# Pagination with Spring Framework 2.x

## Introduction

In a web application, Pagination, or Page Navigation, is a technique often used to display a large set of data. For a large hundred or thousand or even millions of records, it would cause large performance problem to return all of that data in a single page. Pagination is used to present data to user one page at a time, typically 25 to 100 records per page. Moreover, keyword search, advance search option and column sorting are features often needed together with Pagination.

This article will focus on an implementation of Pagination with Spring Framework that will provide full features of pagination, search option and column sorting.

## Technologies

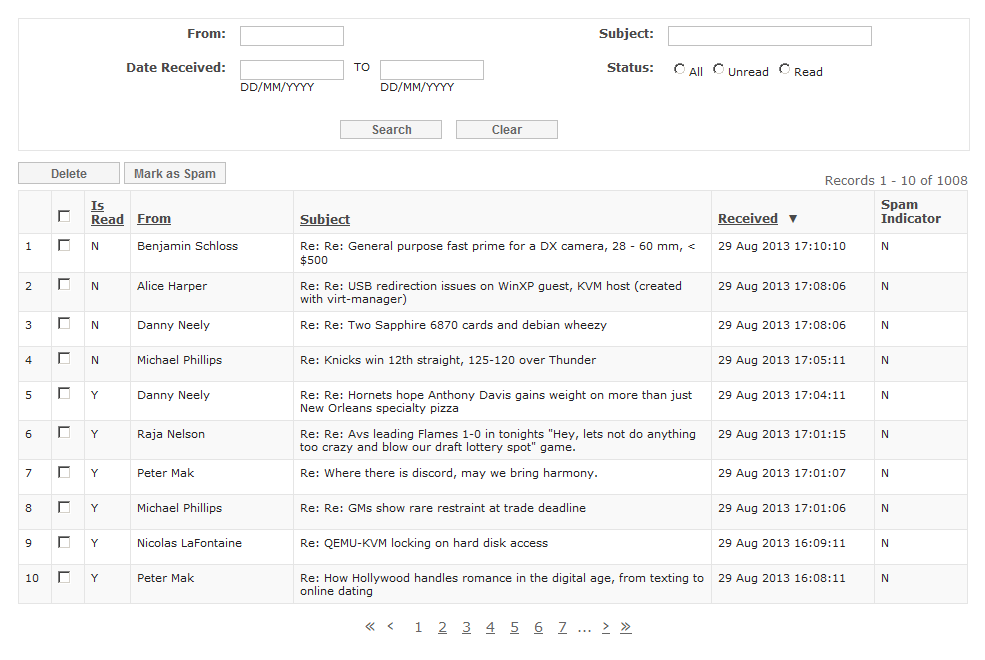
The following technologies are needed:

* Pagination Framework for Spring
* Spring Framework 2.5.2
* Hibernate as JPA provider
* JSP 2.1 and JSTL 1.2 (JEE 5)
* JDK 1.6

## Approach

This article will use a top-down approach, starting with user interface (View), working the way down to Controller, then to Service and Repository (Model / DAO) layer.

Suppose our Project Manager has gathered user requirements for a Mail Client System and an application prototype produced:



Top of the page will have a few search options for users to search through his mails. On the middle left there are 2 action buttons for users to delete or mark mail as spam. On the middle right side total number of emails and the corresponding page number will be shown. The mail records are displayed to user one page at a time, 10 mails per page. Columns can be sorted by Subject, Received Date, etc. And finally, pagination component is rendered for user to change page. The left most part is for go to the first page; follow by link to go to previous page; follow by numeric items to go to a specific page; follow by next page and last page link.

In order to implement pagination, we will need to create the following files:

|  |  |  |
| --- | --- | --- |
| **Object** | **Extends** | **Description** |
| BoMailMessage | BoPaginationResultRow | Business Object for storing search options |
| BoMailSearchParam | BoPaginationParam | Business Object for storing one row of data |
| MailClientController | PaginationControllerAbstract | Controller to interpret user input and set data into model |
| MailClientService | PaginationServiceAbstract | Business Logic is implemented and raw data from JPA is mapped to Business Object |
| MailClientDao | PaginationDaoDbEntityManagerAbstract | Database query |
| inbox.jsp | http://pagination/pagination-spring3.tld | User Interface presentation |
| PgMailMessage  PgPerson | Nil | Database mapping object |

## User Interface (JSP)

To create a JSP page for Mail Client System, we will use Pagination Framework for Spring. Pagination Framework contains a predefined JSP Tag Library for generating our pagination page. We will also need spring tag library and spring form tag library, together with standard JSTL core library.

|  |
| --- |
| <%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>  <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  <%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>  <%@taglib prefix="pg" uri="http://pagination/pagination-spring3.tld" %> |

Next, we will need html form tag for search option submission. We will use spring form tag to create our form. With this form tag, we will expose a business object pparam (short for page parameters) using attribute modelAttribute. The action attribute is set to the page url, which is defined in another business object paginationResult. Business objects (BO) pparam and paginationResult formed the 2 (and only 2) POJO objects that we will use to present our pagination data.

|  |
| --- |
| <form:form id="pgform" method="post" modelAttribute="pparam" action="${paginationResult.pageLink}">  .  .  .  </form:form> |

Next, use Pagination Framework tag library to create our pagination search and data content. Pagination tag library has tag-file name “pagination” (<pg:pagination>) to help generate the pagination content. pg:pagination has 6 attributes: pparam, paginationResult, searchContent, controlButton, columnsContent and contentAfterLastRow. For pparam and paginationResult, I will explain them in more details in Business Object section below. Now let’s first look at how we use <pg:pagination> tag:

|  |
| --- |
| <pg:pagination pparam="${pparam}" paginationResult="${paginationResult}">  <jsp:attribute name="searchContent">  <table class="searchtable">  <tr>  <td class="caption" style="width:200px;">From:</td>  <td style="width:250px;"><form:input path="from" cssStyle="width:100px;"/></td>  <td class="caption" style="width:150px;">Subject:</td>  <td style="width:200px;"><form:input path="subject" cssStyle="width:200px;"/></td>  </tr>  <tr>  <td class="caption" style="width:200px;">Date Received:</td>  <td style="width:250px;">  <span style="float:left;">  <form:input path="fromDate" cssStyle="width:100px;"/><br/><small>DD/MM/YYYY</small>  </span>  <span style="float:left;padding-left:10px;padding-right:10px;"> TO </span>  <span style="float:left;">  <form:input path="toDate" cssStyle="width:100px;"/><br/><small>DD/MM/YYYY</small>  </span>  </td>  <td class="caption" style="width:150px;">Status:</td>  <td style="width:200px;vertical-align:top;">  <form:radiobutton path="status" value="A"/>All  <form:radiobutton path="status" value="U"/>Unread  <form:radiobutton path="status" value="R"/>Read  </td>  </tr>  <tr>  <td style="text-align:center;padding-top:20px;" colspan="4">  <span class="button"><form:button id="searchButton" name="buttonAction" value="searchButton" class="button">Search</form:button></span>  <span style="padding-left:10px;"></span>  <span class="button"><form:button id="clearButton" name="buttonAction" value="clearButton" class="button">Clear</form:button></span>  </td>  </tr>  </table>  </jsp:attribute>  <jsp:attribute name="controlButton">  <div style="padding-top:10px;">  <span class="button"><form:button id="deleteButton" name="buttonAction" value="deleteButton" class="button">Delete</form:button></span>  <span class="button"><form:button id="markSpamButton" name="buttonAction" value="markSpamButton" class="button">Mark as Spam</form:button></span>  </div>  </jsp:attribute>  <jsp:attribute name="columnsContent">  <td class="cell"><span><c:out value="${bo.isRead}"/></span></td>  <td class="cell"><span style="white-space:nowrap;"><c:out value="${bo.fromName}"/></span></td>  <td class="cell" style="width:100%;"><span><c:out value="${bo.subject}"/></span></td>  <td class="cell"><span style="white-space:nowrap;"><fmt:formatDate value="${bo.sendDate}" pattern="d MMM yyyy HH:mm:ss"/></span></td>  <td class="cell"><span><c:out value="${bo.isSpam}"/></span></td>  </jsp:attribute>  </pg:pagination> |

searchContent, controlButton, columnsContent and contentAfterLastRow are all jsp fragments.

searchContent contains the search options; form objects (input, radiobutton, button, etc.) use spring form tag library with attribute “path” set to variable in pparam (remember we set modelAttribute=”pparam” in our form). For example, <form:input path="fromDate"> will reference pparam.fromDate variable.

controlButton contains 2 buttons for delete and mark as spam action.

columnsContent defines how we present the search result, that is each row of data (each mail), to our users. Here an object instance named “bo” is binded to this jsp fragment and can be used to load data from our BoMailMessage object (see below).

contentAfterLastRow should defines an area just below the last row of data in current page. We didn’t need it in this example.

## Business Object (BO)

### pparam

Page Parameter (pparam) is a business-object representing the search options. We declare this object in com.github.paginationspring.example.mailclient.bo.BoMailSearchParam. This object extends com.github.paginationspring.bo.BoPaginationParam (POJO) and added 5 fields from our search options:

|  |
| --- |
| public class BoMailSearchParam extends BoPaginationParam {  private String from;  private String subject;  private String fromDate;  private String toDate;  private String status; |

pparam will keep value of search option in this pojo and use it in our pagination framework.

### paginationResult

Pagination Result is a business-object representing results of our search. It contains one page of data (as seen by user on his current page), and it also contains some settings like record-per-page, default sorting order, etc. Each row of data (in our case, each mail) is represented by declaring object com.github.paginationspring.example.mailclient.bo.BoMailMessage. This object extends com.github.paginationspring.bo.BoPaginationResultRow (POJO) and added 5 fields for display our search results:

|  |
| --- |
| public class BoMailMessage extends BoPaginationResultRow<Integer> {  private int messageId;  private String fromName;  private String fromEmail;  private String toName;  private String toEmail;  private String subject;  private String message;  private Date sendDate;  private String isSpam;  private String isRead;  @Override  public Integer getPk() {  return messageId;  } |

Notice the type inference of <Integer>, this Integer represents the object type of the unique key of this row of data (i.e. messageId).

## Controller

Controller will define settings of our pagination page, map requests URLs into a handler method and pass business logic to service layer. First we define our controller with @Controller annotation and extend a controller base PaginationControllerAbstract<BoMailSearchParam> as shown here:

|  |
| --- |
| @Controller  public class MailClientController extends PaginationControllerAbstract<BoMailSearchParam> { |

In our constructor, we will define some settings for our pagination results.

|  |
| --- |
| public MailClientController() {  setOptionDisplayCheckbox(true);  setOptionDisplaySerialNo(true);  setOptionWidth(950);  setDefaultRecordPerPage(10);  setDefaultSortName("Received");  setDefaultSortAscDesc("d");  setPageLink("/org/pagination/example/mailclient/inbox.do");  setRewriteUrl(true);  } |

|  |  |
| --- | --- |
| setOptionDisplayCheckbox | whether or not the left side checkbox should be displayed |
| setOptionDisplaySerialNo | whether or not the left side serial no. should be displayed |
| setOptionWidth | the width of our search results, in pixels |
| setDefaultRecordPerPage | default record per page |
| setDefaultSortName | default sorting column |
| setDefaultSortAscDesc | default sorting order, value “a” is ascending and “d” is decending. |
| setPageLink | url of your pagination page |
| setRewriteUrl | to give you nicer, SEO (Search Engine Optimization) friendly, urls. For example, by setting setRewriteUrl to true, URL mailclient/inbox.do?sortAscDesc=a&sortName=From becomes mailclient/sort/From/a/inbox.do |

Next, we will need to map request URLs into a handler method. We do this by using the @RequestMapping, @ModelAttribute and @RequestParam annotations:

|  |
| --- |
| @RequestMapping(value={  URL1+"/inbox.do"  , URL2+"/inbox.do"  , URL3+"/inbox.do"  , URL4+"/inbox.do"  }, method = {RequestMethod.GET, RequestMethod.POST})  public String defineJsp(@ModelAttribute(PPARAM) BoMailSearchParam pparam, @RequestParam(value=BUTTON\_ACTION, required=false) String buttonAction, Model model) throws Exception {  Map<String, Object> map = assignModel(pparam, buttonAction);  model.addAllAttributes(map);  if ( "deleteButton".equalsIgnoreCase(buttonAction) && !ArrayUtils.isEmpty(pparam.getSelectedIds()) ) {  // Delete button is pressed  for ( String pk : pparam.getSelectedIds() ) {  log.info("selected id="+pk);  }  }  if ( "markSpamButton".equalsIgnoreCase(buttonAction) && !ArrayUtils.isEmpty(pparam.getSelectedIds()) ) {  // Mark as Spam button is pressed  for ( String pk : pparam.getSelectedIds() ) {  log.info("spam selected id="+pk);  }  }  return "/org/pagination/example/mailclient/view/inbox";  } |

In @RequestMapping, you will notice URL1, URL2, URL 3 and URL 4 variables. There constant variables predefined some url patterns to be used when we set option setRewriteUrl to true. If setRewriteUrl is set to false, simply use @RequestMapping(value=”/inbox.do”…).

In our method declaration, the return type String is logical view name and in our case, the jsp file name. Method argument pparam is annotated with @ModelAttribute to bind request parameters to bean properties from our form object (remember we have <form:form modelAttribute=”pparam”>). Method argument buttonAction is annotated with @RequestParam to access button request parameter searchButton, clearButton, deleteButton and markSpamButton. Finally, method argument model is used to expose paginationResult object to web view.

Inside this method, we will call a parent method assignModel from our Pagination Framework, set the returned objects into our model arguments and result the jsp url. The assignModel method will call a service bean, which we will be injecting into our controller. The service bean implements our business logic in our pagination, which we will explain in the next section.

|  |
| --- |
| Map<String, Object> map = assignModel(pparam, buttonAction);  model.addAllAttributes(map);  .  .  .  return "/org/pagination/example/mailclient/view/inbox"; |

At last, inject our service bean with @Autowired annotation and call a parent method setPaginationService:

|  |
| --- |
| @Autowired  public void setPaginationService(MailClientService mailClientService) {  super.setPaginationService(mailClientService);  } |

## Service

Service bean serve 2 purposes: 1) map database raw data into our business object and 2) define column properties for result data.

First, we define our service bean with @Service annotation and extend a service base PaginationServiceAbstract as shown here:

|  |
| --- |
| @Service  public class MailClientService extends PaginationServiceAbstract<BoMailSearchParam, BoMailMessage, PgMailMessage> { |

Type inference of < BoMailSearchParam, BoMailMessage, PgMailMessage> represent our page parameter, business object and data entity object respectively. Data Entiry PgMailMessage is generated with Hibernate / JBoss Tools from database structure.

To map database raw data into our business object, override assignDataToBo method:

|  |
| --- |
| @Override  protected BoMailMessage assignDataToBo(PgMailMessage en) throws Exception {  BoMailMessage bo = new BoMailMessage();  bo.setMessageId(en.getMessageId());  bo.setFromEmail(en.getFromPgPerson().getEmail());  bo.setFromName(en.getFromPgPerson().getName());  bo.setToEmail(en.getToPgPerson().getEmail());  bo.setToName(en.getToPgPerson().getName());  bo.setSubject(en.getSubject());  bo.setSendDate(en.getSendDate());  bo.setIsSpam(en.getIsSpam());  bo.setIsRead(en.getIsRead());  return bo;  } |

To define column properties, override assignColumnsDefinition method:

|  |
| --- |
| @Override  public void assignColumnsDefinition(List<BoPaginationColumn> columns)  throws Exception {  log.debug("setting columns def.");  BoPaginationColumn col = null;  col = new BoPaginationColumn();  col.setColumnName("Is Read");  col.setOrderColumns("a.isRead");  col.setOrderDirections("asc");  columns.add(col);  col = new BoPaginationColumn();  col.setColumnName("From");  col.setOrderColumns("b.name, a.sendDate");  col.setOrderDirections("asc, desc");  col.setWidth(150);  columns.add(col);  col = new BoPaginationColumn();  col.setColumnName("Subject");  col.setOrderColumns("a.subject");  col.setOrderDirections("asc");  columns.add(col);  col = new BoPaginationColumn();  col.setColumnName("Received");  col.setOrderColumns("a.sendDate");  col.setOrderDirections("desc");  col.setWidth(150);  columns.add(col);  col = new BoPaginationColumn();  col.setColumnName("Spam Indicator");  col.setWidth(80);  columns.add(col);  } |

We create a new BoPaginationColumn bean for each column. BoPaginationColumn has 4 variables: columnName, orderColumns, orderDirections and width. The columnName is the name of column to be displayed to user. The orderColumns is a query string to be added to our JPA query “order by” clause. The orderDirections has value “asc” or “desc” for ascending or descending shorting of the column. For sorting across multiple fields, use comma to separate between required fields (as shown in “From” column above). The width variable define the column width in pixels.

At last, inject our DAO bean with @Autowired annotation and call a parent method setPaginationDao:

|  |
| --- |
| @Autowired  public void setPaginationDao(MailClientDao mailClientDao) {  super.setPaginationDao(mailClientDao);  } |

## Data Access Layer (DAO)

DAO bean load the raw data from database using JPA entity manager. First, we define our DAO with @Repository annotation and extends PaginationDaoDbEntityManagerAbstract<PgMailMessage, BoMailSearchParam> from Pagination Framework:

|  |
| --- |
| @Repository  public class MailClientDao extends PaginationDaoDbEntityManagerAbstract<PgMailMessage, BoMailSearchParam> { |

PgMailMessage is the database entity bean and BoMailSearchParam is the incoming page parameters.

First, we will setup query to load data from database:

|  |
| --- |
| private static final String EJBQL = "select distinct a, b " +  "from PgMailMessage a " +  "join a.fromPgPerson b " +  "join a.toPgPerson c " +  "where a.isSpam = 'N' ";  private static final String[] RESTRICTIONS = {  "c.name = :toCustomized",  "b.name = :from",  "lower(a.subject) LIKE :subjectCustomized",  "a.sendDate >= :sendDateFromCustomized",  "a.sendDate <= :sendDateToCustomized",  "a.isRead = :isReadCustomized"  };  public MailClientDao() {  this.ejbql = EJBQL;  this.restrictions = Arrays.asList(RESTRICTIONS);  } |

Let’s look at our query. We select distinct PgMailMessage (alias a) in order to pass this object to assignDataToBo method in Service bean. In order for the query to order by “From” field (see assignColumnsDefinition in Service bean), we also select fromPgPerson (alias b) in our query. Moreover, we will define toPgPerson (alias c) and in where clause we restrict our query for non-spam email (isSpam=’N’).

Restrictions define our search options. There are 2 kinds of restrictions binding: 1) binding directly to variable in pparam (BoMailSearchParam) and 2) binding via customization.

The second restriction b.name=:from means that fromPgPerson.name are binding directly to variable “from” in pparam. Other restrictions are defined in binding in customizeRestrictions method:

|  |
| --- |
| @Override  public void customizeRestrictions(Map<String, Object> queryParameters, BoMailSearchParam pparam) throws Exception {  queryParameters.put("toCustomized", "Christine Kendrick"); // in reality, this should be the login user id.  if ( StringUtils.isEmpty(pparam.getSubject()) ) {  queryParameters.put("subjectCustomized", null);  } else {  queryParameters.put("subjectCustomized", "%"+pparam.getSubject().toLowerCase()+"%");  }  if ( StringUtils.isEmpty(pparam.getFromDate()) ) {  queryParameters.put("sendDateFromCustomized", null);  } else {  queryParameters.put("sendDateFromCustomized", getDateStart(formatter.parse(pparam.getFromDate())));  }  if ( StringUtils.isEmpty(pparam.getToDate()) ) {  queryParameters.put("sendDateToCustomized", null);  } else {  queryParameters.put("sendDateToCustomized", getDateEnd(formatter.parse(pparam.getToDate())));  }  if ( "U".equals(pparam.getStatus() ) ) {  queryParameters.put("isReadCustomized", "N");  } else if ( "R".equals(pparam.getStatus()) ) {  queryParameters.put("isReadCustomized", "Y");  } else {  queryParameters.put("isReadCustomized", null);  }  } |

In this customizeRestrictions method, toCustomized, subjectCustomized, sendDateFromCustomized, sendDateToCustomized and isReadCustomized are customized and then put into a map queryParameters. Our Pagination Framework will use this map to bind variables into our JPA query.

Finally, we need to inject entityManager via @PersistenceContext annotation:

|  |
| --- |
| @PersistenceContext  public void setEntityManager(EntityManager entityManager) {  super.setEntityManager(entityManager);  } |

In this example, we used JPA Entity Manager (PaginationDaoDbEntityManagerAbstract) for persistence management. There are 2 more options available in our Pagination Framework, one is PaginationDaoDbHibernateTemplateAbstract which uses spring hibernate template and the other is PaginationDaoLuceneAbstract which uses lucene index as datasource.

## Spring Framework Setup

To setup spring framework, we setup spring MVC in web.xml as we normally would:

|  |
| --- |
| <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/applicationContext.xml</param-value>  </context-param>  <listener>  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  </listener>  <servlet>  <servlet-name>dispatcher</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <load-on-startup>2</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>dispatcher</servlet-name>  <url-pattern>/org/pagination/example/mailclient/\*</url-pattern>  </servlet-mapping> |

In applicationContext.xml file, define the following for our annotated components:

|  |
| --- |
| <context:annotation-config />  <context:component-scan base-package="com.github.paginationspring.example.mailclient" />  <mvc:annotation-driven /> |

You should also define EntityManagerFactory for our JPA.

In dispatcher-servlet.xml, define the following for view resolver:

|  |
| --- |
| <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <property name="order" value="2" />  <property name="prefix" value="/WEB-INF/jsp/"/>  <property name="suffix" value=".jsp"/>  </bean> |

## Running our example

Download this zip file and extract the content into your computer. The content included full source code used in this example. Open command prompt and type run.bat (for window OS) or ./run.sh (for linux OS). Choose your application server and wait around 30 seconds for the server to startup. After server started, go to url <http://localhost:8080/pagination-example-spring2>