Hibernate, Spring, Echo2 Base Application

Introduction

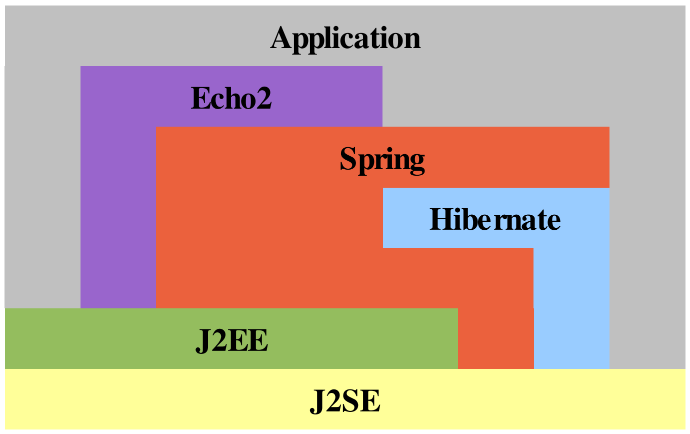
This project is designed as a base application and guide for building AJAX applications using Hibernate, Spring, and Echo2.

It is intentionally left sparse (It contains only a group and user editor) so that it can be used as a base for a full blown application.

Features

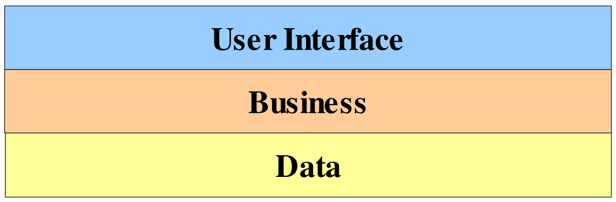
Hibernate, Spring, Echo2 integration:

All three technologies are matched cohesively together to provide a robust base for your AJAX applications.



3-Tier Architecture:

Follows the established pattern of separating the application into data, business, and user-interface tiers.

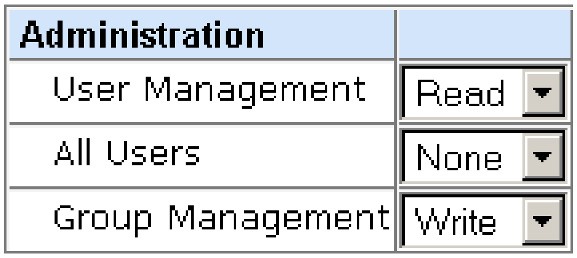


Security:

Allows username/password logon, logoff, and permission checking.
Permissions can be queried using hasPermission(), or enforced using requirePermission().
Enforcement is done in the business tier.

Group-based Permission System:

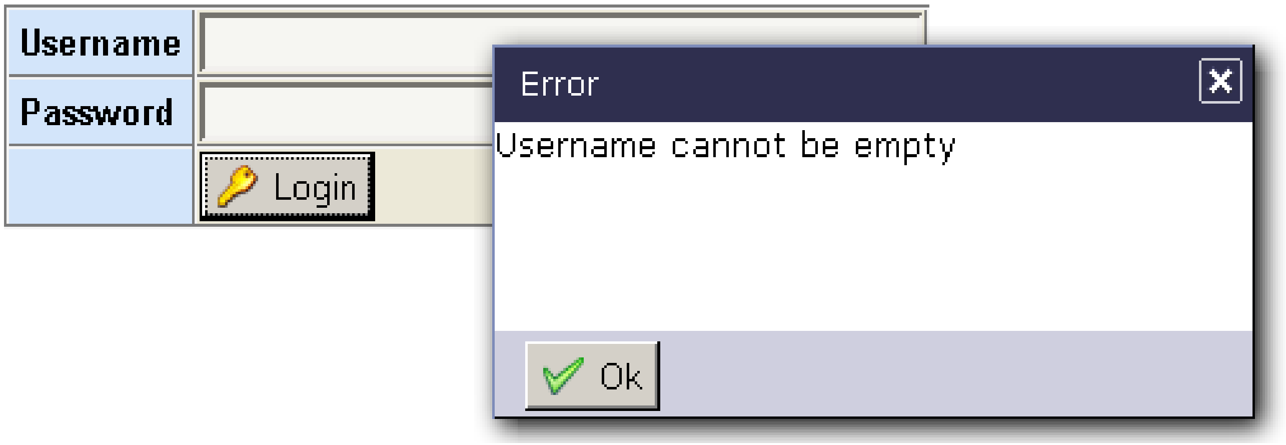
Permissions are group based, which means that users gain permissions based on their group membership (A user can be a member of only one group).
Groups can have multiple permissions, and associate levels for each permission.
The base application uses two methods for interpreting permission levels.
Access based: 0 = DENY, 1 = ALLOW
Read/Write based: 0 = NONE, 1 = READ, 2 = WRITE



Administration	
User Management	Read
All Users	None
Group Management	Write

Input Validation:

A simple and flexible validation system that can handle most common input field validations.
The custom screen and requestor components can display validation error alerts, and can automatically set focus to the offending input field.



Username	
Password	
	Login

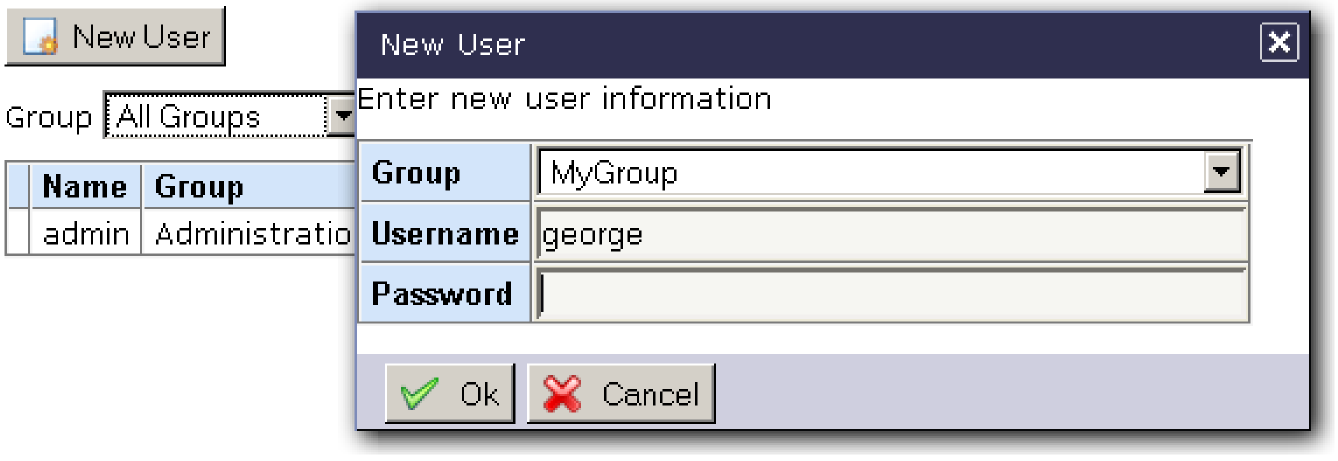
Error

Username cannot be empty

Ok

User Editor:

Example on-screen CRUD editor allowing modification of user properties, as well as group membership.

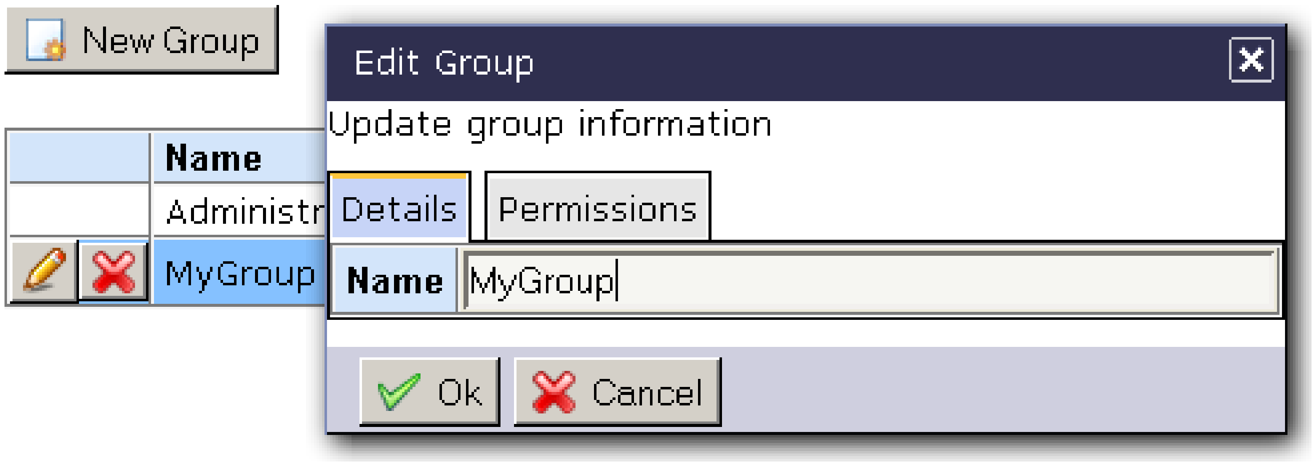


Name	Group
admin	Administratio

Group	MyGroup
Username	george
Password	

Group Editor:

Example on-screen CRUD editor allowing modification of group properties, as well as assigning permission levels.



	Name
	Administr
	MyGroup

Name	MyGroup

Running HSE

Requirements:

- Java 1.4
- A servlet container such as Tomcat

Download:

HSE can be downloaded from http://sourceforge.net/projects/hse

Important Notes:

The application will attempt to make an administrator if none is found in the database.
The default username and password to use are stored in ***application.properties***. By default, they are admin and adminchangeme.

This application uses the Apache Derby database in embedded mode by default.
The driver will attempt to create the database (hsedb) in the directory where the servlet container was started from, owned by the user that started the servlet container.

Upgrading HSE

See ***doc/CHANGES*** for instructions on upgrading your existing application.
You'll need a patch program that can patch at level 1 (i.e. patch -p1).

Architecture

This is a Servlet application, using the Echo2 AJAX framework for its user interface.
Data persistence is provided by Hibernate, and Spring glues it all together.

See ***doc/classDiagram/classDiagram.zuml*** for the class diagram. The PNG images in ***doc/classDiagram*** are the diagram exports.

The primary entry point is *Echo2Servlet*. This class provides application instances (*Echo2Application*). All of the heavy lifting is done by its superclass.

Echo2Application is a subclass of *ApplicationInstance*, which maintains client synchronization information.
This implementation creates a user interface, provides access to session data, and provides server push facilities through *UpdateTasks*

Echo2UserInterface maintains the top level user interface.
This class builds the main screen (title bar, menu, and main screen area), and provides screen swapping facilities.

Security provides login, logout, and permission checks.

Important Classes

Echo2Servlet:

This is the servlet that loads and maintains instances of the Echo2 application. Most of the work is done by its superclass, and so all it has to do is initialize Spring and load a new *ApplicationInstance* when requested.

Echo2Application:

This is the Echo2 application. It maintains state information and synchronizes with the client. Once again, most of the work is done in the superclass.

Echo2UserInterface:

This class maintains the top level user interface.
All of its functionality could be put in *Echo2Application*, but it's already big enough as it is.
This class builds the main screen (Title bar, menu, and main screen area), and provides screen swapping facilities.

EndSessionServlet:

Invalidates the session when it is accessed. Not strictly necessary, but it's a good way to ensure that the session is really clean when a user logs out.

Screen:

The interface that defines a screen.
This application comes with two implementations: *PaneScreen*, which extends *ContentPane*, and *ColumnScreen*, which extends *Column*.
ColumnScreen can be used in situations where you want to add a screen to a component that doesn't support adding a *Pane*.

Security:

Provides login, logout, and permission checks.

Permissions:

Holds references to all permissions available in the application.

Important Files

webapp/index.html:

The default file loaded when a browser points to the application base. This file contains a meta redirect to app, which maps to the application.

Messages.properties:

The i18n text constants file. Contains all of the text strings displayed to the user.

application.properties:

The main application properties. You'll probably want to tweak this.

Configuration

Data Source Configuration:

Although the application does not come configured to use JNDI by default (I want it to be able to run "out-of-the-box"), the JNDI support code is in place, commented out.
To enable JNDI, you must uncomment code in the following files:

- webapp/META-INF/context.xml
- webapp/WEB-INF/applicationContext.xml
- webapp/WEB-INF/web.xml

Be sure to only have one data source uncommented in ***applicationContext.xml***.

Note that the derby connection string specifies the embedded driver. This is fine for toy applications, but you'll probably want a more heavyweight database for industrial use.

Hibernate Configuration:

The bulk of the hibernate configuration is in ***applicationContext.xml*** in beans "hibernateProperties" and "sessionFactory".
You'll notice that the dialect and connection details are placeholders. The actual values are stored in ***application.properties***.
org.springframework.beans.factory.config.PropertyPlaceholderConfigurer provides this placeholder functionality (see the entry in ***applicationContext.xml***).

You'll notice in web.xml under "Hibernate support" a *ContextLoaderListener* and *OpenSessionInViewFilter*.
These tell Spring to manage the Hibernate session for you.
In ***applicationContext.xml***, I set my Hibernate DAO beans to inherit from "txProxyTemplate".
, which adds Spring's AOP transactional support.

In the base application, all DAOs have transactional support added to all methods. Anything not starting with “create”, “update”, or “delete” is set to read-only:

```
<prop key="create*">PROPAGATION_REQUIRED</prop>
<prop key="update*">PROPAGATION_REQUIRED</prop>
<prop key="delete*">PROPAGATION_REQUIRED</prop>
<prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
```

Spring configuration:

The spring configuration is in ***webapp/WEB-INF/applicationContext.xml***.
Look for section "Web Application" to see where the *ApplicationInstance* subclass "webApplication" is defined. This is the Echo2 application.

Spring initialization is done through *Echo2Servlet*, which extends Echo2's *WebContainerServlet*.
It uses *SpringServletHelper* to look for a unique instance of *ApplicationInstance* in the Spring context. Needless to say you must define one and only one *ApplicationInstance* in the spring context file.

Design considerations

Icons and Images:

Icons and tab images are loaded via code rather than through a stylesheet.
I've done this because I can't get consistent behavior (sometimes the icons don't show up), and also because the application detects whether the client supports PNG alpha transparencies and sends a PNG or GIF accordingly. This functionality would be cumbersome if implemented via a stylesheet.

Users and Authentication:

A more versatile way to implement authentication would be to separate the method of authentication from the user class (Creating a credentials interface and associating instances of a particular implementation with user objects).
However, the method used in this application is easy to understand and implement, and is usually enough for an application that doesn't require more exotic authentication methods such as biometrics.

Database Disconnect Detection:

This is a bit tricky to do since the transactional layer tends to get in the way.
To make matters worse, different drivers throw different kinds of exceptions, some more obscure than others.
I've created *ExceptionAnalyzer* to deal with this, and have tested it on Derby, PostgreSQL, and SQLServer. If an exception makes it past this class when a disconnect occurs, you'll have to analyze the stack trace and add more signature checking to isDatabaseDisconnect().

Pooling:

While I've left a commented out entry in applicationContext.xml for DBCP, I'm now using C3P0 exclusively in the application.
C3P0 supports many different ways to test database connections for validity. By default, I'm using testConnectionOnCheckin=true, and idleConnectionTestPeriod=60.
See http://www.mchange.com/projects/c3p0/index.html for a study of the pros and cons to this approach.

HOWTO

Adding a Hibernate persisted class and DAO:

1. Make a new package.
   Example: org.stenerud.data.widget

2. Make a new POJO class.
   Example:
```
package org.stenerud.hse.data.widget;

public class Widget
{
      private int id;
      private String name;

      public int getId()
      {
            return id;
      }
      public void setId(int id)
      {
            this.id = id;
      }
      public String getName()
      {
            return name;
      }
      public void setName(String name)
      {
            this.name = name;
      }
}
```

3. Add a new Hibernate mapping file.
   Example:
```
<?xml version="1.0" encoding="UTF-8"?>
<!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.stenerud.hse.data.widget.Widget" table="HSEWidget">
    <id column="widgetId" name="id" type="int" unsaved-value="-1">
      <generator class="native"/>
    </id>
    <property column="name" length="255" name="name" not-null="true"
       type="string"/>
  </class>
</hibernate-mapping>
```

4. Add the mapping file to the Hibernate configuration.
   Example:

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource"><ref bean="dataSource"/></property>
  <property name="hibernateProperties">
    <ref bean="hibernateProperties" />
  </property>
  <property name="mappingResources">
    <list>
      <value>org/stenerud/hse/data/group/Group.hbm.xml</value>
      <value>org/stenerud/hse/data/security/Permission.hbm.xml</value>
      <value>org/stenerud/hse/data/security/PermissionLevel.hbm.xml</value>
      <value>org/stenerud/hse/data/user/User.hbm.xml</value>
      <value>org/stenerud/hse/data/widget/Widget.hbm.xml</value>
    </list>
  </property>
</bean>
```

5. Make a DAO.
   Example:

```
package org.stenerud.hse.data.widget;

import java.util.List;

/**
 * Data access for widgets.
 *
 * @author You
 */
public interface WidgetDAO
{
      /**
       * Get all widgets. Widgets are returned in alphabetical order.
       *
       * @return a list of all widgets.
       */
      public List getWidgets();

      /**
       * Get a widget by name.
       *
       * @param name the name of the widget.
       * @return the widget or null if not found.
       */
      public Widget getWidget(String name);

      /**
       * Refresh a widget with a version from permanent storage.
       *
       * @param widget the widget to refresh.
       * @return a reference to the widget passed in.
       */
      public Widget refresh(Widget widget);

      /**
       * Create a new widget
       *
       * @param widget the widget to create.
       */
      public void create(Widget widget);
	/**
	 * Update a widget.
	 *
	 * @param widget the widget to update.
	 */
	public void update(Widget widget);

	/**
	 * Delete a widget.
	 *
	 * @param widget the widget to delete.
	 */
	public void delete(Widget widget);
}

package org.stenerud.hse.data.widget;

import java.util.List;

import org.stenerud.hse.data.CriteriaBuffer;
import org.stenerud.hse.data.ExtendedHibernateDaoSupport;

public class WidgetDAODatabase extends ExtendedHibernateDaoSupport implements
WidgetDAO
{
	// ========== CONSTANTS ==========
	private static final String STANDARD_FROM = "FROM Widget widget";
	private static final String STANDARD_ORDER = "ORDER BY widget.name";

	// ========== IMPLEMENTATION ==========

	public void create(Widget widget)
	{
		getHibernateTemplate().save(widget);
	}

	public void delete(Widget widget)
	{
		getHibernateTemplate().delete(widget);
	}

	public Widget getWidget(String name)
	{
		CriteriaBuffer criteria = new CriteriaBuffer(STANDARD_FROM);
		criteria.addCriteria("widget.name =", name);

		return (Widget)getFirst(criteria.getQuery());
	}

	public List getWidgets()
	{
		CriteriaBuffer criteria = new CriteriaBuffer(STANDARD_FROM,
			STANDARD_ORDER);
		return getHibernateTemplate().find(criteria.getQuery());
	}

	public Widget refresh(Widget widget)
	{
		getHibernateTemplate().refresh(widget);
		return widget;
	}

	public void update(Widget widget)
	{
		getHibernateTemplate().update(widget);
	}
}
```

6. Add a bean entry to applicationContext.xml
   Example:

```
<bean id="widgetDAO" parent="txProxyTemplate">
 <property name="target">
  <bean class="org.stenerud.hse.data.widget.WidgetDAODatabase">
   <property name="sessionFactory"><ref local="sessionFactory"/></property>
  </bean>
 </property>
</bean>
```

Adding a screen:

1. Build a screen from PaneScreen, ColumnScreen, or roll your own.
   Example:

```
package org.stenerud.hse.ui.echo2.screen.widgets;

import nextapp.echo2.app.Label;

import org.stenerud.hse.ui.echo2.screen.PaneScreen;

/**
 * The widget screen. Does nothing at the moment.
 *
 * @author You
 */
public class WidgetScreen extends PaneScreen
{
	private static final long serialVersionUID = 1L;

	public String getTitle()
	{
		// Remember to set screen.widgets in Messages.properties!
		return messages.get("screen.widgets");
	}

	protected void initComponents()
	{
		add(new Label("Here there be widgets!"));

	}

	protected void resetComponents()
	{
		// Nothing to do.
	}

}
```

2. Add an entry into applicationContext.xml.
   Example:
```
<!-- Widgets Screen -->
<bean id="widgetsScreen"
    class="org.stenerud.hse.ui.echo2.screen.widgets.WidgetScreen"
    singleton="false">
  <property name="messages"><ref local="messages"/></property>
</bean>
```

3. Add the screen name to Screens.
   Example:
```
public static final String WIDGETS = "widgetsScreen";
```

4. Add a menu entry in Echo2UserInterface.
   Example:
```
private void buildMainMenu()
{
    ...
    mainMenu.addItem(topBarMenu, MainMenu.STYLE_MENU,
        messages.get("screen.widgets"), new ActionListener()
    {
        private static final long serialVersionUID = 1L;

        public void actionPerformed(ActionEvent e)
        {
            setScreen(screenFactory.createScreen(Screens.WIDGETS));
        }
    });
    ...
}
```

5. Update Messages.properties.
   Example:
```
screen.widgets=Widgets
```