements MyTrapper {

    public void sendFailure(String message) { }

}
```



Nota

Lembre-se de que criar uma interface para o trapper e em seguida implementá-la não é obrigatório, mas apenas um direcionamento. Se preferir, você pode construir a classe trapper diretamente (isto é, sem usar o `implements`). Entretanto, a aplicação será mais flexível ao considerar o uso de interfaces na arquitetura de seus componentes.



Importante

Note que os métodos da classe trapper não precisam ser codificados, ou seja, podem ficar vazios. Isso porque neles atuarão interceptadores do CDI. Na versão 1.x do componente isso era realizado através de aspectos com o AspectJ.

Finalmente, para utilizar esse trapper em qualquer parte da aplicação, basta declarar um campo com o tipo da interface e anotá-lo com `@Inject`:

```
public class TrapperTest {

    @Inject
    @Zabbix
    private MyTrapper trapper;

    public void sendFailureMessage() {
        trapper.sendFailure("Hello, zabbix...");
    }

}
```

Uma vez invocado o método `sendFailure()`, o interceptador agirá sobre ele e fará o envio, de forma assíncrona, da mensagem ao servidor Zabbix na forma de uma mensagem.

Neste caso específico, a mensagem com o texto "Hello, zabbix..." chegará ao servidor Zabbix direcionada ao host de nome "Armenia Server" e usando o indicador "app.message". Isso porque estes parâmetros foram definidos no arquivo `demoiselle.properties`. Veremos na seção seguinte como fazer para sobrescrevê-los.

Ao ser executado o método `TrapperTest.sendFailureMessage()`, será transmitida a mensagem ao servidor Zabbix tal como se fosse invocado o binário `zabbix_sender` dessa maneira:

```
$ zabbix_sender -s "Armenia Server" -k app.message -o "Hello, zabbix..."
```

Uma sugestão é utilizar esse método em blocos protegidos com `try..catch` no Java. Se uma determinada exceção for levantada, o trapper pode ser invocado para enviar os detalhes para o servidor Zabbix. Veja:

```
try {
    throw new Exception("Um erro ocorreu intencionalmente");
} catch (Exception e) {
    trapper.sendFailure(e.getMessage());
}
```

18.3. Definindo o host e a chave do indicador

Embora tenhamos usado os valores padrões no primeiro exemplo, as informações de nome de host e chave do indicador podem ser sobrescritas através de anotações no trapper. A sobrescrita das chaves é particularmente importante pois permite a definição de múltiplos indicadores monitoráveis (chamados de *Items* no Zabbix) para um mesmo host.

Para sobrescrever o nome do host especificado no arquivo `demoiselle.properties` é preciso utilizar a anotação `@HostName` na declaração da classe trapper.

Na prática, cada método da classe trapper em Java será responsável por um indicador monitorável no Zabbix. Para definir a chave de um indicador, simplesmente anote o método com `@ItemKey`.

Veja como fica uma classe trapper com múltiplos indicadores e com o nome de host especificado:

```
@Trapper
@Zabbix
@HostName("Marumbi Server")
public class MyZabbixTrapper implements IMyTrapper {

    @ItemKey("app.failure")
    @Override
    public void sendFailure(String message) { }

    @ItemKey("app.database")
    @Override
    public void sendDatabaseInfo(int connections) { }

}
```

Veja alguns exemplos de invocação desses métodos do trapper na aplicação:

```
trapper.sendFailure("webservice fora do ar");

trapper.sendDatabaseInfo(50);
```

Eles serão equivalentes à execução desses comandos:

```
$ zabbix_sender -s "Marumbi Server" -k app.failure -o "webservice fora do ar"

$ zabbix_sender -s "Marumbi Server" -k app.database -o 50
```

Dessa forma, é preciso que o host de nome “Marumbi Server” seja criado no servidor Zabbix. Além disso, esse host deve possuir indicadores com as chaves `app.failure` e `app.database`.



Importante

Se o nome de host e a chave de indicador monitorável informada na mensagem não estiverem previamente cadastrados no Zabbix, a mensagem é ignorada pelo servidor.

18.4. Usando parâmetros curingas nas chaves

Imagine uma situação em que múltiplos indicadores similares precisam ser criados para um mesmo host monitorado pelo Zabbix. Ao utilizar o *Demoiselle Monitoring*, isso significaria criar múltiplos métodos distintos para a classe `trapper`. Por exemplo: enviar a quantidade de conexões em cada um dos três bancos de dados utilizados pela aplicação.

Para facilitar esse tipo de situação, é possível utilizar curingas na string que define as chaves de um indicador na anotação `@ItemKey`. Para ilustrar, considere alterar o método `sendDatabaseInfo()` da classe `MyZabbixTrapper`:

```
@ItemKey("app.database[*]")
public void sendDatabaseInfo(String datasource, int connections) { }
```

Dessa forma, o curinga (representado pelo caracter “*” na string) será substituído pelo valor passado no primeiro argumento do método, `datasource`. Veja alguns exemplos de invocação desse método:

```
trapper.sendDatabaseInfo("db1", 100);

trapper.sendDatabaseInfo("db2", 150);

trapper.sendDatabaseInfo("db3", 200);
```

O *Demoiselle Monitoring* fará a substituição automática das chaves no momento do envio da mensagem. Neste exemplo, o servidor receberá os valores 100, 150 e 200 respectivamente para os indicadores `app.database[db1]`, `app.database[db2]` e `app.database[db3]`. Note que ao usar o curinga, a string da chave pode ser definida de um modo mais flexível e poderoso.



Importante

No servidor Zabbix é preciso cadastrar individualmente cada possível variação de chave para os indicadores. Ou seja, o curinga é um facilitador para organizar o código da aplicação Java.

Podem ser especificados diversos curingas em uma mesma string de chave, sempre usando o caracter "*" para isso. Neste caso, eles serão substituídos pelos argumentos do método na ordem em que forem declarados.

18.5. Consultando MBeans e usando a anotação @JMXQuery

A aplicação Java pode consultar valores de objetos registrados em MBeans no próprio servidor JMX da Máquina Virtual (JVM) em que ela estiver sendo executada. Para isso, o *Demoiselle Monitoring* fornece a classe utilitária `MBeanHelper`. Com o método estático `Object query(String name, String attribute)` dessa classe, basta informar o nome do MBean e o atributo desejado.

Eis alguns exemplos de consulta aos parâmetros do sistema operacional e da Máquina Virtual do Java (JVM) através de JMX com o método `MBeanHelper.query()`:

```
// nome e versao do sistema operacional
String name = (String) MBeanHelper.query("java.lang:type=OperatingSystem", "Name");
String version = (String) MBeanHelper.query("java.lang:type=OperatingSystem", "Version");

// carga media do sistema (load avg)
Double load = (Double) MBeanHelper.query("java.lang:type=OperatingSystem", "SystemLoadAverage");

// nome e versao da JVM
String name = (String) MBeanHelper.query("java.lang:type=Runtime", "VmName");
String version = (String) MBeanHelper.query("java.lang:type=Runtime", "VmVersion");

// dados do class loader
Integer loaded = (Integer) MBeanHelper.query("java.lang:type=ClassLoading", "LoadedClassCount");
Long unloaded = (Long) MBeanHelper.query("java.lang:type=ClassLoading", "UnloadedClassCount");

// valor de um MBean customizado
String atributo = (String) MBeanHelper.query(
    "br.gov.frameworkdemoiselle.monitoring.mbean:name=Exemplo", "Atributo");
```



Dica

O método `MBeanHelper.query()` de consulta a dados da JMX pode ser usado em qualquer parte da aplicação, e em especial na criação de MBeans e trappers para o Zabbix.

No caso específico de trappers do tipo Zabbix, o *Demoiselle Monitoring* oferece um outro recurso: a anotação `@JMXQuery`. Com ela, é possível enviar o resultado de uma consulta JMX automaticamente na invocação de um método do trapper. Para usar esse recurso, simplesmente anote o método do trapper com `@JMXQuery`, especificando os argumentos `mbeanName` e `mbeanAttribute` com respectivamente o nome do MBean e o atributo a ser consultado.

Veja um exemplo de utilização da anotação `@JMXQuery`:

```
@Trapper
@Zabbix
public class MyZabbixTrapper implements IMyTrapper {
```

```

@ItemKey("java.memory[used]")
@JMXQuery(mbeanName = "java.lang:type=Memory", mbeanAttribute = "HeapMemoryUsage.used")
public void sendUsedHeapMemory() { }
}

```

Ou seja, basta invocar o método sem argumentos `MyZabbixTrapper.sendUsedHeapMemory()` para que o valor obtido via JMX com a string `"java.lang:type=Memory[HeapMemoryUsage.used]"` seja enviado ao servidor Zabbix para o indicador com a chave `"java.memory[used]"`.



Nota

Qualquer MBean registrado no servidor JMX da mesma Máquina Virtual em que está rodando a aplicação pode ser consultado com o método `MBeanHelper.query()` ou com a anotação `@JMXQuery`. Isso inclui particularmente os MBeans customizados da própria aplicação.

18.6. Configurando a monitoração trapping no servidor Zabbix

Uma vez criada a classe trapper e tendo a aplicação Java em execução, podemos efetivamente monitorar os indicadores no servidor Zabbix via trapping.

Para exemplificar esse processo, podemos criar a interface `ApplicationTrapper` conforme código a seguir:

```

public interface ApplicationTrapper {

    void sendQtdTurmasIncluidas(int qtd);

    void sendFailure(String message);

    void sendDatabaseInfo(String datasource, int connections);

    void sendSystemLoadAverage();

}

```

Uma classe do tipo trapper chamada `ApplicationTrapperZabbix` implementa a interface anterior. Note que as diversas abordagens para definição de envio de mensagens via trapper são consideradas neste exemplo:

```

@Trapper
@Zabbix
@HostName("Marumbi Server")
public class ApplicationTrapperZabbix implements ApplicationTrapper {

    @ItemKey("escola[turmas]")
    @Override
    public void sendQtdTurmasIncluidas(final int qtd) { }

    @ItemKey("app.failure")
    @Override
}

```

```

public void sendFailure(final String message) { }

@ItemKey("app.database[*]")
@Override
public void sendDatabaseInfo(final String datasource, final int connections) { }

@ItemKey("sys.ldavg")
@JMXQuery(mbeanName = "java.lang:type=OperatingSystem", mbeanAttribute = "SystemLoadAverage")
@Override
public void sendSystemLoadAverage() { }
}

```

Desta forma a aplicação estará pronta para enviar traps para o servidor Zabbix. Veremos agora como configurar o Zabbix para receber essas mensagens do host.

Crie o *Host* no Zabbix (no exemplo foi chamado “Marumbi Server”), e em seguida crie os *Items* para ele. O tipo do indicador (campo *Type*) deve ser “Zabbix trapper” e a chave (campo *Key*) deve corresponder à string definida na anotação `@ItemKey`. Veja:

Figura 18.1. Configuração de indicador no Zabbix

No exemplo, todos os métodos contidos na classe `ApplicationTrapper` foram incluídos para monitoração via trapping no servidor Zabbix. Veja:

Figura 18.2. Configuração de indicadores do host no Zabbix

Inclua as seguintes instruções em algum ponto da aplicação Java e as execute:

```

trapper.sendQtdTurmasIncluidas(210);

trapper.sendFailure("Um erro ocorreu intencionalmente");

trapper.sendDatabaseInfo("db1", 20);
trapper.sendDatabaseInfo("db2", 40);
trapper.sendDatabaseInfo("db3", 60);

trapper.sendSystemLoadAverage();

```

Em seguida confira no Zabbix os últimos dados recebidos para o host “Marumbi Server”:

Figura 18.3. Últimos dados recebidos para o host no Zabbix

18.7. Usando o trapper Zabbix simples

Para que o mecanismo de monitoração seja rapidamente implantado em uma aplicação, o *Demoiselle Monitoring* fornece os *Simple Trappers*. Tratam-se de classes do tipo trapper prontas para serem utilizadas pelo desenvolvedor, ou seja, sem a necessidade de se programar trappers customizadas.

Basicamente o programador fará uso da interface `SimpleTrapper`, presente no pacote `br.gov.frameworkdemoiselle.monitoring.trapping`:

```
public interface SimpleTrapper {  
    void send(String key, String message);  
    void send(String message);  
}
```



Nota

O método `send()` da interface `SimpleTrapper` é sobrecarregado: aquele que possui os parâmetros `key` e `message` recebe a chave do indicador monitorado em tempo de execução. Já o método que possui apenas o parâmetro `message` faz uso de uma chave default armazenada nas configurações da aplicação.

Junto com a interface é disponibilizada a respectiva implementação `SimpleTrapperZabbix`, destinada ao envio de traps Zabbix.

Para utilizar essa funcionalidade na aplicação, basta declarar uma variável do tipo `SimpleTrapper` e usar a anotação `@Inject` com o qualificador `@Zabbix`. Em seguida, qualquer um dos métodos `SimpleTrapper.send()` podem ser invocados na classe em questão.

Eis um exemplo de código:

```
public class SimpleTrapperTest {  
  
    @Inject  
    @Zabbix  
    private SimpleTrapper zabbix;  
  
    public void simpleTrappingSpecifiedKey() {  
        zabbix.send("app.quote", "You say yes. I say no!");  
    }  
  
    public void simpleTrappingDefaultKey() {  
        zabbix.send("Let it be. Let it be!");  
    }  
}
```

O método de exemplo `simpleTrappingSpecifiedKey()` envia a trap Zabbix considerando a chave do indicador passada no seu primeiro argumento (i.e., `app.quote`). Já o método `simpleTrappingDefaultKey()` utiliza a chave padrão configurada no arquivo `demoiselle.properties`:

```
frameworkdemoiselle.monitoring.zabbix.trapper.default_key = app.message
```

Monitorando com múltiplas abordagens

Nos capítulos anteriores foram apresentadas três abordagens diferentes para a monitoração de aplicações Java com o auxílio do componente *Demoiselle Monitoring*: JMX, SNMP e Zabbix. Nas seções seguintes veremos que, embora geralmente iremos escolher apenas uma delas, temos a possibilidade de utilizá-las em conjunto na mesma aplicação.

19.1. Um trapper multiuso: SNMP e Zabbix

Ao construir uma classe trapper, o programador pode fazer uso de ambas as abordagens SNMP e Zabbix com o mesmo código. Esta será efetivamente escolhida no momento da injeção. Considere como exemplo a interface `SystemTrapper`:

```
public interface SystemTrapper {  
  
    void sendAvailability(String version, long uptime);  
  
}
```

A classe do tipo trapper `SystemTrapperSNMPZabbix` abaixo a estará preparada para ser usada tanto com SNMP ou Zabbix:

```
@Trapper  
@SNMP  
@MIB(".3.3")  
@Zabbix  
@HostName("scel5was")  
public class SystemTrapperSNMPZabbix implements SystemTrapper {  
  
    @OID(".1")  
    @SpecificTrap(5)  
    @ItemKey("app[*].uptime")  
    public void sendAvailability(final String version, final @Counter32 long uptime) { }  
  
}
```



Nota

As anotações `@SNMP`, `@MIB`, `@OID`, `@SpecificTrap` e `@Counter32` neste exemplo referem-se à abordagem SNMP. Já as anotações `@Zabbix`, `@HostName` e `@ItemKey` são próprias da abordagem Zabbix. Todavia, elas agem de modo independente e por isso podem ser usadas em declarações na mesma classe.

Eis um exemplo de emprego da classe `SystemTrapperSNMPZabbix` para enviar traps a um servidor SNMP:

```
public class SystemTrapperSNMPTest {

    @Inject
    @SNMP
    private SystemTrapper snmp;

    public void sendTrap() {
        snmp.sendAvailability("2.1.3", 14400);
    }

}
```

Para usar a mesma classe `SystemTrapperSNMPZabbix` para enviar traps a um servidor Zabbix, eis um exemplo:

```
public class SystemTrapperZabbixTest {

    @Inject
    @Zabbix
    private SystemTrapper zabbix;

    public void sendTrap() {
        zabbix.sendAvailability("2.1.3", 14400);
    }

}
```

Existe ainda a possibilidade de a aplicação disparar traps para ambos os servidores SNMP e Zabbix com uma mesma classe trapper:

```
public class SystemTrapperMultiTest {

    @Inject
    @SNMP
    @Zabbix
    private SystemTrapper trapper;

    public void sendTrap() {
        trapper.sendAvailability("2.1.3", 14400);
    }

}
```



Importante

A escolha sobre qual abordagem utilizar é feita no momento da injeção da classe trapper, ou seja, se foi utilizado `@SNMP` e ou `@Zabbix` em conjunto com `@Inject`.

Eis a mensagem enviada ao servidor SNMP com o código acima:

```
2011-08-25 15:49:46 0.0.0.0(via UDP: [127.0.0.1]:36822) TRAP, SNMP v1, community public
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.3.3.1 Enterprise Specific Trap (5)
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.3.3.1.1.0 = STRING: "2.1.3"
SNMPv2-SMI::enterprises.35437.1.1.2425.2011.3.3.1.2.0 = Counter32: 14400
```

E a mensagem enviada ao servidor Zabbix na mesma invocação:

```
INFO [zabbix.Sender] Sending message: [host=scel5was, key=app[2.1.3].uptime, value=14400]
INFO [zabbix.Sender] Actual stream sent:
  <req>
    <host>c2NlbDV3YXM= </host>
    <key>YXBWZiUuMS4zXS5lcHRpbWU= </key>
    <data>MTQ0MDA= </data>
  </req>
```

Fazendo uso dos Checkers

No componente *Demoiselle Monitoring* os trappers se destinam apenas a enviar as mensagens aos respectivos servidores assim que algum de seus métodos é invocado por alguma parte da aplicação Java. Ou seja, as classes estereotipadas com `@Trapper` não realizam o envio automático e periódico dos indicadores.



Importante

Os checkers são particularmente essenciais para o envio de informações em intervalos regulares do host monitorado ao servidor. As classes trappers somente enviam mensagens quando um de seus métodos é invocado pela aplicação.

20.1. Criando e utilizando um Checker

Para permitir com que o envio de traps possa ser previamente agendado na aplicação, um outro artefato é necessário: o *checker*. Ao desenvolvedor resta criar uma classe que deve ser declarada com o estereótipo `@Checker`. Essa classe será instanciada e carregada automaticamente pelo mecanismo do CDI, que também procurará por métodos nela anotados com `@Scheduled`. Nessa anotação será definido o intervalo de repetição para a execução do comando.

Eis um exemplo de implementação de checker:

```
@Checker
public class EscolaChecker {

    @Inject
    private EscolaTrapper trapper;

    @Scheduled(interval = 30, unit = TimeUnit.SECONDS)
    public void checkDatabases() {
        trapper.sendDatabaseInfo("db1", conn1);
        trapper.sendDatabaseInfo("db3", conn3);
    }

    @Scheduled(interval = 1, unit = TimeUnit.MINUTES)
    public void checkUsedMemory() {
        trapper.sendHeapMemoryUsed();
    }

}
```

Ou seja, o estereótipo `@Checker` indica que a classe conterá métodos agendados. Estes devem ser anotados com `@Scheduled` e entre os seus parâmetros deve ser indicada a periodicidade de execução.

No exemplo em questão, o método `checkDatabases()` será invocado automaticamente pela aplicação a cada 30 segundos, fazendo com que o trapper `EscolaTrapper` envie as mensagens ao servidor. Da mesma forma, o método `checkUsedMemory()` será executado a cada minuto e provocará o envio de um trap.



Nota

O ciclo de vida de um objeto do tipo checker é controlado pela implementação do CDI, isto é, o programador não precisará se preocupar em instanciar um checker na aplicação.

20.2. Injeção de Trappers em um Checker

Objetos do tipo trapper podem ser facilmente injetados na classe checker com a anotação `@Inject`. Além disso, um mesmo checker pode conter referências a múltiplos trappers, tal como no exemplo a seguir:

```
@Checker
public class EscolaChecker {

    @Inject
    @SNMP
    private EscolaTrapper snmpTrapper;

    @Inject
    @Zabbix
    private EscolaTrapper zabbixTrapper;

    @Scheduled(interval = 30, unit = TimeUnit.SECONDS)
    public void checkDatabases() {
        snmpTrapper.sendDatabaseInfo("db1", conn1);
        snmpTrapper.sendDatabaseInfo("db3", conn3);
        zabbixTrapper.sendDatabaseInfo("db1", conn1);
        zabbixTrapper.sendDatabaseInfo("db3", conn3);
    }

    @Scheduled(interval = 1, unit = TimeUnit.MINUTES)
    public void checkUsedMemory() {
        snmpTrapper.sendHeapMemoryUsed();
        zabbixTrapper.sendHeapMemoryUsed();
    }

}
```

Apêndice A. Conceitos de Monitoração

A.1. NMS

Um *NMS (Network Management System)* ou *Sistema de Gerenciamento de Redes* é responsável pelas aplicações que monitoram e controlam os dispositivos gerenciados. Normalmente é instalado em um (ou mais de um) servidor de rede dedicado a estas operações de gerenciamento, que recebe informações (i.e. pacotes SNMP) de todos os dispositivos gerenciados daquela rede.

Entre os NMS mais populares podemos citar:

- ZABBIX: <http://www.zabbix.com/>
- Nagios: <http://www.nagios.org/>
- Zenoss: <http://www.zenoss.com/>
- OpenNMS: <http://www.opennms.org/>

A.2. SNMP

SNMP (Simple Network Management Protocol) é o protocolo de gerência típico de redes TCP/IP que facilita o intercâmbio de informação entre os dispositivos de rede. Ele possibilita aos administradores de rede gerenciar o desempenho da rede, encontrar e resolver seus eventuais problemas, e fornecer informações para o planejamento de sua expansão, dentre outras.

O "simple" do acrônimo SNMP diz respeito ao fato de que as interações entre os dispositivos gerenciados e um servidor não são terrivelmente complexas. Na versão original do protocolo, conhecida como SNMPv1, apenas quatro operações são disponíveis entre o servidor e o dispositivo a ser gerenciado:

Tabela A.1. Operações SNMPv1

Operação	Descrição
GET	Usada pelo servidor para obter uma única informação do dispositivo gerenciado. Por exemplo, podemos requisitar ao dispositivo o percentual de utilização da CPU. GET é uma operação de leitura.
GETNEXT	Usada pelo servidor para obter mais de uma informação do dispositivo gerenciado. Por exemplo, podemos requisitar o percentual de utilização da CPU e o uptime do sistema no dispositivo. GETNEXT também é uma operação de leitura.
SET	Usada pelo servidor para configurar um dispositivo com um valor específico. Por exemplo, para configurar um limiar no dispositivo que dispare um alerta ao servidor toda vez que a utilização de memória atingir 80% para um determinado período de tempo. SET é uma operação de escrita.
TRAP	Esta operação é diferente de todas as demais, pois ela é invocada pelo dispositivo gerenciado, e não pelo servidor. Uma mensagem do tipo "trap" geralmente é usada para alertar o servidor quando um determinado limiar é atingido ou no caso de ocorrer um erro ou evento de algum tipo específico. Por exemplo, um roteador pode ser

Operação	Descrição
	configurado para enviar um trap para o servidor assim que a sua utilização média de CPU estiver acima de 70% pelo período de um minuto. Mensagens do tipo trap são ditas assíncronas, no sentido de que o dispositivo gerenciado as envia sem que o servidor as tenha requerido.

A segunda versão do protocolo, chamada de SNMPv2, introduziu dois comandos adicionais:

Tabela A.2. Operações SNMPv1

Operação	Descrição
GETBULK	Esta operação foi implementada para permitir que os servidores obtenham múltiplos pedaços de informação de um dispositivo gerenciado a partir de uma única requisição. Tal como o GET, o GETBULK se trata de uma operação de leitura.
INFORM	Esta operação permite que um dado servidor reencaminhe a outros servidores informações do tipo trap que ele tenha recebido.

Além dos novos comandos apresentados, a versão 2 do SNMP oferece uma boa quantidade de melhorias em relação à versão inicial do protocolo, incluindo melhor performance, segurança, confidencialidade, novos tipos de dados e comunicações do tipo gerente-para-gerente.

O protocolo SNMP foi projetado para usar UDP como protocolo de transporte, e essa característica faz com que o SNMP seja considerado "leve", uma vez que o overhead do estabelecimento de uma sessão TCP é evitado. Quando um servidor precisa configurar um dispositivo ou coletar informação deste, ele fará uma requisição ao dispositivo através da porta de destino 161 UDP. Como na maioria dos serviços no mundo TCP/IP, a porta de origem geralmente será escolhida dinamicamente.

Talvez a funcionalidade mais importante do SNMP de uma perspectiva de monitoração seja a habilidade de um dispositivo gerenciado enviar mensagens do tipo trap para um servidor de forma assíncrona e tempestiva. Mensagens do tipo "trap" iniciadas pelo dispositivo são geralmente enviadas para a porta 162 UDP do servidor.

Ao protocolo SNMPv3 foram incluídos três importantes serviços: autenticação, privacidade e controle de acesso, conforme ilustrado no diagrama abaixo:

As implementações do SNMP oferecem suporte para as múltiplas versões existentes (vide [RFC-3584](http://tools.ietf.org/html/rfc3584) [http://tools.ietf.org/html/rfc3584]), tipicamente SNMPv1, SNMPv2c e SNMPv3.

A.3. JMX

JMX (Java Management Extensions) é uma tecnologia Java que fornece ferramentas para gerenciamento de monitoramento de aplicações, objetos de sistema, dispositivos e redes orientadas a serviço. É uma API que usa o conceito de agentes e com isso permite monitorar elementos da Máquina Virtual Java (JVM) e aplicativos que nela estão sendo executados.

Os recursos disponíveis na JMX são representados por objetos chamados *MBeans (Managed Beans)*, especificamente classes que podem ser dinamicamente carregadas e instanciadas e que podem ser usadas para se obter e ou alterar configurações das aplicações, coleta de estatísticas e notificação de eventos.

A JMX proporciona a integração do Java com o NMS e por isso pode ser encarada como um SNMP para aplicações em Java.