Paginação de conteúdo sob demanda.

Recupere o conteúdo de sua página consumindo menos recursos, paginando eles por demanda.

Cenário

Considere uma aplicação que recupera grande volume de dados e pretende exibi-los para o usuário em uma tabela, trata-se – nesse caso – de um sistema de listagem de clientes que exibe todos os clientes em um datatable onde cada linha representa um cliente. Obviamente, para não tornar a listagem muito longa, as linhas são exibidas por partes que chamamos de páginas e o tamanho delas é determinado pelo desenvolvedor. Sendo assim, vamos analisar a forma como o conteúdo dessa tabela é gerenciado, de que maneira as páginas dessa tabela são exibidas e como carregar o conteúdo para elas, para isso temos duas possibilidades.

A primeira possibilidade, considerada como má prática, consiste no cenário em que todos os clientes cadastrados na base são carregados e colocados em memória. Facilmente encontramos aplicações que fazem seleção em todos os registros da base de dados (sejam Clientes, Usuários entre outros) e inserem esses objetos em uma coleção para serem visualizados pelo usuário final. Muitas vezes alguma ação do usuário, por exemplo, qualquer filtro aplicado ou algum critério de ordenação no datatable faz com que os registros sejam todos carregados novamente.

O segundo cenário que vamos apresentar nesse artigo, considerado boa prática, é aquele no qual recuperamos da base de dados apenas os registros que estão sendo visualizados pelo usuário, por exemplo, suponhamos que existam 4 mil clientes na base, com essa abordagem vamos selecionar apenas a parte que compõe a primeira página do datatable, ignorando o restante dos registros que serão carregados à medida que alguma ação do datatable for realizada (mudar a página, aplicar filtro, buscar, ordenar entre outras).Utilizando a plataforma Java, JSF 2.2, Primefaces 5, Wildfly, JPA e EJB 3 fizemos uma aplicação para mostrar uma forma melhor de paginar um grande volume de dados, tendo em vista a performance, assim como a redução no tráfego de dados. Vamos comparar uma implementação que obtém os resultados através das duas formas mencionadas.

Sumário

[Cenário 1](#_Toc450033931)

[Sumário 1](#_Toc450033932)

[Introdução 1](#_Toc450033933)

[Implementação da PoC 2](#_Toc450033934)

[JavaServer™ Faces e PrimeFaces 2](#_Toc450033935)

[Enterprise Java Beans (EJB) 2](#_Toc450033936)

[O ambiente de desenvolvimento 2](#_Toc450033937)

[Criando o projeto 3](#_Toc450033938)

[Configurando o datasource 8](#_Toc450033939)

[Referências 10](#_Toc450033940)

[Autores 11](#_Toc450033941)

Introdução

Geralmente, as aplicações web que densevolvemos necessitam exibir uma grande quantidade de registros em uma tabela, a paginação desses registros é uma forma de interação com o usuário que permite dividir o conteúdo em várias páginas, evitando que as páginas fiquem desnecessariamente longas. Existem vários artigos e discussões na web bem relevantes sobre implementar ou não paginação no conteúdo da nossa aplicação e, caso seja implementado, qual deve ser o tamanho das páginas. Veja por exemplo a postagem [Avoid the Pains of Pagination](http://uxmovement.com/navigation/avoid-the-pains-of-pagination/) e os artigos [The Impact of Paging vs. Scrolling on Reading Online Text e Passages](http://usabilitynews.org/the-impact-of-paging-vs-scrolling-on-reading-online-text-passages/) [Paging vs. Scrolling: Looking for the Best Way to Present Search Results](http://usabilitynews.org/paging-vs-scrolling-looking-for-the-best-way-to-present-search-results/).

Uma vez que vamos implementar paginação para o conteúdo exibido em nossa aplicação, há uma porção de coisas que devemos nos atentar. Principalmente, a forma como o conteúdo é recuperado da base de dados e colocado na memória para que fique disponível ao usuário final.

A solução comumente adotada para essa tarefa é a de recuperar uma única vez o conteúdo da base de dados e depois adiciona-lo em uma coleção para ser utilizado pelo usuário e, a partir dessa coleção, pode-se paginar os dados em uma tabela. Note que, independente do volume de dados, no momento que o usuário solicita – por exemplo – os clientes cadastrados no sistema, a aplicação obtém tudo e coloca isso disponível na memória. Ou seja, a paginação pode ser feita com o que foi recuperado da base de dados, mas esse conteúdo na verdade já está totalmente disponível para o cliente (embora não esteja visível), pois a aplicação o colocou em memória. Então, dizemos que a paginação é feita em memória e consideramos essa abordagem como má prática, pois usa de forma indevida os recursos disponíveis.

Para resolver esse problema, utilizamos o que chamamos de paginação por demanda, que consiste em obter da base de dados apenas o conteúdo no qual o usuário possa interagir, ou seja apenas a página que é mostrada. Assim, é reduzido tanto a quantidade memória quanto o trafego de dados que é utilizado pela aplicação. A seguir, veremos ambas as implementações em funcionamento, tanto a paginação em memória quanto a paginação por demanda e faremos um estudo de caso que consiste em uma comparação da quantidade de memória e tempo utilizado no momento que consultamos os registros cadastrados na base.

Implementação da PoC

A aplicação que implementamos para provar esse conceito e comparar os resultados de ambas as formas de paginação utiliza a API Java EE 7. Para a interface com o usuário é utilizado o framework [PrimeFaces](http://primefaces.org/) 5.3 que implementa a especificação do [JavaServer™ Faces](https://javaserverfaces.java.net/) 2.2, além disso, utilizamos EJB 3 onde temos um session bean que tornará possível o acesso com a base de dados.

A seguir, detalhes das tecnologias que foram utilizadas para construir essa prova de conceito, mais informação sobre elas pode ser encontrada na secção de referências desse artigo.

JavaServer™ Faces e PrimeFaces

O JavaServer Faces ou simplesmente JSF é um framework que adota o padrão MVC (Model-View-Controller) orientado a eventos que, atualmente, encontra-se na versão 2.2 e foi lançado há mais de dez anos. É utilizado para a construção da interface com o usuário e, graças aos seus componentes e os managed beans, além de outras funcionalidades, vem ganhando cada vez mais aceitação entre os desenvolvedores Java. O PrimeFaces é uma biblioteca de código aberto que possui diversas funcionalidades e recursos além dos já existentes no JSF, podemos citar, por exemplo: mais de cem componentes prontos, suporte a Ajax e suporte a dispositivos móveis.

Enterprise Java Beans (EJB)

Enterprise JavaBeans ou EJB é um componente da plataforma JEE que tem o objetivo de fornecer um objeto que seja distribuível, transacional, seguro e portável. Atualmente, encontra-se na versão 3.2, entre as versões 2.0 e 3.0 houve uma grande mudança relacionado ao suporte as anotações Java, tornando a implementação dos EJB cada vez mais prática.

O ambiente de desenvolvimento

Assim como dito inicialmente, faremos uma aplicação que tem como principal funcionalidade consultar os clientes que foram cadastrados no sistema. Nosso objetivo é implementar as duas formas de paginação que mostramos anteriormente e comparar os resultados, para isso considere os seguintes softwares utilizados:

* Servidor de aplicação Wildfly 10.0.0 Final.
* Eclipse Luna para desenvolvedores Java EE.
* Java Development Kit 1.7.0 Update 67.
* SGDB PostgreSQL 9.5.
* Maven (embutido no Eclipse).

Criando o projeto

O projeto que usaremos como prova de conceito pode ser, facilmente, criado e configurado seguindo os passos descritos nessa seção, é importante dizer que que cada instrução aqui, deve ser feita tal qual é dito, com o intuito de garantir o funcionamento do exemplo.

Abra o Eclipse e clique no menu File > New > Maven Project para criar um novo projeto. Na janela, marque a opção: “Create a simple Project (skip archtype selection) “ e clique em Next, na próxima tela preencha os valores da forma que é mostrado na Figura 1 e clique em Finish, após isso o Maven criará um projeto que possui estrutura mostrada na Tabela 1.

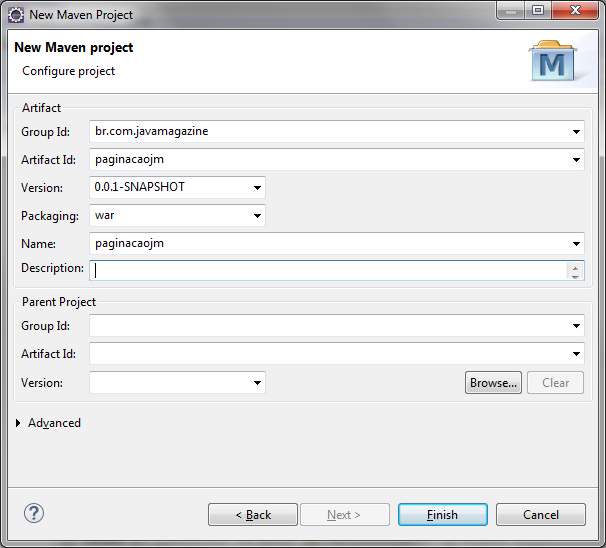


Figura 1 Criando *Maven Project*

|  |  |  |
| --- | --- | --- |
| Diretórios do projeto | | Descrição |
| src/main/ |  |  |
|  | java/ | Classes Java. |
|  | resources/ | Arquivos de propriedades. |
|  | webapp/ | Páginas JSF. |
| test/main/ |  |  |
|  | java/ | Classes de testes unitários. |
|  | resources/ | Arquivos de propriedades utilizados pelos testes. |
| target/ |  | Arquivos construídos pelo *Maven*. |
| pom.xml |  | Arquivo POM do *Maven.* |

Tabela 1 Estrutura do projeto criado.

Abra o arquivo pom.xml localizado na raiz do projeto e insira o conteúdo mostrado na Listagem 1, com isso as dependências necessárias para o projeto serão gerenciadas pelo Maven. Feito isso, vamos configurar o projeto no Eclipse. Primeiramente, clique com o botão direito no projeto e selecione a opção Java EE Tools > Generate Deployment Descriptor Stub para que seja criado o arquivo web.xml na pasta src/main/webapp/WEB-INF/, depois, vá nas propriedades do projeto e procure por Project Facets e então marque a opção JPA, observe que será mostrado uma falha de configuração do projeto logo abaixo das opções que é mostrado a mensagem Further configuration required (ver Figura 2), clique nessa frase para abrir as configurações da faceta JPA, veja as Figura 3 e Figura 4 e faça a configuração da forma que é mostrado lá.

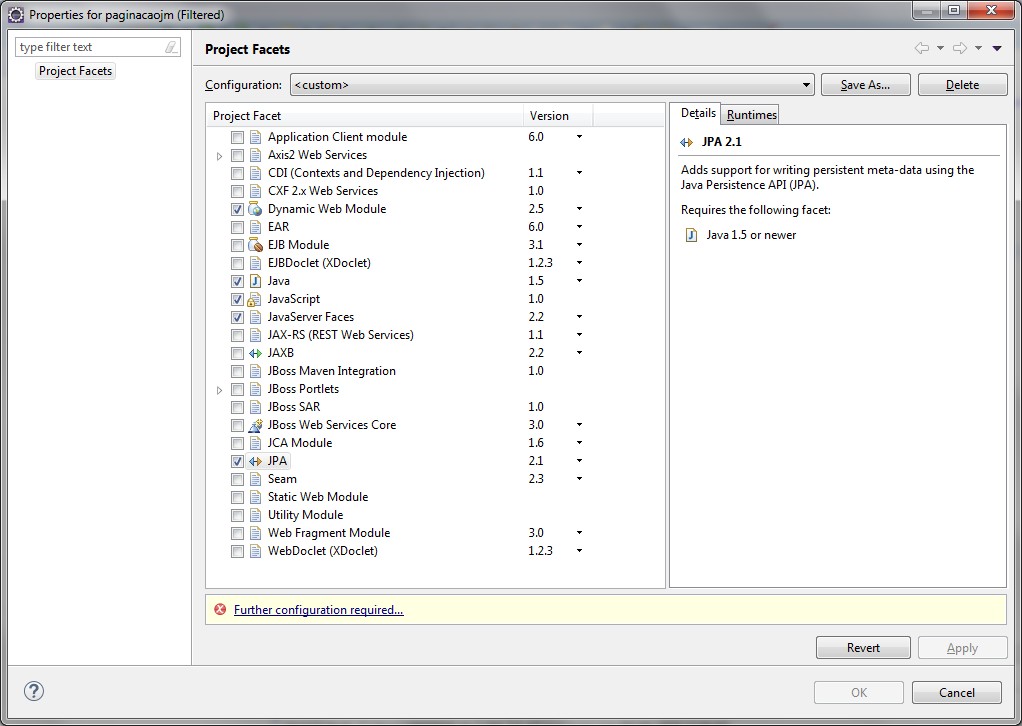


Figura 2 Janela Project Facets.

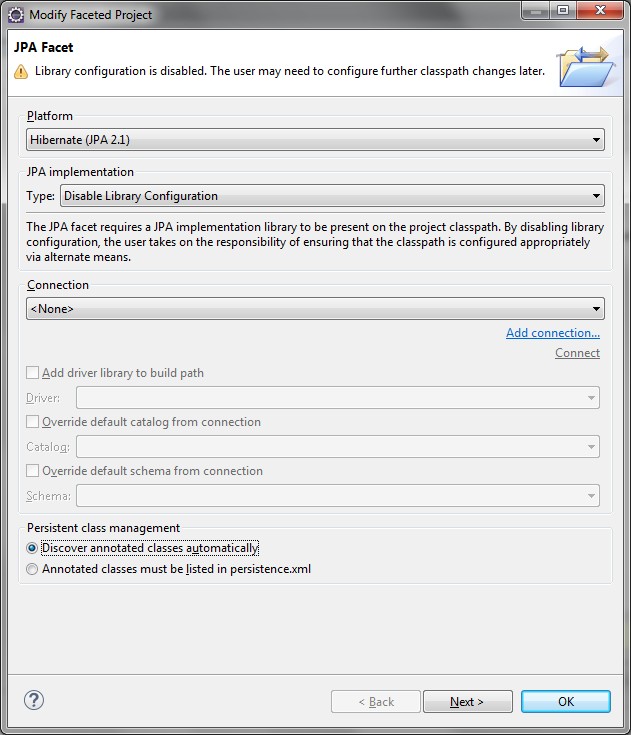


Figura 3 Configuração da faceta JPA do projeto.

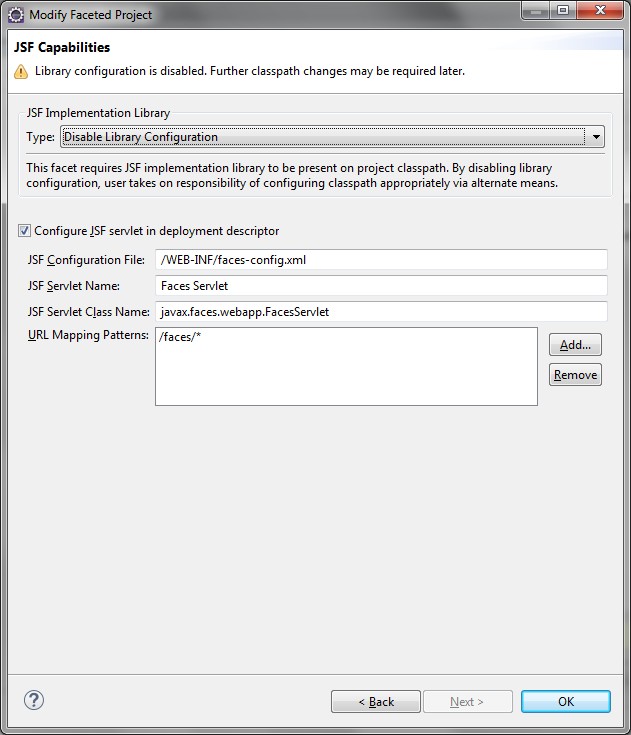


Figura 4 Configuração da faceta JPA do projeto.

Agora, vamos inserir nossas classes Java, portanto crie os quatro pacotes a seguir dentro do diretório src/main/java/:

* br.com.javamagazine.paginacaojm.domain,
* br.com.javamagazine.paginacaojm.ejb,
* br.com.javamagazine.paginacaojm.managedbean e
* br.com.javamagazine.paginacaojm.paginacao.

O pacote domain é utilizado para colocar todas as entidades do sistema, ou seja, classes que são representadas no banco de dados por tabelas e seus respectivos atributos são colunas delas. Utilizamos annotations para dizer ao JPA como essa entidade é representada no banco de dados. Então, crie uma classe chamada Cliente com conteúdo igual ao mostrado na Listagem 2. O pacote ejb é destinado aos Enterprise JavaBeans da aplicação, no nosso caso, teremos apenas o ClienteEJB (ver Listagem 3) que possui um método para cada forma de listagem de clientes. No pacote paginacao é colocado classes referentes a paginação dos objetos, nele colocamos a classe LazyClienteDataModel (ver ) que estente a classe do Primefaces chamada LazyDataModel, ambas as classes são responsáveis por lidar com grande quantidade de dados, levando em consideração o padrão de projeto conhecido como lazy loading (ver referências).

**Listagem 1** Arquivo POM do projeto.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>br.com.javamagazine</groupId>

<artifactId>paginacaojm</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>war</packaging>

<name>paginacaojm</name>

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.0.2</version>

<configuration>

<source>1.7</source>

<target>1.7</target>

</configuration>

</plugin>

</plugins>

</build>

<dependencies>

<!-- Bibliotecas JavaEE 7 que já estão no Wildfly, incluindo o JSF 2.2 -->

<dependency>

<groupId>javax</groupId>

<artifactId>javaee-api</artifactId>

<version>7.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.primefaces</groupId>

<artifactId>primefaces</artifactId>

<version>5.3</version>

</dependency>

<dependency>

<groupId>org.apache.myfaces.tomahawk</groupId>

<artifactId>tomahawk21</artifactId>

<version>1.1.14</version>

</dependency>

</dependencies>

</project>

**Listagem 2** Classe Cliente.java.

package br.com.javamagazine.paginacaojm.domain;

import java.util.Date;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="cliente")

public class Cliente {

@Id

private Integer id;

private String nome;

@Column(name="data\_nascimento")

private Date dataNascimento;

private String cpf;

@Column(name="qtd\_filhos")

private Integer qtdFilhos;

private String naturalidade;

private String endereco;

private String numero;

private String complemento;

private String bairro;

private String cep;

private String cidade;

private String uf;

@Column(name="telefone\_fixo")

private String telefoneFixo;

@Column(name="telefone\_celular")

private String telefoneCelular;

@Column(name="telefone\_comercial")

private String telefoneComercial;

private String email;

/\*\* Getters and Setter \*\*/

}

**Listagem 3** Classe ClienteEJB.java.

package br.com.javamagazine.paginacaojavamagazine.ejb;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import javax.ejb.Stateless;

import javax.persistence.PersistenceContext;

import org.hibernate.Criteria;

import org.hibernate.Session;

import org.hibernate.criterion.MatchMode;

import org.hibernate.criterion.Order;

import org.hibernate.criterion.Projections;

import org.hibernate.criterion.Restrictions;

import org.primefaces.model.SortOrder;

import br.com.javamagazine.paginacaojavamagazine.domain.Cliente;

@Stateless

public class ClienteEJB {

@PersistenceContext

private Session session;

public List <Cliente> listaClientes() {

Criteria criteria = session.createCriteria(Cliente.class);

criteria.addOrder(Order.asc("nome"));

return criteria.list();

}

public List <Cliente> listaClientesPorDemanda(Map <String, Object> filtros, String campoOrdenacao,

SortOrder tipoOrdenacao, int qtdRegistrosPorPagina, int registroInicial)

throws NoSuchFieldException, SecurityException {

Criteria criteria = session.createCriteria(Cliente.class);

// se algum filtro foi aplicado no datatable, adiciona as cláusulas e valores no where

if (!filtros.isEmpty()) {

adicionaClausulasEValoresNoWhere(filtros, criteria);

}

// se foi aplicado algum critério de ordenação no datatable

if (campoOrdenacao != null) {

criteria.addOrder(tipoOrdenacao ==

SortOrder.ASCENDING ? Order.asc(campoOrdenacao):Order.desc(campoOrdenacao));

} else { // senão, ordena pelo nome

criteria.addOrder(Order.asc("nome"));

}

// determina a quantidade de registros que deve ser retornada (qtd de linhas por página do datatable)

criteria.setMaxResults(qtdRegistrosPorPagina);

// Determina a partir de que registro deve retornar, por conta da página que o usuário

// selecionou no DataTable. Se está na página 2 e tem 10 registros por página, registroInicial

// será 10.

criteria.setFirstResult(registroInicial);

return criteria.list();

}

public Long retornaQuantidadeDeClientes(Map <String, Object> filtros)

throws NoSuchFieldException, SecurityException {

Long size = null;

Criteria criteria = session.createCriteria(Cliente.class);

// se algum filtro foi aplicado no datatable, adiciona as cláusulas no where e os parâmetros na consulta

if (filtros != null && !filtros.isEmpty())

adicionaClausulasEValoresNoWhere(filtros, criteria);

criteria.setProjection(Projections.rowCount());

size = (Long) criteria.uniqueResult();

return size;

}

private void adicionaClausulasEValoresNoWhere(Map <String, Object> filtros, Criteria criteria)

throws NoSuchFieldException, SecurityException {

for (Iterator < String > it = filtros.keySet().iterator(); it.hasNext();) {

String propriedadeFiltrada = it.next();

if (Cliente.class.getDeclaredField(propriedadeFiltrada).getType().equals(String.class))

// Se estiver filtrando um campo do tipo String, transforma o valor pesquisado e o

// valor do banco para maiúsculo para ignorar o case do campo e adiciona um parâmetro

// com o nome do campo pesquisado

criteria.add(Restrictions.like(propriedadeFiltrada,

(String) filtros.get(propriedadeFiltrada), MatchMode.ANYWHERE));

else

// Se o campo não for String, adiciona uma cláusula para verificar se o valor do

// campo no banco é igual ao valor passado adicionando um parâmetro com o nome

// do campo pesquisado

criteria.add(Restrictions.eq(propriedadeFiltrada, filtros.get(propriedadeFiltrada)));

}

}

}

**Listagem 4** Classe LazyClienteDataModel.java.

package br.com.javamagazine.paginacaojavamagazine.paginacao;

import java.util.List;

import java.util.Map;

import org.primefaces.model.LazyDataModel;

import org.primefaces.model.SortOrder;

import br.com.javamagazine.paginacaojavamagazine.domain.Cliente;

import br.com.javamagazine.paginacaojavamagazine.ejb.ClienteEJB;

public class LazyClienteDataModel extends LazyDataModel<Cliente> {

private List<Cliente> listaPaginadaDeClientes;

private ClienteEJB clienteEJBHibernate;

public LazyClienteDataModel(ClienteEJB clienteEJBHibernate) {

this.clienteEJBHibernate = clienteEJBHibernate;

}

@Override

public Cliente getRowData(String rowKey) {

for (Cliente cliente : listaPaginadaDeClientes)

if (cliente.getId().equals(rowKey))

return cliente;

return null;

}

@Override

public Object getRowKey(Cliente cliente) { return cliente.getId(); }

@Override

public List<Cliente> load(int primeiroRegistro, int qtdRegistrosPagina, String campoOrdenacao,

SortOrder tipoOrdenacao, Map<String, Object> filtros) {

try {

// Guardo a quantidade de registros retornada passando o possível filtro

// aplicado, sem utilizar a ordenação porque não precisa

this.setRowCount(clienteEJBHibernate.retornaQuantidadeDeClientes(filtros).intValue());

// Retorno a lista de clientes devolvida de acordo com o possível filtro aplicado

// e utilizando um eventual critério de ordena??o especificado pelo usuário

return listaPaginadaDeClientes = clienteEJBHibernate.listaClientesPorDemanda(filtros,

campoOrdenacao, tipoOrdenacao, qtdRegistrosPagina, primeiroRegistro);

} catch (NoSuchFieldException|SecurityException e) {

e.printStackTrace();

return null;

}

}

}

Configurando o datasource

Há bastante material que mostra como configurar corretamente um datasource utilizando WildFly e PostgreSQL, por exemplo, buscar alguma referência do DevMedia. Caso você já tenha configurado, pule para a próxima seção.

Após descarregar e descompactar o WildFly no diretório que chamaremos de WILDFLY\_HOME, primeiramente, vamos configurar o módulo do PostgreSQL. Na pasta modules em WILDFLY\_HOME, crie o seguinte diretório org/postgres/main, depois insira dentro o driver JDBC que deve ser usado para conectar-se com o banco de dados (na seção de referências há um link onde o mesmo pode ser encontrado). Junto com o driver crie um arquivo chamado module.xml com o conteúdo mostrado na **Erro! Fonte de referência não encontrada.Erro! Fonte de referência não encontrada.**Listagem 4.

Com o módulo configurado, vamos ajustar a instância standalone do servidor para carregar o driver e o datasource. Edite o arquivo standalone.xml encontrado dentro do diretório WILDFLY\_HOME/ standalone/ configuration/, acrescente ao arquivo dentro da marcação datasources o conteúdo indicado na Listagem 5. Por fim, certifique-se de ter no persistence.xml o conteúdo mostrado na Listagem 6.

**Listagem 5** Arquivo de configuração do módulo do PostgreSQL.

<?xml version=”1.0” ?>

<module xmlns=”urn:jboss:module:1.1” name=”org.postgres”>

<resources>  
 <resource-root path=”postgresql-X.X.XXXX.jar”/>  
 </resources>

<dependencies>  
 <module name=”javax.api”/>  
 <module name=”javax.transaction.api”/>  
 <module name=”javax.servlet.api” optional=”true”/>  
 </dependencies>  
</module>

**Listagem 6** Arquivo de configuração standalone.xml do WildFly.

<datasource jndi-name="java:jboss/datasources/PaginacaoDs"

pool-name="PaginacaoDs" enabled="true" use-java-context="true">

<connection-url>jdbc:postgresql://localhost:5432/PaginacaoDs?ApplicationName=PaginacaoDs</connection-url>

<driver>postgres</driver>

<pool>

<min-pool-size>5</min-pool-size>

<initial-pool-size>5</initial-pool-size>

<max-pool-size>100</max-pool-size>

<prefill>true</prefill>

</pool>

<security>

<user-name>postgres</user-name>

<password>postgres</password>

</security>

<validation>

<valid-connection-checker

class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLValidConnectionChecker"/>

<exception-sorter

class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLExceptionSorter"/>

</validation>

</datasource>

<xa-datasource jndi-name="java:jboss/datasources/PaginacaoDsXa"

pool-name="PaginacaoDsXa" enabled="true" use-java-context="true">

<xa-datasource-property name="ServerName">localhost</xa-datasource-property>

<xa-datasource-property name="PortNumber">5432 </xa-datasource-property>

<xa-datasource-property name="DatabaseName">PaginacaoDs</xa-datasource-property>

<driver>postgres</driver>

<xa-pool>

<min-pool-size>5</min-pool-size>

<initial-pool-size>5</initial-pool-size>

<max-pool-size>100</max-pool-size>

<prefill>true</prefill>

</xa-pool>

<security>

<user-name>postgres</user-name>

<password>postgres</password>

</security>

<validation>

<valid-connection-checker

class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLValidConnectionChecker"/>

<exception-sorter

class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLExceptionSorter"/>

</validation>

</xa-datasource>

<drivers>

<driver name="postgres" module="org.postgres">

<xa-datasource-class>org.postgresql.xa.PGXADataSource</xa-datasource-class>

<datasource-class>org.postgresql.ds.PGSimpleDataSource</datasource-class>

</driver>

</drivers>

**Listagem 7** Arquivo de configuração de contexto JPA, persistence.xml.

<?xml version="1.0" encoding ="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd" version="2.0">

<persistence-unit name="paginacaoPU" transaction-type="JTA">

<provider>org.hibernate.ejb.HibernatePersistence</provider>

<jta-data-source>java:jboss/datasources/PaginacaoDs</jta-data-source>

<properties>

<property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />

<property name="hibernate.show\_sql" value="false" />

<property name="hibernate.format\_sql" value="false" />

<property name="hibernate.hbm2ddl.auto" value="create" />

</properties>

</persistence-unit>

</persistence>

Referências

* <http://uxmovement.com/navigation/avoid-the-pains-of-pagination/>
* <http://usabilitynews.org/the-impact-of-paging-vs-scrolling-on-reading-online-text-passages/>
* <http://usabilitynews.org/paging-vs-scrolling-looking-for-the-best-way-to-present-search-results/>
* <http://www.primefaces.org/docs/guide/primefaces_user_guide_5_3.pdf>
* <https://github.com/primefaces/primefaces>
* <https://pt.wikipedia.org/wiki/Enterprise_JavaBeans>
* <http://wildfly.org/>
* <http://www.eclipse.org/>
* <https://jdbc.postgresql.org/>
* <https://en.wikipedia.org/wiki/Lazy_loading>

Autores

|  |  |
| --- | --- |
| C:\DevMedia\Revistas\Java Magazine\_Edicoes\jm 105\pablo.png |  |
| **Pablo Bruno de Moura Nóbrega** <http://pablonobrega.wordpress.com> | **Joel Xavier Rocha**  <http://joelxr.github.io> |
| [pablonobrega2004@gmail.com](mailto:pablonobrega2004@gmail.com) | [joelxr@gmail.com](mailto:joelxr@gmail.com) |
| Líder de Projeto Java, certificado OCJP e OCWCD, Graduado em Ciências da Computação pela Universidade de Fortaleza – UNIFOR, Mestre em Computação pela Universidade Estadual do Ceará – UECE, cursa MBA em Gerenciamento de Projetos na UNIFOR, trabalha na Secretaria Municipal de Finanças de Fortaleza – SEFIN e desenvolve sistemas há cerca de oito anos. | Analista de Sistemas Java, certificado OCJP, bacharel em Engenharia da Computação pelo Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE). Trabalha na Secretária de Finanças do Município de Fortaleza no Ceará (SEFIN) e desenvolve e mantem sistemas há mais de cinco anos. |