CocoonAndHibernateTutorial

Tutorial on using Hibernate with Cocoon

- TARGET-AUDIENCE: beginner ***advanced*** expert
- COCOON-RELEASES: 2.1.5 2.1.6 2.1.7 2.1.10
- DOCUMENT-STATUS: ***draft*** reviewed released

Recent changes

2007/05/28:

Completely reworked most parts of the implementation, following important insights from a discussion on the users list. Sessions now have no longer to be opened in flowscript any more, in fact, the dependency between flowscript and hibernate is effectively remove. The new OpenSessionInView Filter now works analogously as described on http://www.hibernate.org/43.html.

Everything is also migrated to Cocoon 2.1.10 and Hibernate 3. For the Impatient, I created a hibernate-interface.jar with all required classes and Hibernate configuration files and a languages.zip with a minimal CRUD Demo application (not much more than a Rails scaffold).

What you will get from this page

This page proposes a way of using Hibernate from your Cocoon application. This is **not** an introduction to Hibernate or Cocoon itself. It just covers the specific issues of integration between the two.

Database connections are drawn from Cocoon's connection pool. I strongly recommend using the DAO (Data Access Object) pattern as an additional layer of abstraction, however if you wish to do so, you can also access Hibernate directly from flowscript or Java Flow.

Alternative approach

You should also consider the frequently used Cocoon / Hibernate / Spring approach, which adds additional abstraction and supports the IOC patterns. A starting point is the "Spring Petstore" at http://new.cocoondev.org/main/117-cd/43-cd.html .

Topics covered

The two main problems you have when integrating Hibernate into Cocoon are the following:

- Hibernate's builtin connection pool is for testing purposes only, so sooner or later you will have to use another one. (At least this was true for earlier versions of Hibernate; am not sure if this is still an issue).
- When persistent objects are navigated by the view, closing Hibernate sessions from flowscript (=the control) may lead to exceptions due to race conditions.

The approach presented here addresses these problems by covering the following topics:

- Installing Hibernate
- Configuring Hibernate to use cocoon.xconf datasources
- Setting up a Servlet Filter for Hibernate session management (a.k.a. "Open Session in View" Filter)

Your basic skills

Time. If you are new to the topic of O/R mapping, you will need a lot of time and patience, because it is not an easy topic. Learning Cocoon already made you re-think your concept of a Web Application; learning Hibernate, you will have to do it again.

You should have basic Cocoon knowledge, i.e. about the Sitemap, Flowscript, and JX Templates. The samples included will also use CForms. Did I mention Flowscript ? That one is especially important.

If you want to use Hibernate you should also be fluent in Java. Be warned that you are going beyond the "no programming required" - statement sometimes associated with Cocoon. If you want to understand what you are doing, you should have a basic notion of what Avalon is. We are also going to write a Servlet Filter which takes care of Hibernate Session, so if you don't know what a Servlet Filter is get your favourite book or search engine and read up 🙂

You will have a basic notion of what Hibernate is and what it does, otherwise you would not have come to this page 🙂 But make sure you have understood what lazy collection initialization is and what it does. If you can get a copy of the "Hibernate in Action" book, go ahead. It is a little old now, but still incredibly useful.

Technical prerequisites

A running cocoon installation. This was written for and tested on Cocoon 2.1.10, but should also work with earlier versions down to 2.1.6 at least (which the first version of this tutorial was based on).

Conceptual overview

Again, before you begin, consider using the Cocoon / Hibernate / Spring approach. Unlike in earlier versions, the approach presented here is now equally "clean" as this solution, but you might want to use Spring anyway, and the upcoming versions of Cocoon will be based on Spring instead of Avalon.

The philosophy of this approach is to hide Hibernate from the flowscript layer as much as possible. This means that a layer of DAOs (Data Access Objects) will be used from flowscript instead of directly interacting with Hibernate. DAO is a well-established pattern, you can read more about it at http://www.hibernate.org/328.html . On that page, you can also see how elegantly DAO can be implemented with Java 5. However, here we use Java 1.4 compatible code.

We will use Cocoon's builtin connection pool to supply JDBC connections to Hibernate. A few "tricks" are required to tie DAOs, Cocoon and the Servlet Filter together, but this work has already been done for you.

Setting up

First of all, you need an active database connection set up and configured in cocoon.xconf. There are several wiki and documentation pages on how to do this. For instance, if you use MySQL have a look at MySQL.

For now, I will assume that your SQL datasource is named "hibernate-datasource".

Now download hibernate from http://www.hibernate.org. As I'm writing this, 3.3 is the current production release. The following jars from the Hibernate distribution need to be copied to your WEB-INF/lib directory:

- hibernate3.jar

From the "lib" directory of the Hibernate distribution:

- dom4j-1.4.jar
- cglib-full-2.0.2.jar
- jta.jar
- asm.jar
- asm-attrs.jar

Note: "asm.jar" and "asm-attrs.jar" may already be present in the WEB-INF/lib directory. At least in Cocoon 2.1.7, these versions were not compatible with Hibernate 3. The only option was to replace them with those shipped with Hibernate. I can not guarantee that Cocoon does not rely on the particular version it ships with, though.

Someone reported that if you build cocoon exclude ojb block, this lib is also required: odmg-3.0.jar. However, I could not confirm that with Cocoon 2.1.10.

Registering Hibernate with Avalon

Now we will register Hibernate with Cocoon. If you are like me, you will fire up Eclipse to accomplish the following part. If you are lazy, simply copy the JAR "hibernate-interface.jar" which is attached to this page to your "WEB-INF/lib" folder. Then you may skip the following sections and read on at "Registering the HibernateFactory".

The first thing we'll do is create an interface `PersistenceFactory` that extends the Component interface from Avalon: (Note: This approach originates from the page CformsHibernateAndFlow!)

```
package org.apache.cocoon.hibernate; // or some other package as you please

import org.apache.avalon.framework.component.Component;

public interface PersistenceFactory extends Component {
        String ROLE = PersistenceFactory.class.getName();

        public org.hibernate.Session createSession();
}
```

As you can see the `PersistenceFactory` will be responsible for the creation of Hibernate Sessions. This is the point where we'll have to decide how Hibernate will connect to the database. My prefered solution is using the cocoon connection pool. Many documents out there will tell you to create a file called hibernate.properties and put your database access information in there. This will work, but Hibernate will then use its own builtin connection pool, and the documentation states clearly that **'the Hibernate builtin connection pool is not for production use**'.

For example, when using MySQL, you might experience problems when leaving the webapp running overnight and coming back in the morning. MySQL connections die after 8 hours of idletime, and the Hibernate builtin pool is not capable of dealing with this. So the first thing you will see is an error message. It goes away after reloading the page, but still it's unacceptable for production use. That's why I encourage you to spend the little extra effort and use Coccoon connection pooling right away.

Where we will do this is in the actual implementation of the `PersistenceFactory` interface, in a class called `HibernateFactory`

IMPORTANT NOTE: Make sure you check out the comments in `initialize()` for instructions on how to add persistent classes. If you are using this to port an existing Hibernate / Cocoon application to the Cocoon connection pool, you will need to change that part according to your configuration.

```
package org.apache.cocoon.hibernate;

import org.apache.avalon.excalibur.datasource.DataSourceComponent;
import org.apache.avalon.framework.activity.Disposable;
import org.apache.avalon.framework.activity.Initializable;
import org.apache.avalon.framework.configuration.Configurable;
import org.apache.avalon.framework.configuration.Configuration;
import org.apache.avalon.framework.configuration.ConfigurationException;
import org.apache.avalon.framework.context.Context;
import org.apache.avalon.framework.context.ContextException;
import org.apache.avalon.framework.context.Contextualizable;
import org.apache.avalon.framework.logger.AbstractLogEnabled;
import org.apache.avalon.framework.service.ServiceException;
import org.apache.avalon.framework.service.ServiceManager;
import org.apache.avalon.framework.service.ServiceSelector;
import org.apache.avalon.framework.service.Serviceable;
import org.apache.avalon.framework.thread.ThreadSafe;
import org.apache.cocoon.Constants;
import org.apache.cocoon.environment.http.HttpContext;

public class HibernateFactory extends AbstractLogEnabled implements
                PersistenceFactory, Configurable, Serviceable, Initializable,
                Disposable, Contextualizable, ThreadSafe {

        private boolean initialized = false;

        private boolean disposed = false;

        private ServiceManager manager = null;

        // do not confuse with Avalon Configuration, Cocoon Session, etc.
        org.hibernate.cfg.Configuration cfg;

        org.hibernate.SessionFactory sf;

        // for debugging: which instance am I using?
        long time_start;

        public HibernateFactory() {
                //System.out.println("Hibernate factory instance created");
        }

        public final void configure(Configuration conf)
                        throws ConfigurationException {
                if (initialized || disposed) {
                        throw new IllegalStateException("Illegal call");
                }
                getLogger().debug("Hibernate configure called");
        }

        public final void service(ServiceManager smanager) throws ServiceException {
                if (initialized || disposed) {
                        throw new IllegalStateException("Illegal call");
                }

                if (null == this.manager) {
                        this.manager = smanager;
                }
                getLogger().debug("Hibernate service called");
        }

        public final void initialize() throws Exception {
                if (null == this.manager) {
                        throw new IllegalStateException("Not Composed");
                }

                if (disposed) {
                        throw new IllegalStateException("Already disposed");
                }

                // adapt:
```

```
                // build sessionfactory
                // map all classes we need
                // keep in sync with configuration file
                //
                try {
                        cfg = new org.hibernate.cfg.Configuration();

                        /* ***** ADD PERSISTENT CLASSES, VARIANT 1 ***** */
                        // persistent classes can be added here using
                        // cfg.addClass(org.test.myClass.class);
                        // Make sure the corresponding .class and .hbm.xml files are located
                        // in
                        // (the same directory of) your classpath (e.g. WEB-INF/classes)
                        // sf = cfg.buildSessionFactory();
                        /* ***** ADD PERSISTENT CLASSES, VARIANT 2 ***** */
                        // alternatively, you might be using a hibernate.cfg.xml file to
                        // load mappings,
                        // then use the following line instead:
                        sf = cfg.configure().buildSessionFactory();

                        // no additional cfg.addClass(...) statements needed, since you can
                        // define
                        // mappings in the XML config file
                } catch (Exception e) {
                        getLogger().error("Hibernate:" + e.getMessage());
                        return;
                }
                this.initialized = true;
                getLogger().debug("Hibernate initialize called");
        }

        public final void dispose() {
                //
                try {
                        sf.close();
                } catch (Exception e) {
                        getLogger().error("Hibernate:" + e.getMessage());
                } finally {
                        sf = null;
                        cfg = null;
                }
                this.disposed = true;
                this.manager = null;
                getLogger().debug("Hibernate dispose called");
        }

        public org.hibernate.Session createSession() {
                org.hibernate.Session hs;
                DataSourceComponent datasource = null;

                // When creating a session, use a connection from
                // cocoon's connection pool
                try {
                        // Select the DataSourceComponent named "test"
                        // This is a normal SQL connection configured in cocoon.xconf
                        ServiceSelector dbselector = (ServiceSelector) manager
                                        .lookup(DataSourceComponent.ROLE + "Selector");
                        datasource = (DataSourceComponent) dbselector.select("hibernate-datasource");
                        // name as defined in cocoon.xconf
                        hs = sf.openSession(datasource.getConnection());
                } catch (Exception e) {
                        getLogger().error("Hibernate: " + e.getMessage());
                        hs = null;
                }
                return hs;
        }

        public void contextualize(Context context) throws ContextException {
                getLogger().debug("Contextualize called: "+context);
                /* Register the HibernateFactory in the Servlet Context so it
                 * may be accessed by the Servlet Filter.
                 */
            HttpContext sContext = (HttpContext)context.get(Constants.CONTEXT_ENVIRONMENT_CONTEXT);
            sContext.setAttribute(PersistenceFactory.ROLE, this);
        }
}
```

Compile these two files. I needed the following jars in my classpath: avalon-framework-api-4.3.jar, excalibur-datasource-2.1.jar, and hibernate3.jar. All these should be in your WEB-INF/lib folder anyway. When you're done, copy the .class files to a directory "org/apache/cocoon/hibernate" (which obviously depends on the package name you chose) in the "WEB-INF/classes/" folder of your cocoon installation, or export them as a JAR and place them in WEB-INF/lib.

Now, you need a hibernate.cfg.xml file in your "WEB-INF/classes" folder. Since we supply connections ourselves, this boils down to:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
      <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
      <!-- Mapping files -->
      <!-- <mapping resource="org/test/beans/Language.hbm.xml"/> -->
</session-factory>
</hibernate-configuration>
```

The "mapping" tag is meant as an example. We do not actually map anything yet, so it is commented out. We will come back to this point later. The most important line is where you tell Hibernate about the actual SQL dialect you are using. If you do not supply this information, Hibernate won't be able to behave accordingly. For example, Hibernate will try to use subqueries which are not supported if you use MySQL.

Registering the HibernateFactory

To register your `HibernateFactory` with Cocoon, put the following line in cocoon.xconf:

```
        <component class="org.apache.cocoon.hibernate.HibernateFactory" role="org.apache.cocoon.hibernate.
PersistenceFactory"/>
```

(Not sure whether it is actually important where you put this. There is a bunch of other component ... class ... statements in cocoon.xconf, so I just put mine above the Xalan XSLT processor component. (Still, after 3 years) need some feedback here!)

If you want to see log messages from Hibernate (you probably want to at this point), create a file "log4j.properties" in "WEB-INF/classes" and put the following into it:

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=info, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

log4j.logger.org.apache.commons.digester=INFO
```

Now restart cocoon. You should see a bunch of messages from Hibernate, meaning that the HibernateFactory is being initialized correctly. However, since we did not create an actual session yet, we can not yet be sure that the Database connection is working right. So, we now want to open and close some sessions, which will be the task of our "OpenSessionInView" filter.

A Servlet Filter for Disposing Hibernate Sessions

Why ?

(If you just want to continue installing and are not bothering about the Why, skip this part and come back later ...)

If you're getting serious about Hibernate, sooner or later you will want to use lazy collection initialization (Read the corresponding section in GettingStarted WithCocoonAndHibernate for an introduction on this topic). Say you are accessing Hibernate sessions from flowscript (you should not, but this is just an example), i.e. as follows:

```
var factory = cocoon.getComponent(Packages.org.test.PersistenceFactory.ROLE);
var hs = factory.createSession();

var data = hs.find("from org.test.Data");

cocoon.sendPage("data-pipe", {data: data} );

hs.close();
```

This will work for simple data objects and also for complex ones, i.e. objects that contain collections, as long as you are not lazily initializing them. If not, you might experience the following problem: Say your object contains a mapped collection `otherData`. In the flow snippet above, you did not access this collection, so Hibernate won't load its items from the database. But it is still possible that your view pipeline wants to access `otherData`.

Now the problem is that the flowscript will just continue to process right after it has invoked the view pipeline, so it is possible that the `hs.close()` will occur BEFORE your view has rendered completely. So when the view will access `otherData` after that, you will get an exception telling you that Hibernate failed to lazily initialize the collection. A classical Race Condition.

Obviously that's no good. The solution is to move Hibernate session disposal out of the flowscript layer and even out of cocoon. I.e., we will create a servlet filter. A servlet filer is invoked before the request is passed to cocoon and will be notified when the request is processed completely. At that point we can safely close the Hibernate session, given we will not continue to work with it from flowscript after the view pipeline has been invoked. However, invoking the view pipeline should be the last thing you do in a flow script anyway, so that won't be too much of a problem.

Creating the filter

Again, if you don't want to compile the filter yourself, it is already in the JAR. You may then skip to "Installing the filter".

Our filter must communicate with cocoon in two ways: First, it must somehow obtain the HibernateFactory in order to be able to open sessions; this is done via the "ServletContext" which exists precisely to share resources among servlets, and is also usable by filters, of course. The "contextualize" method in the HibernateFactory provides this link. The opened sessions must be passed to the DAOs which live totally outside of the Servlet world; this is done via an InheritableThreadLocal, wrapped in a convenience object "PersistenceUtil":

```
package org.apache.cocoon.hibernate;

import java.util.HashMap;

import org.hibernate.Session;

public class PersistenceUtil {

        private static InheritableThreadLocal scope = new InheritableThreadLocal() {
        protected Object initialValue() {
            return null;
          }
        };

        public static void initializeScope(){
                scope.set(new HashMap());
        }

        public static void destroyScope(){
                scope.set(null);
        }

        public static Object getScopeAttribute(String key) {
                return ((HashMap)scope.get()).get(key);
        }

        public static void setScopeAttribute(String key, Object value){
                ((HashMap)scope.get()).put(key,value);
        }

        public static Session getHibernateSession(){
                return (Session) getScopeAttribute("HibernateSession");
        }

        public static void setHibernateSession(Session hs){
                setScopeAttribute("HibernateSession",hs);
        }

        public static void flushTransaction(){
                getHibernateSession().getTransaction().commit();
                getHibernateSession().beginTransaction();
        }
}
```

This is a generic method to communicate between Cocoon, the DAOs and the Filter. Apart from managing the session itself, this is also useful for internationalization: the current locale may be stored in via setScopeAttribute and then accessed by the DAOs. You can read more about i18n and Hibernate at http://www.theserverside.com/tt/blogs/showblog.tss?id=HibernateInternational .

Note that the filter will close the session after the view has been rendered completely, thus solving the LazyInitializationExpection problem. In addition to commiting the transaction, you may want to extend the filter to roll it back in case something goes wrong.

To compile the filter you need the following jars in your classpath:

- hibernate3.jar
- servlet-2.3.jar (or higher; e.g. from the common/lib directory of a Tomcat distribution)

```
package org.apache.cocoon.hibernate;

import java.io.IOException;
import java.sql.SQLException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import org.hibernate.HibernateException;
import org.hibernate.Session;

public class OpenSessionInViewFilter implements Filter {

 private PersistenceFactory persistenceFactory = null;
 private ServletContext ctxt = null;

  public void init(FilterConfig config) throws ServletException {
        /* Get and store ServletContext.
         *
         * Since this will probably be called *before* Cocoon
         * is initialized, we cannot obtain the persistenceFactory
         * right here, but must do so in the doFilter method.
         */
        this.ctxt = config.getServletContext();

  }

  public void destroy() {
          ctxt = null;
          persistenceFactory = null;
  }

  public void doFilter(ServletRequest request, ServletResponse response,
                     FilterChain chain) throws IOException, ServletException {
    /* Obtain persistenceFactory from the servlet context
     * if possible
     */
        if( persistenceFactory == null )
                persistenceFactory = (PersistenceFactory)ctxt.getAttribute(PersistenceFactory.ROLE);

        /* Create request-local scope */
        PersistenceUtil.initializeScope();

        if( persistenceFactory != null ){
                /* If we have a persistenceFactory,
                 * open a Hibernate session to be used throughout
                 * this request.
                 */
                Session hs = persistenceFactory.createSession();
                if( hs != null ){
                        PersistenceUtil.setHibernateSession(hs);
                    hs.beginTransaction();
                    chain.doFilter( request, response );
                    /* The transaction closed here is not the same
                     * one we opened before if the user has called
                     * PersistenceUtil.flushTransaction()
                     */
                    hs.getTransaction().commit();
                    try {
                            /* Although this method is deprecated, as of
                             * Hibernate 3.2 there is no alternative method
                             * to obtain the underlying connection ... this is
                             * scheduled for Hibernate 3.3!
                             * So, it is not possible for the moment to replace
                             * this with something that does not give a
                             * deprecation warning :/
                             */
                               hs.connection().close();
                        } catch (HibernateException e) {
                                e.printStackTrace();
                        } catch (SQLException e) {
                                e.printStackTrace();
                        }
                    hs.close();
                } else {
                        System.out.println("Hibernate Session could not be opened!");
                        chain.doFilter( request, response );
                }
        } else {
                /* Otherwise, do nothing. */
                chain.doFilter(request, response );
        }
        PersistenceUtil.destroyScope();
  }
}
```

Installing the filter

In the file WEB-INF/web.xml, insert the following code avove the "Servlet Configuration" part: (as always, insert your favourite package name)

```
<!-- Filter Configuration ========================================== -->

<filter>
  <filter-name>OpenSessionInViewFilter</filter-name>
  <filter-class>org.apache.cocoon.hibernate.OpenSessionInViewFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>OpenSessionInViewFilter</filter-name>
  <url-pattern>/languages/*</url-pattern>
</filter-mapping>
```

This tells the container to invoke our filter on all requests on the path "/languages/*". Unfortunately, filter mappings are rather unflexible: you can only map to whole subtrees or to all files with a specific ending (like "*.jsp"). Be careful not to open Hibernate sessions on image URLs, for example; although this would not do any damage, it would still slow the request down unnecessarily.

An Example "CRUD" Application

"Now that Hibernate is installed, how do I continue?" - Well the answer is up to you, since there are so many ways of using Hibernate. What follows is some kind of a minimal example so you get a feeling how Hibernate **could** interact with Cocoon. As always, you can compile the classes here yourself, or use the hibernate-interface.jar.

In this example we'll use Hibernate to manage a list of ISO language codes. So the first thing we need is a Java Bean that represents those language codes. Our bean will just have two attributes, the code and a name for each language, plus a numerical ID (I use one for each and every class). The code doesn't look all that interesting:

```
package org.test.beans;

public class Language {
    private long id;
    private String isocode;
    private String name;
    /**
     *
     */
    public Language() {
            super();
    }
    /**
     * @return Returns the iD.
     */
    public long getId() {
            return id;
    }
    /**
     * @param id The iD to set.
     */
    public void setId(long id) {
            this.id = id;
    }
    /**
     * @return Returns the isoCode.
     */
    public String getIsocode() {
            return isocode;
    }
    /**
     * @param isoCode The isoCode to set.
     */
    public void setIsocode(String isocode) {
            this.isocode = isocode;
    }
    /**
     * @return Returns the name.
     */
    public String getName() {
            return name;
    }

    /**
     * @param name The name to set.
     */
    public void setName(String name) {
            this.name = name;
    }
}
```

The language beans will be accessed via DAOs, which could inherit from an abstract class as follows:

```
package org.test.daos;
import org.apache.cocoon.hibernate.PersistenceUtil;
import org.hibernate.Session;

public abstract class DAO {
        public Session session(){
        if (PersistenceUtil.getHibernateSession() == null)
            throw new IllegalStateException("Session has not been set on DAO before usage");
                return PersistenceUtil.getHibernateSession();
        }

        public void flush(){
                PersistenceUtil.flushTransaction();
        }
}
```

The actual implementation of a DAO for languages might look like this:

```
package org.test.daos;

import java.util.List;

import org.hibernate.criterion.Order;
import org.test.beans.Language;

public class LanguageDAO extends DAO {
        public List findAll(){
                return session()
                        .createCriteria(Language.class)
                        .addOrder(Order.asc("isocode"))
                        .list();
        }

        public Language findById(Long id){
                return (Language)session().load(Language.class, id);
        }

        public void makePersistent(Language l){
                session().saveOrUpdate(l);
        }

        public void makeTransient(Language l){
                session().delete(l);
        }
}
```

Compile these classes and store them in the corresponding subfolders of WEB-INF/classes. Next, we will create a table in our database to store the objects in.

```
CREATE TABLE `language` (
  `id` int(11) NOT NULL auto_increment,
  `isocode` varchar(255) NOT NULL default '',
  `name` varchar(255) NOT NULL default '',
  PRIMARY KEY  (`ID`)
)
```

Fill in some sample values, or use the following if you're too lazy: 🙂

```
 INSERT INTO `language` VALUES (1,'de','german');
 INSERT INTO `language` VALUES (2,'en','english');
 INSERT INTO `language` VALUES (3,'fr','french');
 INSERT INTO `language` VALUES (4,'ru','russian');
 INSERT INTO `language` VALUES (5,'pl','polish');
 INSERT INTO `language` VALUES (6,'da','danish');
```

The glue between our POJO and the SQL table is the hibernate mapping definition. This is an XML file which should be called `DCLanguage.hbm.xml`, since our class is called `DCLanguage.class`. Create the file with the following content and put it into the subfolder of WEB-INF/classes where your DCLanguage class resides:

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
        <class name="org.test.beans.Language" table="language">
                <id name="id" column="id" type="long">
                        <generator class="native"/>
                </id>
                <property name="isocode" column="isocode" type="string" not-null="true"/>
                <property name="name" column="name" type="string" not-null="true"/>
        </class>
</hibernate-mapping>
```

Attention, there is a catch: The `<generator class="native"/>` only works if your database is able to generate IDs automatically. (MySQl, for example, is). Refer to the Hibernate documentation if that does not hold in your case (HSQL?).

Note that Hibernate comes with some tools which can automatically generate Java and SQL source code from a mapping file. They only work up to a certain level of complexity, though.

One other thing to note: make sure that the DOCTYPE declaration is correct for the version of Hibernate which you are using. Hibernate3 will need this:

```
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

Now you're almost done. Just insert the following statement:

`<mapping resource="org/test/beans/Language.hbm.xml"/>`

into the `hibernate.cfg.xml` (see *ADD PERSISTENT CLASSES, VARIANT 2* in the source code) and restart Cocoon. Congratulations, you should now be able to manage your ISO language codes via Hibernate and Cocoon 🙂

To verify this, create a testing sub-sitemap, attach a flowscript and insert the following mini-function into it:

```
function language_list()
{
        var ldao = new Packages.org.test.daos.LanguageDAO();
        cocoon.sendPage("language-list.jxt",{languages:ldao.findAll()});
}
```

Create the corresponding pipelines in your sitemap:

```
<map:match pattern="language-list.jxt">
        <map:generate type="jx" src="documents/language-list.jxt"/>
        <map:serialize type="xml"/>
</map:match>

<map:match pattern="language-list">
        <map:call function="language_list"/>
</map:match>
```

And save the following as `documents/language-list.jxt` (relative to your sitemap):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<languages xmlns:jx="http://apache.org/cocoon/templates/jx/1.0">
         <jx:forEach var="lang" items="${languages}">
                <language name="${lang.getName()}" isocode="${lang.getIsoCode()}" />
        </jx:forEach>
</languages>
```

Alas, you're done. Surf to "base-uri-of-your-sitemap"/language-list and contemplate the power of Cocoon and Hibernate 🙂

As said before, this is a minimal example. Now you're ready to continue on your own, the Hibernate docs will be your friend. Also, the attached ZIP file contains an example "application" which covers all CRUD operations and uses CForms.

If you've really read the whole tutorial, yes, you have finally come to an end. If you try this out and run into problems, questions and feedback are appreciated. Also make sure to search through the mailing list archives.

Appendix

Links to other information sources

[GettingStartedWithCocoonAndHibernate]

[CformsHibernateAndFlow]

[UsingHibernateToMakeYourJavaBeansPersistent]

---

page metadata

- AUTHOR:Johannes Textor
- AUTHOR-CONTACT: textor@zeile2.de
- REVIEWED-BY:
- REVIEWER-CONTACT: