Parancoe Meta-Framework

Reference Guide

2.x

www.parancoe.org

Copyright © 2008 Lucio Benfante

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Introduction iii
1. Starting a new project with Parancoe 1
1.1. Software requisites 1
1.2. Starting up your own project 1
1.3. Using the project with an IDE 1
1.4. Setting up your Maven environment 1
1.5. Adding a CRUD to your application 1
1.5.1. Add the entity 2
1.5.2. Add the DAO 2
1.5.3. Add the controller 3
1.5.4. Add the views (JSP) 4
1.5.5. Add the validation 6
1.6. Configure your database 7
1.7. Database data initialization 8
2. Persistence 10
2.1. Introduction 10
2.2. Persistent entities 10
2.3. Entity DAOs 12
2.3.1. Define a DAO for an entity 12
2.3.2. Use the DAO 13
2.4. Add finder methods to the DAOs 13
2.4.1. Find by field equality 14
2.4.2. Ordering results 14
2.4.3. Find a single object 14
2.4.4. Write the query in JPA-QL/HQL 15
2.4.5. Methods with support for the pagination of results 15
2.4.6. Using different strategies for comparing parameter values 16
2.5. Search with criteria 16
2.6. Transaction demarcation 17
3. Parancoe Plugins 18
3.1. Introduction 18
3.1. DWR 18
3.1.1. How to add the plugin to your application 18
3.1.2. A simple example 19
3.2. Italy 22
3.2.1. How to add the plugin to your application 22
3.2.2. A simple example 23

Introduction

Parancoe is a project aiming to simplify the release of web applications promoting the convention over configuration philosophy and the DRY principle. This project is promoted by the JUG Padova, and everybody can partecipate.

Parancoe is a Java meta-framework aggregating in an useful way Hibernate/JPA, Spring 2, Spring MVC and, for the AJAX support, DWR.

Parancoe purpose is to give to developers a set of libraries ready to build standard web applications (which in most cases are just crud applications) without worrying of long and harmful configurations files. Parancoe will be composed of a full MVC stack.

Parancoe is open source and is released under the Apache License, Version 2.0.

Are you interested in the parancoe word meaning? A “parancoa” is a scaffolding in the dialect of the Venice (Italy) area.

Chapter 1. Starting a new project with Parancoe

1.1. Software requisites

For developing with Parancore you'll need:

- Java SDK 5+
- Maven 2.0.9+
- An IDE (NetBeans, Eclipse, etc.)

1.2. Starting up your own project

You can start the development of a new Web application using the Parancoe Web Application archetype:

```
mvn archetype:create -DarchetypeGroupId=org.parancoe \
    -DarchetypeArtifactId=parancoe-webarchetype \
    -DarchetypeVersion=2.0.3 \
    -DgroupId=com.mycompany \
    -DartifactId=testApp \
    -DpackageName=com.mycompany.testapp
```

Of course you can personalize the groupId, artifactId and package. After the execution you’ll have a complete web project with a common good layout, configured authentication and authorization, some Parancoe examples. Starting from this skeleton you can personalize it for developing your own application.

1.3. Using the project with an IDE

The Parancoe web application archetype produces a Maven 2 project. Of course you can use it with your preferred IDE.

- NetBeans: install the Mevenide2 plug-in and use the project as a native NetBeans project. You'll find this plugin already available in the NetBeans plugin link starting from NetBeans 6.0.
- Eclipse: run `mvn eclipse:eclipse` and open the project with Eclipse.
- IntelliJ IDEA: run `mvn idea:idea` and open the project with IDEA.

1.4. Setting up your Maven environment

The easyest way to develop a project with Parancoe is to use Maven 2 as build tool.

Parancoe artifacts are distributed through the Maven Central Repository and its mirrors. You don’t have to configure anything for using this repository.

If you are behind a proxy, check the Maven guide to using proxies to configure maven 2 with your proxy.

1.5. Adding a CRUD to your application

In the following `$PRJ` indicates the directory in wich you created the project.

1.5.1. Add the entity

Add to the `$PRJ/src/main/java/com/mycompany/testapp/po` directory the `Person.java` file with the following content:

```
package com.mycompany.testapp.po;

import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import org.parancoe.persistence.po.hibernate.EntityBase;

@Entity
public class Person extends EntityBase {

    private String firstName;
    private String lastName;
    private Date birthDate;
    private String email;

    @Temporal(TemporalType.DATE)
    public Date getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

1.5.2. Add the DAO

Add to the `$PRJ/src/main/java/com/mycompany/testapp/dao` directory the `PersonDao.java` file with the following content:

```
package com.mycompany.testapp.dao;

import com.mycompany.testapp.po.Person;
```

```
import org.parancoe.persistence.dao.generic.Dao;
import org.parancoe.persistence.dao.generic.GenericDao;

@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
}
```

1.5.3. Add the controller

Add to the `$PRJ/src/main/java/com/mycompany/testapp/controllers` directory the `PersonController.java` file with the following content:

```
package com.mycompany.testapp.controllers;

import com.mycompany.testapp.dao.PersonDao;
import com.mycompany.testapp.po.Person;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

@Controller
@RequestMapping({"/person/*.form", "/person/*.html"})
@SessionAttributes(PersonController.PERSON_ATTR_NAME)
public class PersonController {

    public static final String PERSON_ATTR_NAME = "person";
    public static final String EDIT_VIEW = "person/edit";
    public static final String LIST_VIEW = "person/list";
    @Autowired
    private PersonDao personDao;

    @RequestMapping
    public String edit(@RequestParam("id") Long id, Model model) {
        Person person = personDao.get(id);
        if (person == null) {
            throw new RuntimeException("Person not found");
        }
        model.addAttribute(PERSON_ATTR_NAME, person);
        return EDIT_VIEW;
    }

    @RequestMapping
    public String save(@ModelAttribute(PERSON_ATTR_NAME) Person person,
            BindingResult result, SessionStatus status) {
        personDao.store(person);
        status.setComplete();
        return "redirect:list.html";
    }

    @RequestMapping
    public String list(Model model) {
        List<Person> people = personDao.findAll();
        model.addAttribute("people", people);
        return LIST_VIEW;
    }

    @RequestMapping
    public String create(Model model) {
```

```
        Person person = new Person();
        model.addAttribute(PERSON_ATTR_NAME, person);
        return EDIT_VIEW;
    }

    @RequestMapping
    public String delete(@RequestParam("id") Long id, Model model) {
        Person person = personDao.get(id);
        if (person == null) {
            throw new RuntimeException("Person not found");
        }
        personDao.delete(person);
        return "redirect:list.html";
    }
}
```

1.5.4. Add the views (JSP)

Create the `$PRJ/src/main/webapp/WEB-INF/jsp/person` directory.

Add to the `$PRJ/src/main/webapp/WEB-INF/jsp/person` directory the `list.jsp` file with the following content:

```
<%@ include file="../common.jspf" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <%@ include file="../head.jspf" %>
    </head>
    <body>
        <div id="nonFooter">
            <jsp:include page="../header.jsp"/>
            <div id="content">
                <div id="content_main">
                    <h1>People</h1>
                    <table>
                        <c:forEach var="person" items="${people}">
                            <tr>
                                <td>${person.firstName}</td>
                                <td>${person.lastName}</td>
                                <td>${person.birthDate}</td>
                                <td>${person.email}</td>
                                <td>
                                    <a href="edit.form?id=${person.id}">Edit</a>
                                    <a href="delete.html?id=${person.id}">Delete</a>
                                </td>
                            </tr>
                        </c:forEach>
                    </table>
                    <c:if test="${empty people}">
                        No people in the DB
                    </c:if>
                    <a href="create.html">New</a>
                </div>
                <jsp:include page="../menu.jsp"/>
            </div>
        </div>
        <jsp:include page="../footer.jsp"/>
    </body>
</html>
```

Add to the `$PRJ/src/main/webapp/WEB-INF/jsp/person` directory the `edit.jsp` file with the following content:

```
<%@ include file="../common.jspf" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <%@ include file="../head.jspf" %>
    </head>
    <body>
        <div id="nonFooter">
            <jsp:include page="../header.jsp"/>
            <div id="content">
                <div id="content_main">
                    <h1>Edit person</h1>
                    <form:form commandName="person" method="POST"
                            action="${cp}/person/save.form">
                        <table>
                        <tr>
                            <td>First name:</td>
                            <td><form:input path="firstName"/></td>
                        </tr>
                        <tr>
                            <td>Last name:</td>
                            <td><form:input path="lastName"/></td>
                        </tr>
                        <tr>
                            <td>Birth date:</td>
                            <td><form:input path="birthDate"/> (dd/MM/yyyy)</td>
                        </tr>
                        <tr>
                            <td>E-mail:</td>
                            <td><form:input path="email"/></td>
                        </tr>
                        <tr>
                            <td> </td>
                            <td><input type="submit" value="Submit"/><br/><br/></td>
                        </tr>
                        </table>
                        <form:errors path="*" cssClass="errorBox"/>
                    </form:form>
                </div>
                <jsp:include page="../menu.jsp"/>
            </div>
        </div>
        <jsp:include page="../footer.jsp"/>
    </body>
</html>
```

Add an item to the page menu in the `$PRJ/src/main/webapp/WEB-INF/jsp/menu.jsp` file:

```
<%@ include file="common.jspf" %>
<div id="content_menu">
    <p class="menuTitle"><spring:message code="label_menu"/></p>
    <p class="menuLevel0"><a href="${cp}/"><spring:message code="menu_home"/></a></p>
    <p class="menuLevel0"><a href="${cp}/person/list.html">People</a></p>
    <p class="menuLevel0">
        <a href="${cp}/admin/index.html"><spring:message code="menu_administration"/></a>
    </p>
    <authz:authorize ifAnyGranted="ROLE_ADMIN,ROLE_PARANCOE">
        <p class="menuLevel0"><a href="${cp}/logout.secure">Logout</a></p>
    </authz:authorize>
</div>
```

```
```

1.5.5. Add the validation

Add the validation rules to your form bean (in this example it's also the persistence entity) in the `$PRJ/src/main/java/com/mycompany/testapp/po/Person.java`. The rules are expressed through annotations.

```
package com.mycompany.testapp.po;

import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import org.parancoe.persistence.po.hibernate.EntityBase;
import org.springmodules.validation.bean.conf.loader.annotation.handler.Email;
import org.springmodules.validation.bean.conf.loader.annotation.handler.NotBlank;

@Entity
public class Person extends EntityBase {

    @NotBlank
    private String firstName;
    @NotBlank
    private String lastName;
    private Date birthDate;
    @Email
    private String email;

    @Temporal(TemporalType.DATE)
    public Date getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Add the `@Validation` annotation to the `save` method of the `PersonController` class:

```
@RequestMapping
@Validation(view=EDIT_VIEW)
public String save(@ModelAttribute(PERSON_ATTR_NAME) Person person,
        BindingResult result, SessionStatus status) {
    personDao.store(person);
    status.setComplete();
    return "redirect:list.html";
}
```

1.6. Configure your database

The application generated by the Parancoe Web Application has its database configuration in the `$PRJ/src/main/webapp/WEB-INF/database.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/testAppDS"/>
  </bean>

  <bean id="sessionFactory" parent="abstractSessionFactory">
    <property name="hibernateProperties">
      <props merge="true">
        <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
        <prop key="hibernate.show_sql">true</prop>
      </props>
    </property>
    <property name="eventListeners">
      <map>
        <entry key="merge">
          <bean class=
   "org.springframework.orm.hibernate3.support.IdTransferringMergeEventListener"/>
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

As you see in the previous file, the database connection is configured through a datasource, so you just need to define a datasource with that name in your application server. If you are using Tomcat, you can define the datasource in the `$PRJ/src/main/webapp/META-INF/context.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/testApp" reloadable="true">
  <Resource name="jdbc/testAppDS"
            type="javax.sql.DataSource"
            auth="Container"
            driverClassName="org.hsqldb.jdbcDriver"
            url="jdbc:hsqldb:mem:testApp"
            username="sa"
            password=""
            maxActive="5"
            />
</Context>
```

For ease of configuration, the default database is an HSQL in-memory database. Of course you need to change it to a database with real persistence.

When you change your database, you need to change your datasource and the Hibernate dialect (if you aren't using an HSQL database server).

1.7. Database data initialization

A common case of every application is the initialization of the database with the data needed for putting the application itself in a proper initial state after its first deploy.

For example the application generated by the Parancoe Web Archetype populate the database with the `ROLE_PARANCOE` and `ROLE_ADMIN` roles, and the `parancoe` and `admin` users. This is achieved adding the entities that must be initialized to the `clazzToPopulate` list of the `PopulateInitialDataContextListener` class:

```
package com.mycompany.testapp;

import com.mycompany.testapp.po.Person;
import org.parancoe.plugins.security.Authority;
import org.parancoe.plugins.security.User;

public class PopulateInitialDataContextListener
        extends org.parancoe.web.PopulateInitialDataContextListener {

    public PopulateInitialDataContextListener() {
        // Add here to the clazzToPopulate collection
        // the entity classes you need to populate
        clazzToPopulate.add(Authority.class);
        clazzToPopulate.add(User.class);
    }

}
```

The data to load are defined in the `$PRJ/src/main/resources/initialData` directory by a set of "fixtures" in YAML format:

Example 1.1. Authority.yml

```
- &Authority-parancoe
  role: ROLE_PARANCOE
- &Authority-admin
  role: ROLE_ADMIN
```

Example 1.2. User.yml

```
- &User-parancoe
  username: parancoe
  # password : parancoe
  password: d928b1a8468c96804da6fcc70bff826f
  enabled: true
  authorities:
    - *Authority-parancoe
- &User-admin
  username: admin
  # password : admin
  password: ceb4f32325eda6142bd65215f4c0f371
  enabled: true
  authorities:
    - *Authority-admin
```

As you can see, it's very easy to define data sets, even with references to other data. The main rules are:

1. Put your data in a file named `<entity_name>.yml`
2. Identify your data items with `&<entity_name>-<some-useful-identifier>`
3. Use `*<entity_name>-<some-useful-identifier>` for referencing that item;

Chapter 2. Persistence

2.1. Introduction

The persistence layer of Parancoe is based on the Hibernate implementation of the Java Persistence API (JPA). Moreover it provides:

- utility classes for easy definition of persistent entities
- a DAO system for common methods and for easy definition of finder methods
- a standard default configuration

2.2. Persistent entities

The recommended method for creating a persistent entity with Parancoe is to use the JPA annotations. As at present the JPA implementation used by Parancoe is the Hibernate one, you can read the Hibernate documentation for a complete reference about the JPA annotations.

Parancoe provides the `org.parancoe.persistence.po.hibernate.EntityBase` class, that can be used as the base class for your entities. The features inherited from the `EntityBase` are:

- an auto-generated identifier of type `Long`;
- versioning for optimistic locking;
- properly defined `hashCode` and `equals` methods;
- `toString` method, printing the class name and the identifier of the persistent instance.

The following code is an example of defining an entity class extending the `EntityBase`:

Example 2.1. The Person entity

```
package org.parancoe.example.po;

import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;


                    @Entity
public class Person
                    extends EntityBase {
    private String firstName;
    private String lastName;
    private Date birthDate;
    private String email;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

                    @Temporal(TemporalType.DATE)
    public Date getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

The entities are auto-discovered using the following tag in your Spring configuration (already included in the configuration produced by the Parancoe Web Archetype):

```
    <parancoe:discover-persistent-classes basePackage="org.parancoe.example.po"/>
```

So the only thing you need to do is to add the entities classes in the correct package. No per-class configuration is needed.

2.3. Entity DAOs

For using the persistent entities you need to write almost no code. Parancoe provides a Data Access Object (DAO) layer very powerful and easy to use.

2.3.1. Define a DAO for an entity

In Parancoe the DAOs are not classes, just interfaces, without an explicit implementation. For example, for defining the DAO for the Person entity, you need to write:

Example 2.2. The Person DAO

```
package org.parancoe.example.dao;

import org.parancoe.example.po.Person;
import org.parancoe.persistence.dao.generic.Dao;
import org.parancoe.persistence.dao.generic.GenericDao;

@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
}
```

The generic parameters of the `GenericDao` interface are the type of the persistent entity, and the type of its identifier.

You don't have to write an implementation of this interface. The implementation will be automatically added at runtime through Spring AOP. The current implementation provides the following methods to each DAO:

```
T read(PK id);
T get(PK id);
void create(T transientObject);
void store(T transientObject);
void delete(T persistentObject);
List<T> findAll();
List<T> searchByCriteria(Criterion... criterion);
List<T> searchByCriteria(DetachedCriteria criteria);
List<T> searchByCriteria(DetachedCriteria criteria, int firstResult, int maxResults);
Page<T> searchPaginatedByCriteria(int page, int pageSize, Criterion... criterion);
Page<T> searchPaginatedByCriteria(int page, int pageSize, DetachedCriteria criteria);
int deleteAll();
long count();
long countByCriteria(DetachedCriteria criteria);
```

Note that even these methods are parametrized with the type of the entity (`T`) and the type of the entity identifier (`PK`). So, for example, the `findAll` method of the DAO of the `Person` entity doesn't return a generic `List<Object>`, but it can return a more usefull type-checked `List<Person>`.

The DAOs are auto-discovered using the following tag in your Spring configuration (already included in the configuration produced by the Parancoe Web Archetype):

```
    <parancoe:define-daos basePackage="org.parancoe.example.dao"/>
```

So the only thing you need to do is to add the DAO interfaces classes in the correct package. No per-DAO configuration is needed.

2.3.2. Use the DAO

To use a DAO you simply need to get it from the Spring application context. You could of course retrieve it by name from the context, using its unqualified name. For example you could retrieve the `PersonDao` throught the "personDao" name. But the easiest way for obtaining a reference to a DAO is to auto-wire it in a Spring managed bean.

For example, for using the `PersonDao` in a controller of your application:

Example 2.3. A controller using a DAO

```
package org.parancoe.example.controllers;

import org.parancoe.example.dao.PersonDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/people/*.html")
public class PeopleController {

                        @Autowired
    private PersonDao personDao;


    @RequestMapping
    public String list(Model model) {
        model.addAttribute("people",
                        personDao.findAll());
        return "people/list";
    }
}
```

2.4. Add finder methods to the DAOs

The `GenericDao` interface provides the base methods you need for interacting with persistent entities. You will add to your DAO interface the methods your application needs for finding the data in the database. Again, you don't have to implements those methods, they will be solved at runtime through Spring AOP, using some simple rules:

1. if it exists a *named query* with name `<entityName>.methodName`, then execute that query and return the result as the result of the method;
2. else if the *method name* is in the form `findBy[<And-separated list of attributes>][OrderBy[<And-separated list of attributes>]`, then the method signature is parsed for generating a query. The generated query will be executed and the result will be returned as the result of the method;
3. else call the method on the base DAO implementation.

The return type of the finder methods can be a `List<T>` or a single `T`, where `T` is the type of the persistent entity. In the case of single `T`:

- if the query will produce a single result, that result will be returned;

- if the query will produce more than a single result, only the first one will be returned;
- if the query will produce no results, `null will be returned.`

Some pratical examples will help you being very productive with this technique.

2.4.1. Find by field equality

For example, you need to find all people with first name equals to "John". Add to the interface the following method:

```
@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
  List<Person> findByFirstName(String firstName);
}
```

If the finder needs to compare on more fields, simply list them in the method name separated by the “And” keyword:

```
@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
  List<Person> findByFirstName(String firstName);
  List<Person> findByFirstNameAndLastNameAndBirthDate(
      String firstName, String lastName, Date birthDate);
}
```

2.4.2. Ordering results

If you need to order (ascending) the results, declare a method as the following:

```
@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
  List<Person> findByBirthDateOrderByLastName(Date birthDate);
}
```

Again, If you need to order on more than one field, list the fields you need:

```
List<Person> findByBirthDateOrderByLastNameAndFirstName(Date birthDate);
```

If you don’t want to filter by equality, but the result must contains all records, omit the list of fields:

```
List<Person> findByOrderByLastNameAndFirstName();
```

2.4.3. Find a single object

Define the finder method returning the entity type, not a List:

```
Person findByFirstNameAndLastName(String firstName, String lastName);
```

If the query will produce more than one result, only the first one in the list will be returned. If the query will produce no results, the method will return null.

2.4.4. Write the query in JPA-QL/HQL

You can write an JPA-QL/HQL query and add a method in the DAO interface that will execute it when invoked. Declare the method as you like in the DAO interface, and declare the query as a named query in the DAO's entity. The name of the query must be `<entityName>.methodName`. The method parameters are passed in the same order as values for the query parameters. For example:

```
@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
  List<Person> findByPartialUncasedLastName(String partialLastName);
}
```

```
@Entity()
@NamedQueries({
  @NamedQuery(
    name="Person.findByPartialUncasedLastName",
    query="from Person p where lower(p.lastName) like lower(?) order by p.lastName")
})
public class Person extends EntityBase {
  // ...
}
```

Now You can invoke it:

```
List <Person> people = dao.findByPartialUncasedLastName("B%");
```

2.4.5. Methods with support for the pagination of results

You can annotate the finder method parameters with the `@FirstResult` and `@MaxResults` annotations. For example:

```
@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
  List<Person> findByLastName(String lastName,
                              @FirstResult int firstResult,
                              @MaxResults int maxResults);
}
```

The type of the annotate parameter must be int. You can apply this technique to finder methods associated to named queries, without the need to modify the query. For example:

```
@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
  List<Person> findByPartialUncasedLastName(String partialLastName,
                                            @FirstResult int firstResult,
                                            @MaxResults maxResults);
}
```

```
@Entity()
@NamedQueries({
  @NamedQuery(
    name="Person.findByPartialUncasedLastName",
    query="from Person p where lower(p.lastName) like lower(?) order by p.lastName")
})
public class Person extends EntityBase {
  // ...
```

```
}
```

If there are situations in which you don't want to limit the number of records, you can pass a negative value for the *maxRecords* parameter:

```
List<Person> people = dao.findByPartialUncasedLastName("B%", 0, -1);
```

It will return all records, from the first (0) to the end.

2.4.6. Using different strategies for comparing parameter values

In totally dynamic finders you can choose the type of the comparison applied to each parameter using the `@Compare` annotation. For example:

```
@Dao(entity=Person.class)
public interface PersonDao extends GenericDao<Person, Long> {
  List<Person> findByLastName(
                 @Compare(CompareType.ILIKE) String lastName);
}
```

The supported comparison strategies are:

EQUAL
: equality, it's the default

LIKE
: like, using '%'

ILIKE
: case-insensitive like

GE
: greater than or equal

GT
: greater than

LE
: less than or equal

LT
: less than

NE
: not equal

2.5. Search with criteria

Very complex queries can be implemented using criteria in place of writing JPA-QL/HQL queries. This is particularly useful for implementing search forms, where some or all parameters are optional.

The methods of the base DAO that support this kind of queries are:

```
List<T> searchByCriteria(Criterion... criterion);
List<T> searchByCriteria(DetachedCriteria criteria);
List<T> searchByCriteria(DetachedCriteria criteria, int firstResult, int maxResults);
Page<T> searchPaginatedByCriteria(int page, int pageSize, Criterion... criterion);
Page<T> searchPaginatedByCriteria(int page, int pageSize, DetachedCriteria criteria);
long countByCriteria(DetachedCriteria criteria);
```

For example:

Example 2.4. Searching using a `DetachedCriteria`

```
DetachedCriteria personCriteria =
        DetachedCriteria.forClass(Person.class);
if (StringUtils.isNotBlank(firstNameValue)) {
    personCriteria.add(Restrictions.ilike("firstName",
            firstNameValue, MatchMode.ANYWHERE));
}
if (StringUtils.isNotBlank(lastNameValue)) {
    personCriteria.add(Restrictions.ilike("lastName",
            lastNameValue, MatchMode.ANYWHERE));
}
List<Person> result = personDao.searchByCriteria(personCriteria);
```

2.6. Transaction demarcation

Without any additional configuration the transaction is identified by the single DAO method call.

The application generated by the Parancoe Web Archetype is configured for using a transaction-per-request strategy.

The transaction will be commited at the end of the request, except if an uncaught exception has been thrown.

You can rollback the current transaction calling the `rollbackTransaction` method of any DAO.

Chapter 3. Parancoe Plugins

3.1. Introduction

Parancoe provide a plugin system for easily add features to your application. A Parancoe plugin is a JAR containing classes, the plugin configuration and the plugin definition. For adding a plugin to your Parancoe application, you simply have to put its JAR in your application classpath, or, if you are using Maven, to add the plugin in your application dependencies. The plugin will be automatically discovered, and its features will be ready to be used in your application.

At present the following plugins are available.

- DWR
- Italy
- Security Encrypted
- Spring Security
- World

3.1. DWR

This plugin provides an easy integration of the Direct Web Remoting (DWR) framework, version 3. DWR is a framework for easily implement AJAX functionalities in a Java application.

3.1.1. How to add the plugin to your application

For adding the DWR plugin to your application, if your are using Maven, simply add the plugin dependency to your `pom.xml`

Example 3.1. DWR plugin dependency

```
<dependency>
    <groupId>org.parancoe</groupId>
    <artifactId>parancoe-plugin-dwr</artifactId>
    <version>2.0.2</version>
</dependency>
```

The DWR controller is mapped on the `/dwr/*` URLs. So add that URL pattern to the mapping of the `parancoe` servlet in your deployment descriptor (`$PRJ/src/main/webapp/WEB-INF/web.xml`):

```

    <servlet-mapping>
        <servlet-name>parancoe</servlet-name>
        <url-pattern>/dwr/*</url-pattern>
    </servlet-mapping>

```

For testing if the plugin is correctly installed, build and deploy you application, and point your browser to the `<base_address>/dwr/index.html` address (your base address will be something similar to `http://localhost:8080/myapp`). You should see the following page:

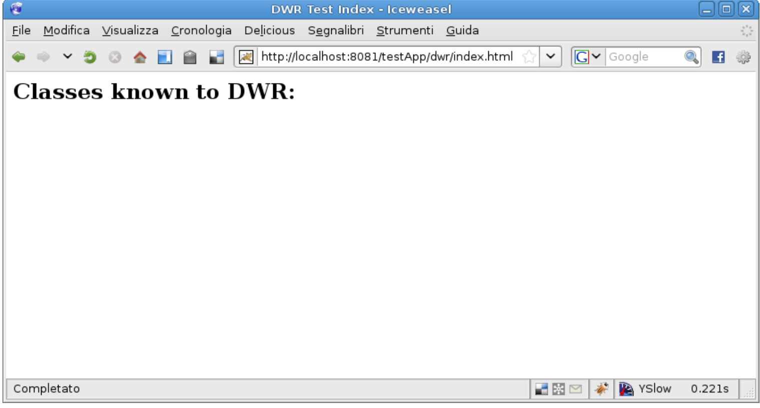

Figure 3.1. The empty DWR Test Page

Of course, just after the installation you'll have no classes known to DWR.

3.1.2. A simple example

As a simple example we will implement a search page for searching the users of your application. The page will contain a single text field for typing the partial username to search. Every second, if the content of the text field has changed, an AJAX call will search the user database, and will build a list of the results in the page (of course without the need of a full refresh of the page).

Let's start writing the Java server-side code that will produce the results:

Example 3.2. UserSearch.java

```
package com.mycompany.testapp.ajax;

import java.util.List;
import org.directwebremoting.annotations.RemoteMethod;
import org.directwebremoting.annotations.RemoteProxy;
import org.parancoe.plugins.security.User;
import org.parancoe.plugins.security.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
@RemoteProxy(name = "userSearch")
public class UserSearch {

    @Autowired
    private UserDao userDao;

    @RemoteMethod
    public String[] search(String partialUsername) {
        List<User> users =
                userDao.findByPartialUsername(partialUsername);
        String[] usernames = new String[users.size()];
        int i = 0;
        for (User user : users) {
            usernames[i++] = user.getUsername();
        }
        return usernames;
    }
}
```

Add the scanning of the package of this class in your `parancoe-servlet.xml` file, if it's not already included:

```
<context:component-scan base-package="com.mycompany.testapp.ajax"/>
```

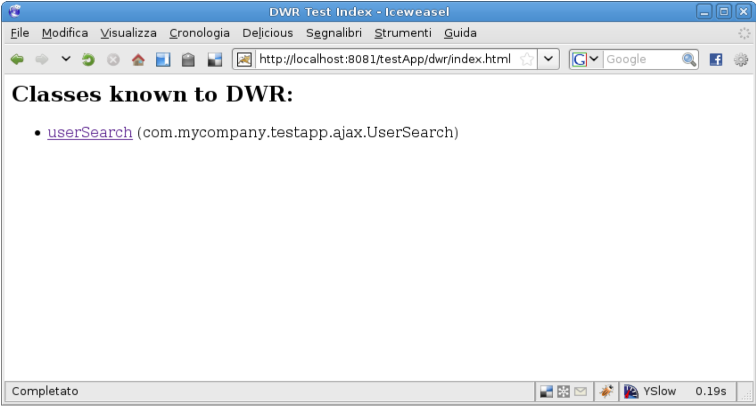

Figure 3.2. The not empty DWR Test Page

You can try to ajax-call your method in the class test page:

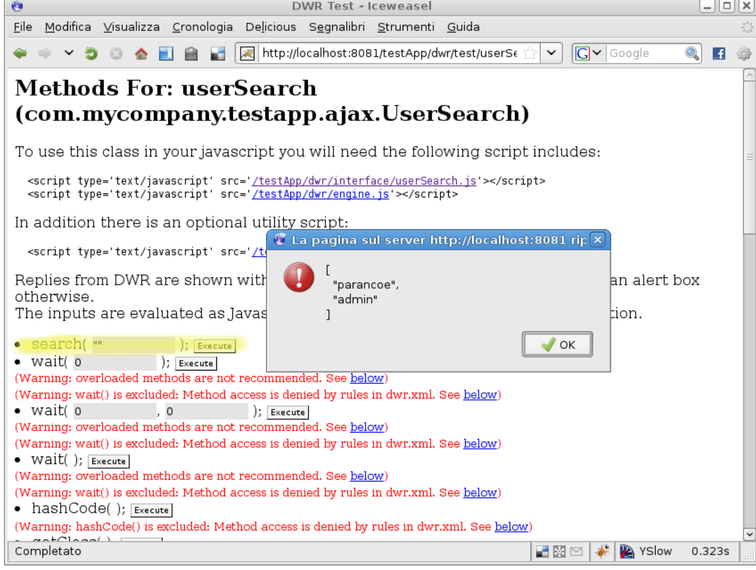

Figure 3.3. The UserSearch DWR Test Page

Now you can use that call in your pages. For example, add the `$PRJ/src/main/webapp/userSearch.jsp` page:

Example 3.3. userSearch.jsp

```
<%@ include file="WEB-INF/jsp/common.jspf" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <%@ include file="WEB-INF/jsp/head.jspf" %>
        <script src="${cp}/dwr/interface/userSearch.js" type="text/javascript"></script>
        <script src="${cp}/dwr/engine.js" type="text/javascript" ></script>
        <script src="${cp}/dwr/util.js" type="text/javascript"></script>
    </head>
    <body>
        <h1>Search Users</h1>

        <div>
            <form id="userSearchForm" action="" method="get">
                <label>Username:</label><input id="username" name="username" type="text"/>
            </form>
        </div>

        <div style="clear: both;"><ul id ="userSearch_result"></ul></div>

        <script type="text/javascript">
            new Form.Observer('userSearchForm', 1, function(el, value) {
                var formValues = value.parseQuery();
                userSearch.search(formValues.username, function(users) {
                    $('userSearch_result').update('');
                    users.each(function(user) {
                        $('userSearch_result').insert('<li>'+user+'</li>');
                    });
                });
            });
        </script>
    </body>
</html>
```

Pointing you browser to that page you'll see the desired result:

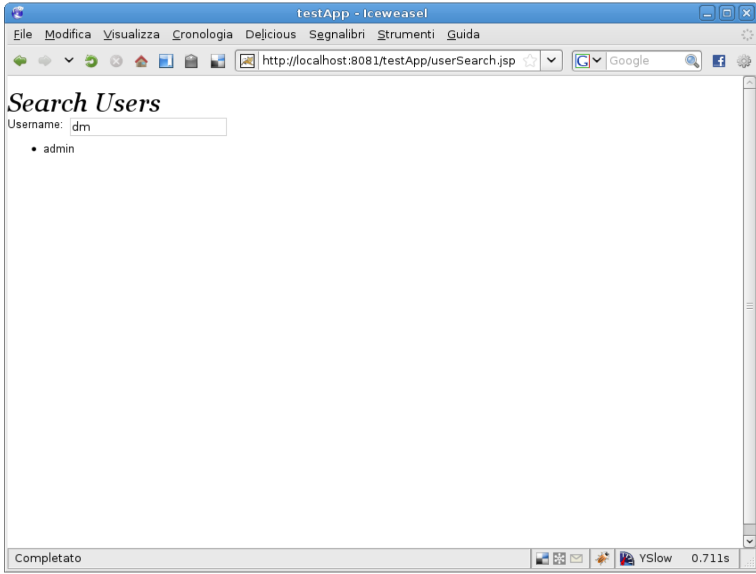

Figure 3.4. The userSearch.jsp page in action

Using directly the collection of objects, in place of array of strings, make the code even simpler.

Example 3.4. UserSearch.java returning a list of User objects

```
package com.mycompany.testapp.ajax;

import java.util.List;
import org.directwebremoting.annotations.RemoteMethod;
import org.directwebremoting.annotations.RemoteProxy;
import org.parancoe.plugins.security.User;
import org.parancoe.plugins.security.UserDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
@RemoteProxy(name = "userSearch")
public class UserSearch {

    @Autowired
    private UserDao userDao;

    @RemoteMethod
    public List<User> search(String partialUsername) {
        return userDao.findByPartialUsername(partialUsername);
    }
}
```

The code in the JSP doesn't almost change. Update just the single line where you expose the data in the page, accessing the username attribute of the JavaScript object:

```
$('userSearch_result').insert('<li>'+user.username+'</li>');
```

Finally, for passing User objects, you need to declare it in your DWR configuration in your `parancoe-servlet.xml` file:

```
<dwr:configuration>
    <dwr:convert type="bean" class="org.parancoe.plugins.security.User" />
    <dwr:convert type="bean" class="org.parancoe.plugins.security.Authority" />
</dwr:configuration>
```

When you'll put the application in production, probably you'll want the DWR Test Page being accessible no more. For disabling it, override the DWR controller in your `parancoe-servlet.xml`, setting the `debug` attribute to `false`:

```
<dwr:controller id="dwrController" debug="false">
```

3.2. Italy

This plugin provides the data of the italian regions, provinces and municipalities.

3.2.1. How to add the plugin to your application

For adding the Italy plugin to your application, if your are using Maven, simply add the plugin dependency to your `pom.xml`

Example 3.5. Italy plugin dependency

```
<dependency>
    <groupId>org.parancoe</groupId>
    <artifactId>parancoe-plugin-italy</artifactId>
    <version>2.0.1</version>
</dependency>
```

3.2.2. A simple example

The plugin provides persistent classes and a DAOs for accessing the italian data. For example, for retrieving the list of all italian regions:

Example 3.6. A class using the RegioneDao

```
package com.mycompany.testapp.blo;

import java.util.List;
import javax.annotation.Resource;
import org.parancoe.plugins.italy.Regione;
import org.parancoe.plugins.italy.RegioneDao;
import org.springframework.stereotype.Component;

@Component
public class RegionalBusiness {

    @Resource
    private RegioneDao regioneDao;

    public List<Regione> retriveAllRegions() {
        return regioneDao.findAll();
    }
}
```

For checking if the plugin is working, you can write a simple test of the previous class:

Example 3.7. A test for the class of the example

```
package com.mycompany.testapp.blo;

import java.util.List;
import org.parancoe.plugins.italy.Comune;
import org.parancoe.plugins.italy.Procura;
import org.parancoe.plugins.italy.Provincia;
import org.parancoe.plugins.italy.Regione;
import org.parancoe.web.test.BaseTest;
import org.springframework.beans.factory.annotation.Autowired;

public class RegionalBusinessTest extends BaseTest {

    @Autowired
    private RegionalBusiness regionalBusiness;

    public void testRetriveAllRegions() {
        List<Regione> result = regionalBusiness.retriveAllRegions();
        assertSize(20, result);
    }

    @Override
    public Class[] getFixtureClasses() {
        return new Class[]{Regione.class, Provincia.class, Comune.class,
                    Procura.class
                };
    }
}
```

3.1. Security Encrypted

TBW

3.2. Spring Security

TBW

3.3. World

TBW