Tapestry4Spring

NOTE: This is outdated information that applies only to Tapestry 4. For the current versions of Tapestry see

http://tapestry.apache.org/integrating-with-spring-framework.html

Combining Tapestry 4 and Spring

How can you reference Spring beans from Tapestry 4? In earlier versions of Tapestry, the most common method was to extend the BaseEngine class. However, in Tapestry 4 the BaseEngine class is deprecated, and we now need to use or extend SpringBeanFactoryHolder.

In the following varying methods are described of setting up Spring support in Tapestry.

(*Basic knowledge of Java, Tapestry, and Spring assumed.*)

Step 1: Hivemind Configuration

Note: You can skip this step by downloading tapestry-spring.jar from http://sourceforge.net/projects/diaphragma and placing it on your classpath.

If you already take care of Spring elsewhere, you can avoid implementing your own SpringBeanFactoryHolder implementation class, just add an entry in your hivemodule.xml:

```
<implementation service-id="hivemind.lib.DefaultSpringBeanFactoryHolder">
  <invoke-factory>
    <construct autowire-services="false" class="engine.PersonalSpringBeanFactoryHolder">
      <event-listener service-id="hivemind.ShutdownCoordinator" />
      <set-object property="context" value="service:tapestry.globals.WebContext" />
    </construct>
  </invoke-factory>
</implementation>
```

Continue reading, if you're interested in how it works...

Tapestry uses Hivemind, behind the scenes, for dependency-injection. Hivemind's XML configuration is similar to Spring. The easiest way to contribute to the whole Hivemind registry is by creating hivemodule.xml in your WEB-INF directory.

Here is what you need in this project to provide a new implementation of the DefaultSpringBeanFactoryHolder:

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

<module id="diaphragma.tapspr" version="1.0.0">
    <implementation
        service-id="hivemind.lib.DefaultSpringBeanFactoryHolder">
        <invoke-factory>
            <construct autowire-services="false"
                class="diaphragma.tapspr.XSpringBeanFactoryHolderImpl">
                <event-listener
                    service-id="hivemind.ShutdownCoordinator" />
                <set-object property="context"
                    value="service:tapestry.globals.WebContext" />
            </construct>
        </invoke-factory>
    </implementation>
</module>
```

Next job is to create XSpringBeanFactoryHolderImpl.java. It look like this:

```
package diaphragma.tapspr;

import java.io.PrintStream;
import org.apache.hivemind.events.RegistryShutdownListener;
import org.apache.hivemind.lib.impl.SpringBeanFactoryHolderImpl;
import org.apache.tapestry.web.WebContext;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ConfigurableApplicationContext;


public class XSpringBeanFactoryHolderImpl extends SpringBeanFactoryHolderImpl
    implements RegistryShutdownListener
{
    private WebContext context;

    public XSpringBeanFactoryHolderImpl()
    {
    }

    public void setContext(WebContext webcontext)
    {
        context = webcontext;
    }

    public WebContext getContext()
    {
        return context;
    }

    public BeanFactory getBeanFactory()
    {
        if(super.getBeanFactory() == null)
            super.setBeanFactory(XWebApplicationContextUtils.getWebApplicationContext(getContext()));
        return super.getBeanFactory();
    }

    public void registryDidShutdown()
    {
        ((ConfigurableApplicationContext)super.getBeanFactory()).close();
    }

}
```

As we can see, this class extends the default SpringBeanFactoryHolder. The important thing is what you see in getBeanFactory() method, there you define where our BeanFactory located. In this example, I use WebApplicationContextUtils.getRequiredWebApplicationContext() method. There is another method which doesn't throw exception WebApplicationContextUtils.getWebApplicationContext().

Next, we implement XWebApplicationContextUtils.java

```
package diaphragma.tapspr;

import org.apache.tapestry.web.WebContext;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;

public class XWebApplicationContextUtils extends WebApplicationContextUtils
{

    public XWebApplicationContextUtils()
    {
    }

    public static WebApplicationContext getWebApplicationContext(WebContext webcontext)
    {
        Object obj = webcontext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
        if(obj == null)
            return null;
        if(obj instanceof RuntimeException)
            throw (RuntimeException)obj;
        if(obj instanceof Error)
            throw (Error)obj;
        if(!(obj instanceof WebApplicationContext))
            throw new IllegalStateException((new StringBuilder()).append("Root context attribute is not of type
WebApplicationContext: ").append(obj).toString());
        else
            return (WebApplicationContext)obj;
    }

    public static WebApplicationContext getRequiredWebApplicationContext(WebContext webcontext)
        throws IllegalStateException
    {
        WebApplicationContext webapplicationcontext = getWebApplicationContext(webcontext);
        if(webapplicationcontext == null)
            throw new IllegalStateException("No WebApplicationContext found: no ContextLoaderListener
registered?");
        else
            return webapplicationcontext;
    }
}
```

Step 2: Spring Configuration

Spring in servlet mode need two things, it needs the XML file for bean configuration and also a filter in web.xml. In web.xml, you will have to add this:

```
<context-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
<listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
         </listener-class>
</listener>
```

And now let us try a simple XML file (place it under WEB-INF/ as applicationContext.xml).

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
        <bean id="person" class="com.example.model.Person">
        <property name="name">
                <value>Nanda Firdausi</value>
        </property>
        </bean>
</beans>
```

It defines one object with one property.

Step 3: Access Spring property from Tapestry page specification

Now time to see whether our bean can be accessed from a Tapestry page. First, let us create the page specification (Home.page).

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE page-specification PUBLIC
        "-//Apache Software Foundation//Tapestry Specification 4.0//EN"
        "http://jakarta.apache.org/tapestry/dtd/Tapestry_4_0.dtd">

<page-specification class="org.apache.tapestry.html.BasePage">
        <inject property="person" object="spring:person" />
</page-specification>
```

The page template is trivial, one super-simple-example is like this (Home.html).

```
<span jwcid="@Insert" value="ognl:person.name" />
```

Please make comments, suggestions, and any other things related to this tutorial.

This information was originally written in an email to the Tapestry User's List on March 7, 2005 by Nanda Firdausi

HowardLewisShip: This is pretty much overkill, since you can just acquire your AC and store it into the DefaultSpringBeanFactoryHolder service with no additional work. I do like that you have a shutdown option, though. I'm putting together a proper implementation for Tapestry @ JavaForge.

HowardLewisShip: I've put together a basic Spring integration module: tapestry-spring. The solution on this page is a little more flexible, however (at least for the moment).

NandaFirdausi: I've seen your implementation, and I like it too, just like your other code 😉. I thing your implementation doesn't need spring listener anymore, am I right? If so, then the choice to the user is if they do have another spring wep application with nothing to do with tapestry, it's better to set the spring listener and do like this page says. If your web application is all tapestry based (with spring as back-end support), then your code looks cleaner for me 😉

Tapestry 4 (another solution)

JarekWoloszyn: Here is another solution for Tapestry4.

We need a helper class which will create WebApplicationContext for the ServletContext. Hivemind can't call factory methods (from WebApplicationContext Utils), so we create a POJO. Spring Context is re-created everytime we change ServletContext.

```
package org.your.application;

import javax.servlet.ServletContext;

import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;


public class SpringContextFactory {
    private ServletContext servletContext;
    private WebApplicationContext appContext;

    public WebApplicationContext getAppContext() {
        return appContext;
    }

    public ServletContext getServletContext() {
        return servletContext;
    }

    public void setServletContext(ServletContext servletContext) {
        this.servletContext = servletContext;
        appContext = WebApplicationContextUtils.getWebApplicationContext(getServletContext());
    }
}
```

Tapestry 4 has Spring integration out of the box. We must only say which BeanFactory should be used. In hivemind.xml, we define a service-point for our helper class. This class takes ServletContext as parameter. We configure then Hivemind to use appContext member as spring-bean factory.

```
<?xml version="1.0"?>
<module id="app" version="1.0.0" package="org.your.application">

  <contribution configuration-id="hivemind.ApplicationDefaults">
    <default symbol="hivemind.lib.spring-bean-factory" value="service-property:app.SpringContextFactory:
appContext"/>
  </contribution>

    <service-point id="SpringContextFactory">
        Create WebApplicatonContext for Spring
        <invoke-factory>
            <construct class="SpringContextFactory">
                <set-service property="servletContext" service-id="tapestry.globals.ServletContext"/>
            </construct>
        </invoke-factory>

    </service-point>

</module>
```

And that's all. Now you can use spring: prefix to access Spring beans:

```
    @InjectObject("spring:DAOFactory")
    public abstract DAOFactory getDao();
```

Injecting Spring Beans

You may inject beans managed by Spring into pages and components just like you inject anything else as described by http://tapestry.apache.org/tapestry4/UsersGuide/injection.html. You can describe the injection in the .page/.jwc file or using an @InjectObject annotation. Just use the 'spring:' prefix like:

```
@InjectObject("spring:name.from.applicationContext.xml")
```

You may also wire Spring managed beans in HiveMind services such as IEngine services.

```
<service-point id="MyService" interface="org.apache.tapestry.engine.IEngineService">
  <invoke-factory>
    <construct class="engine.MyService">
    <set-object property="springBean" value="spring:name.of.spring.bean"/>
    </construct>
  </invoke-factory>
</service-point>
```

Spring, Tapestry and Hibernate

In order to use lazy loading and for most Spring Hibernate templates, you need access to a Hibernate Session. The traditionnal Spring solution is to use the "OpenSessionInView" pattern where a Hibernate session is open for the whole request. However, OpenSessionInView has to be configured in the web.xml which is not very friendly when you setup some particular Tapestry URLs.

A better solution is to integrate the session opening/closing in Hivemind by using the WebRequestServicerFilter interface. Also, for Tapestry assets, it is not necessary to create a session. This code is based on the Spring OpenSessionInViewFilter.

This requires to implement a class and wire it into Tapestry. Here is the code for the class:

```
package actualis.web.tapestry.framework;

import java.io.IOException;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hivemind.lib.SpringBeanFactoryHolder;
import org.apache.tapestry.Tapestry;
import org.apache.tapestry.services.ServiceConstants;
import org.apache.tapestry.services.WebRequestServicer;
import org.apache.tapestry.services.WebRequestServicerFilter;
import org.apache.tapestry.web.WebRequest;
import org.apache.tapestry.web.WebResponse;
import org.hibernate.FlushMode;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.dao.DataAccessResourceFailureException;
import org.springframework.orm.hibernate3.SessionFactoryUtils;
import org.springframework.orm.hibernate3.SessionHolder;
import org.springframework.transaction.support.TransactionSynchronizationManager;

public class HibernateRequestFilter implements WebRequestServicerFilter {

  private static Log logger = LogFactory.getLog(HibernateRequestFilter.class);

  public static final String DEFAULT_SESSION_FACTORY_BEAN_NAME = "sessionFactory";

  private String sessionFactoryBeanName = DEFAULT_SESSION_FACTORY_BEAN_NAME;

  private SpringBeanFactoryHolder _beanFactoryHolder;

  /**
   * Set the bean name of the SessionFactory to fetch from Spring's root
   * application context. Default is "sessionFactory".
   *
   * @see #DEFAULT_SESSION_FACTORY_BEAN_NAME
   */
  public void setSessionFactoryBeanName(String sessionFactoryBeanName) {
    this.sessionFactoryBeanName = sessionFactoryBeanName;
  }

  /**
   * Return the bean name of the SessionFactory to fetch from Spring's root
   * application context.
   */
  protected String getSessionFactoryBeanName() {
    return sessionFactoryBeanName;
  }
```

```
  public void service(WebRequest request, WebResponse response,
      WebRequestServicer servicer) throws IOException {
    String svcValue = request.getParameterValue(ServiceConstants.SERVICE);
    if (Tapestry.ASSET_SERVICE.equals(svcValue)) {
      servicer.service(request, response);
      return;
    }
    logger.debug("entering into Hibernate Request Filter " + " service:" + svcValue
        + " context:" + request.getContextPath() + " activation:"
        + request.getActivationPath() + " path:" + request.getPathInfo());
    SessionFactory sessionFactory = lookupSessionFactory(request);
    Session session = null;

    // single session mode

    logger.debug("Opening single Hibernate Session in OpenSessionInViewFilter");
    session = getSession(sessionFactory);
    TransactionSynchronizationManager.bindResource(sessionFactory, new SessionHolder(
        session));

    try {
      servicer.service(request, response);
    }

    finally {
      // single session mode
      TransactionSynchronizationManager.unbindResource(sessionFactory);
      logger.debug("Closing single Hibernate Session in OpenSessionInViewFilter");
      try {
        closeSession(session, sessionFactory);
      } catch (RuntimeException ex) {
        logger.error("Unexpected exception on closing Hibernate Session", ex);
      }
    }

  }

  /** For injection. */
  public final void setBeanFactoryHolder(SpringBeanFactoryHolder beanFactoryHolder) {
    _beanFactoryHolder = beanFactoryHolder;
  }

  protected SessionFactory lookupSessionFactory(WebRequest request) {
    if (logger.isDebugEnabled()) {
      logger.debug("Using SessionFactory '" + getSessionFactoryBeanName()
          + "' for OpenSessionInViewFilter");
    }
    return (SessionFactory) _beanFactoryHolder.getBeanFactory().getBean(
        getSessionFactoryBeanName(), SessionFactory.class);
  }

  protected Session getSession(SessionFactory sessionFactory)
      throws DataAccessResourceFailureException {
    return openSession(sessionFactory);
  }

  protected Session openSession(SessionFactory sessionFactory)
      throws DataAccessResourceFailureException {
    Session session = SessionFactoryUtils.getSession(sessionFactory, true);
    // session.setFlushMode(FlushMode.NEVER);
    session.setFlushMode(FlushMode.COMMIT);
    return session;
  }

  protected void closeSession(Session session, SessionFactory sessionFactory) {
    session.close();
    // SessionFactoryUtils.releaseSession(session, sessionFactory);
  }

}
```

The wiring into tapestry is the following:

```
    <service-point id="HibernateServicerFilter"
        interface="org.apache.tapestry.services.WebRequestServicerFilter">
        <invoke-factory>
            <construct class="actualis.web.tapestry.framework.HibernateRequestFilter">
                                <set-object property="beanFactoryHolder" value="service:hivemind.lib.
DefaultSpringBeanFactoryHolder"/>
          </construct>
        </invoke-factory>
    </service-point>

    <contribution configuration-id="tapestry.request.WebRequestServicerPipeline">
        <filter name="HibernateServicerFilter"
            object="service:HibernateServicerFilter"/>
    </contribution>
```