Quando trabalhamos com sistemas JSF, precisamos tomar cuidado com as consultas que fazemos e exibimos em tabelas de dados.

O componente <p:dataTable> do PrimeFaces possui o recurso de paginação de dados, mas embora possa parecer que exibir os dados paginados para o usuário seja suficiente quando trabalhamos com grandes conjuntos de dados, não é bem assim!

Neste artigo você vai aprender a fazer paginação de dados com o componente DataTable do PrimeFaces usando o recurso de Lazy Loading, e assim você vai poder evitar estouros de memória no servidor ao trabalhar com consultas que retornam milhares de registros.

Como você já deve saber, pra configurar paginação em uma tabela de dados do PrimeFaces, tudo que precisamos fazer é adicionar a propriedade paginator="true".

|  |
| --- |
| <p:dataTable var="lancamento" value="#{consultaLancamentosBean.lancamentos}"   paginator="true" rows="5" lazy="true">  ...  </p:dataTable> |

Nesse exemplo, se a pesquisa que carrega a coleção “lancamentos” retornar muitos objetos, eles podem ocupar toda a memória disponível pra JVM e acabar ocorrendo um estouro de memória no servidor.

Mesmo que isso não aconteça, o sistema pode começar a ficar muito lento, porque os milhares objetos retornados são armazenados na memória, de acordo com o escopo do bean “consultaLancamentosBean”.

Se por exemplo uma consulta retornar 50.000 registros, não faz muito sentido guardarmos tudo isso na memória, esperando o usuário navegar em todas as páginas. Seria um desperdício muito grande dos recursos do servidor.

Imagine que configuremos cada página do DataTable para apresentar 50 registros de cada vez. Dessa forma, no caso da consulta de 50.000 registros, teríamos 1.000 páginas.

Podemos programar o DataTable para trabalhar de forma “preguiçosa”, ou seja, buscando apenas os registros que precisam ser exibidos na página atual.

Pra fazer isso, além de configurar o DataTable, temos que pensar também na consulta. Nesse caso, estamos usando JPA e uma classe de repositório.

Criei uma classe FiltroLancamento para representar alguns atributos que vamos precisar filtrar no método de consulta do repositório.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | **public class FiltroLancamento implements Serializable {** |   private static final long serialVersionUID = 1L;  private String descricao;  private int primeiroRegistro;  private int quantidadeRegistros;  private String propriedadeOrdenacao;  private boolean ascendente;    // getters e setters    ...    **}** |

O repositório Lancamentos é a classe responsável por abstrair o acesso à tabela de lançamentos no banco de dados.

Nessa classe, criamos o método filtrados, que retorna uma coleção de lançamentos, dado um objeto do tipo FiltroLancamento. A consulta deve levar em consideração o primeiro registro e máximo permitido.

Usamos a API de Criteria do Hibernate pra definir isso, a partir dos métodos setFirstResult e setMaxResults.

Outro método que criamos é o quantidadeFiltrados, que retorna a quantidade total de lançamentos, também baseado no objeto de filtro passado como parâmetro.

|  |
| --- |
| **public class Lancamentos implements Serializable {**    private static final long serialVersionUID = 1L;    @Inject  private EntityManager manager;    @SuppressWarnings("unchecked")  public List<Lancamento> filtrados(FiltroLancamento filtro) {  Criteria criteria = criarCriteriaParaFiltro(filtro);    criteria.setFirstResult(filtro.getPrimeiroRegistro());  criteria.setMaxResults(filtro.getQuantidadeRegistros());    if (filtro.isAscendente() && filtro.getPropriedadeOrdenacao() != null) {  criteria.addOrder(Order.asc(filtro.getPropriedadeOrdenacao()));  } else if (filtro.getPropriedadeOrdenacao() != null) {  criteria.addOrder(Order.desc(filtro.getPropriedadeOrdenacao()));  }    return criteria.list();  }    public int quantidadeFiltrados(FiltroLancamento filtro) {  Criteria criteria = criarCriteriaParaFiltro(filtro);    criteria.setProjection(Projections.rowCount());    return ((Number) criteria.uniqueResult()).intValue();  }    private Criteria criarCriteriaParaFiltro(FiltroLancamento filtro) {  Session session = manager.unwrap(Session.class);  Criteria criteria = session.createCriteria(Lancamento.class);    if (StringUtils.isNotEmpty(filtro.getDescricao())) {  criteria.add(Restrictions.ilike("descricao", filtro.getDescricao(), MatchMode.ANYWHERE));  }    return criteria;  }  **}** |

Quando especificamos o primeiro registro e o máximo usando os métodos setFirstResult e setMaxResults, a consulta SQL gerada para o MySQL é algo parecido com o seguinte:

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | select          this\_.id as id1\_0\_1\_,          this\_.data\_pagto as data\_pag2\_0\_1\_,          this\_.data\_vencimento as data\_ven3\_0\_1\_,          this\_.descricao as descrica4\_0\_1\_,          this\_.pessoa\_id as pessoa\_i7\_0\_1\_,          this\_.tipo as tipo5\_0\_1\_,          this\_.valor as valor6\_0\_1\_,          pessoa2\_.id as id1\_1\_0\_,          pessoa2\_.nome as nome2\_1\_0\_      from          Lancamento this\_      inner join          Pessoa pessoa2\_              on this\_.pessoa\_id=pessoa2\_.id limit ?,          ? | |

Veja que a cláusula limit é usada para para restringir o número de registros que a consulta deve retornar.

Agora, precisamos criar uma classe que herda LazyDataModel.

Vamos criar uma classe anônima dentro do managed bean que gerencia a página de consulta de lançamentos, porque não pretendemos reaproveitá-la.

|  |
| --- |
| **public class ConsultaLancamentosBean implements Serializable {**  private static final long serialVersionUID = 1L;  @Inject  private Lancamentos lancamentos;    private FiltroLancamento filtro = new FiltroLancamento();  private LazyDataModel<Lancamento> model;    public ConsultaLancamentosBean() {  model = new LazyDataModel<Lancamento>() {  private static final long serialVersionUID = 1L;    @Override  public List<Lancamento> load(int first, int pageSize,  String sortField, SortOrder sortOrder,  Map<String, Object> filters) {    filtro.setPrimeiroRegistro(first);  filtro.setQuantidadeRegistros(pageSize);  filtro.setAscendente(SortOrder.ASCENDING.equals(sortOrder));  filtro.setPropriedadeOrdenacao(sortField);    setRowCount(lancamentos.quantidadeFiltrados(filtro));    return lancamentos.filtrados(filtro);  }    };  }  ...  } |

Quando herdamos a classe LazyDataModel, precisamos substituir e implementar o método load, que recebe alguns parâmetros.

Nesse método, devemos consultar a quantidade total de registros e invocar o método setRowCount.

Além disso, precisamos também consultar e retornar os objetos filtrados e limitados para a página atual.

Agora ficou fácil! Na tabela de dados, vamos apenas adicionar a propriedade lazy="true" e vincular a DataTable com a instância da classe anônima que criamos, filha de LazyDataModel.

|  |
| --- |
| <p:dataTable var="lancamento" value="#{consultaLancamentosBean.model}"     paginator="true" rows="5" lazy="true">     ...  </p:dataTable> |

Pronto! Temos uma tabela de dados que, além de exibir apenas os registros da página atual, também só consulta e armazena em memória esses mesmos registros.

Assista ao vídeo de 35 minutos com o passo a passo e demonstração do projeto de exemplo, que vai ficar muito mais fácil pra você entender ainda melhor.

Além disso, o [código-fonte do projeto de exemplo está no GitHub](https://github.com/algaworks/aula-datatable-lazy-loading-primefaces)!

E aí, o que achou desse recurso do PrimeFaces? Você já usa? Acha que isso pode te ajudar em algum projeto? Espero seu comentário! ;)

Para aprender mais sobre PrimeFaces, [conheça nosso curso online de Sistemas Comerciais Java EE com CDI, JPA e PrimeFaces](http://www.algaworks.com/curso/sistemas-comerciais-java-ee-com-cdi-jpa-e-primefaces/?utm_source=blow-aw-datatable-lazy&utm_medium=link-final&utm_campaign=curso-javaee), que é completo e substitui a necessidade de cursos presenciais.