keel
META FRAMEWORK
Any Component
Any Service
Any Deployment

The Keel Meta-framework: A Hands On Tutorial

Shash Chatterjee

Published September 29, 2003 on TheServerSide.com

TheServerSide
Your J2EE Community

Table of Contents

Index of Figures

Introduction

There has been a movement afloat for a while in the Java world to move away from proprietary, one of a kind developments, to the use of application development frameworks. Application development frameworks offer many benefits. Chief among them is the reduction in the development effort and time to bring a product to market. This benefit arises from using an integrated set of features that over time have proven to be common to all large and medium scale developments. As well, using the same tools over and over in projects by multiple people and organizations, allows a framework to mature far more rapidly, and reach a level of stability, that one of a kind developments can rarely match. Frameworks provide structure to an application by providing established implementation patterns. The very fact that makes frameworks attractive, being repositories of integrated functions and components, may also sometimes pose a somewhat steep initial learning curve. This article is an attempt to take one such framework, the Keel Meta-framework, and lead the uninitiated through the first paces.

The Keel Meta-framework

The Keel Meta-framework, available at http://www.keelframework.org, is an Open Source "framework of frameworks". The question is, if Keel is a framework or not? Certainly, Keel is a framework, in that it provides common services such as a component container, logging, persistence, configuration, and so on and so forth. However, Keel is more than that, in that it provides its services less by means of its own proprietary code base. Instead, Keel concentrates on providing a very thin backbone, and a set of core interfaces for each of its services. Keel then relies on other, best-of-breed, Open Source frameworks and libraries to provide implementations of its core services. This allows the application architect to pick and choose which Open Source (and, Commercial, for that matter) frameworks and libraries best suit the end application. So, Keel is a framework, but also a framework of frameworks, hence a "Meta-framework".

The Keel Meta-framework has some other salient features which distinguish it from other frameworks. Being a meta-framework, it isn't itself a presentation framework. But, Keel goes to great lengths to keep the MVC separation of the view. As such, as will be seen later in this tutorial, Keel applications are written in a vacuum without thought to the eventual user-interface. When done, user-interfaces can be added, with the presentation framework choices being Struts, Cocoon, Velocity, Command Line Interface, and Axis/SOAP. Keel is designed to be distributable, and has the concepts of a "client side" and a "server side" (which is a bit confusing, since we are still talking about Java "server side" applications, in general). Having this architectural separation allows for easy distribution of the client and server across JVMs or hosts. So, the client webapp or web-service or even a future Swing client can run on a single VM, and communicate to the Keel server running in a distributed JVM, using JMS or SOAP as communication protocols. As a simplification, both the Keel server and client sides can be packaged together and run in a single VM, which is the version we will use in this tutorial.

An article called "Keel - The Next Generation Framework", discussing the architecture of Keel in more detail, was published recently on TheServerSide. The article provides useful information for understanding this tutorial, and is available at http://www.theserverside.com/resources/article.jsp?l=Keel.

About the Tutorial

This tutorial came about because of a live, hands-on tutorial about the Keel Meta-framework presented to the Hands On Java SIG of JavaMUG (http://www.javamug.org). The JavaMUG users' group, based in the Dallas/Fort Worth Metroplex, is one of the largest and most active Java users' groups in the nation. The monthly Hands On Java Sessions are in a format where the participants each follow along live with their own desktop PC, while the presenter leads on a PC at the podium where the display is projected on to a large screen. Typical sessions are three hours long, with the application pre-installed on the systems. The sample files of the tutorial came first, this document was written later to supplant the live presenter. Therefore, the overall tone of the tutorial is cookbook-like, and does not delve into details of each and every aspect of what is being done.

Given the origins of this tutorial, three hours is plenty of time to go through it. Additional time should be allocated to download and install Keel, and more if the Java and Ant environments are not already setup. Certainly, a day is plenty of time to go through the entire setup and tutorial.

The tutorial has the following objectives for the reader to accomplish:

- Download and install Keel
- Build Keel with JVM/Ant
- Setup infra-structure for a typical application
- Create a "Hello World" application
- Add configuration to the application
- Add authorization (security) to the application
- Add persistence to the application
- Add an user-interface to the application

A Few Notes

As mentioned before, Keel can be deployed in a myriad of ways. For this tutorial we will concentrate on a non-distributed, single-VM webapp, where the web-application, Keel client side and Keel server side all reside and run within the same VM. For a web container, we will use Tomcat. The user interface will be demonstrated with Struts and JSPs.

One of the fascinating aspects of Keel has been the rapid pace with which new services and modules have been added to Keel. Over time, therefore, the exact list of services in Keel will change such that the examples provided in this tutorial may look somewhat different from the true list. However, the process of adding properties and services to Keel is generic enough that the pattern should be quite evident.

All the code developed for this tutorial is available as a downloadable bundle on Keel's SourceForge page at http://www.sourceforge.net/projects/keel. A version of the code and this document is available from Keel CVS as the "app-hoj" module. The current plans are for the CVS version of this tutorial to be kept up to date as Keel changes over time.

If help is needed with this tutorial, a great resource for Keel developer information is the Keel Documentation Wiki, mentioned in the references section at the end of this tutorial. The user-maintained Wiki has a plethora of information about the different Keel deployments, "Howto" articles on the various Keel services, a "getting started" tutorial, and many other subjects. Other great resources, particularly for asking questions, are the Keel user and developer mailing lists, mentioned in the references section.

Infra-structure

Prerequisites for Keel

Before your Keel-based application can be developed and deployed, there are three things that must be installed and available on your system:

- Java development environment
- Ant build system
- Keel distribution

The Java Environment

The first thing that is needed is the Java environment using a Java Software Development Kit (JDK). If Java is not already installed on your system, it can be obtained from http://java.sun.com/downloads/. Keel mandates use of Java version 1.4, ideally JDK-1.4.1_03. after you have installed Java, as a check, type the following command:

```
java -version
```

If Java is installed properly, you should see output similar to:

```
java version "1.4.1_03" Java(TM) 2 Runtime Environment, Standard Edition (build
        1.4.1_03-b02) Java HotSpot(TM) Client VM (build 1.4.1_03-b02, mixed mode)
```

The Build Environment

Keel's build system is based on the popular Ant package. If you don't have it, download and install Ant from http://ant.apache.org. Keel will work with any version of Ant above version 1.5. After Ant is installed, as a check type the following command:

```
ant -version
```

If Ant is installed properly, you should see output similar to:

```
Apache Ant version 1.5.3 compiled on April 16 2003
```

Keel Meta-framework

Keel can be downloaded as a compressed source distribution from Sourceforge. For the die-hard developers that want to keep up with the latest developments in Keel, anonymous CVS access is offered. This article was written with Keel-2.0 features in mind. Any 2.0 or later release of Keel should work with this tutorial. The closest Keel release available for download at the time of writing is 2.0-ea3. To download any version of Keel, point your browser to http://sourceforge.net/projets/keel.

You can also get Keel from CVS. Instructions on acquiring Keel through CVS are on the Keel Wiki page at http://66.105.113.115/vqwiki-2.3.5/jsp/Wiki?topic=CvsPage. For the purposes of this tutorial, checking out the "default" module, which is an alias, will checkout all the necessary modules.

Installing Keel and Required Tools

The biggest part of installing Keel is simply uncompressing the distribution bundle into a local directory on your computer. Keel can be installed anywhere on the filesystem. Select a directory where you want to install Keel, and use your favourite "unzip" utility to extract the files from the Keel distribution into this directory. The directory where Keel is installed will be referred to in the rest of this tutorial with "KEEL". This would be the equivalent of "$KEEL" on Unix systems, and "%KEEL%" on Windows systems. Once extracted, if you browse the KEEL directory, you'll find directories such as "keel-build", "keel-core", "keel-server" in there. In addition, you'll find other directories starting with the prefix "comm-" (Keel client-server communications), "clnt-" (Keel client adapters), "svc-" (Keel service implementations), and "app-" (Keel applications).

If you are checking out Keel from CVS, then each module that you checkout from CVS, should be placed in the KEEL directory.

Keel supports a myriad of configuration choices, and as such, configuration could get quite harrowing. The Ant build scripts for Keel have been written to generalize the approach of acquiring, installing and then adapting Keel to these tools. All of this power in the Ant scripts can be harnessed by the alteration of some property files, called "deployment property files". These files all end with the suffix "-deploy.properties", and predefined ones can be found in KEEL/keel-build. For this tutorial we'll use the "default" deployment, which simply refers to "default-deploy.properties"; this is a property file pre-configured for a Keel deployment that runs the Keel server and client in a single VM, embedded within a Struts/JSP based webapp that is deployed to the Tomcat container.

Downloading Tomcat and Struts

Technically Tomcat and Struts are not part of Keel. However, a big part of Keel is about integrating disparate tools. Keel provides Ant scripts to facilitate the downloading and installation of all the third-party tools, not just Tomcat and Struts, needed to ultimately deploy an application. If you have Tomcat and Struts already installed on your system, you can simply change some properties and instruct Keel to deploy using your installed tools. However, for purposes of learning Keel, it is far easier to just let Keel download and install Keel on its own. This has two benefits. First, it allows for working with a known quantity; it is easier to communicate with other developers to get help since all are familiar with the standard locations of tools. Second, it will leave your standard installation untouched; if errors are made during the learning process the package in question can simply be deleted and the process can be started over.

To download the required 3rd-party tools, simply change directory to KEEL/keel-build, and type the following:

```
ant download-keeltools
```

This will start downloading the versions of Tomcat and Struts that Keel was integrated with. The downloaded distributions will be palced in KEEL/keel-build/deploy, by default. The deployment directory can be changed with, you guessed it, some Ant property changes. If you were using another deployment, sunch as openjms-cocoon, running the identical command would have resulted in the download of Tomcat, Cocoon and OpenJMS. This step obviously requires connectivity to the Internet when the command is run; if you do not have network access you will have to acquire the distributions in some other fashion and place them manually in KEEL/keel-build/deploy.

Installing Tomcat and Struts

Once the 3rd-party distributions are in place, to install them, run the command:

```
ant install-keeltools
```

This will expand the distributions in KEEL/keel-build/deploy, and will run installation scripts, if necessary.

Building and Deploying Keel

Now that the prior steps have the required environment all setup, it is time to build and deploy Keel. To build and deploy Keel, all that needs to done is the following command:

```
ant assemble-deploy
```

This will prompt for a deployment name, supply "default" at the prompt. The prompting could be avoided by passing an extra parameter when Ant is invoked:

```
ant -D"deploy.name=default" assemble-deploy
```

The build takes a while, compiling, processing meta-data, building JARs, assembling configuration files, building webapp archive (WAR) files, etc. When it finally stops, Tomcat can be started with the command:

```
ant tomcat-start
```

Once Tomcat starts up, Keel can be accessed from a broswer by accessing http://localhost:8080/struts When it is time to shut it down, Tomcat can be shut down with the command:

```
ant tomcat-stop
```

Important Directories

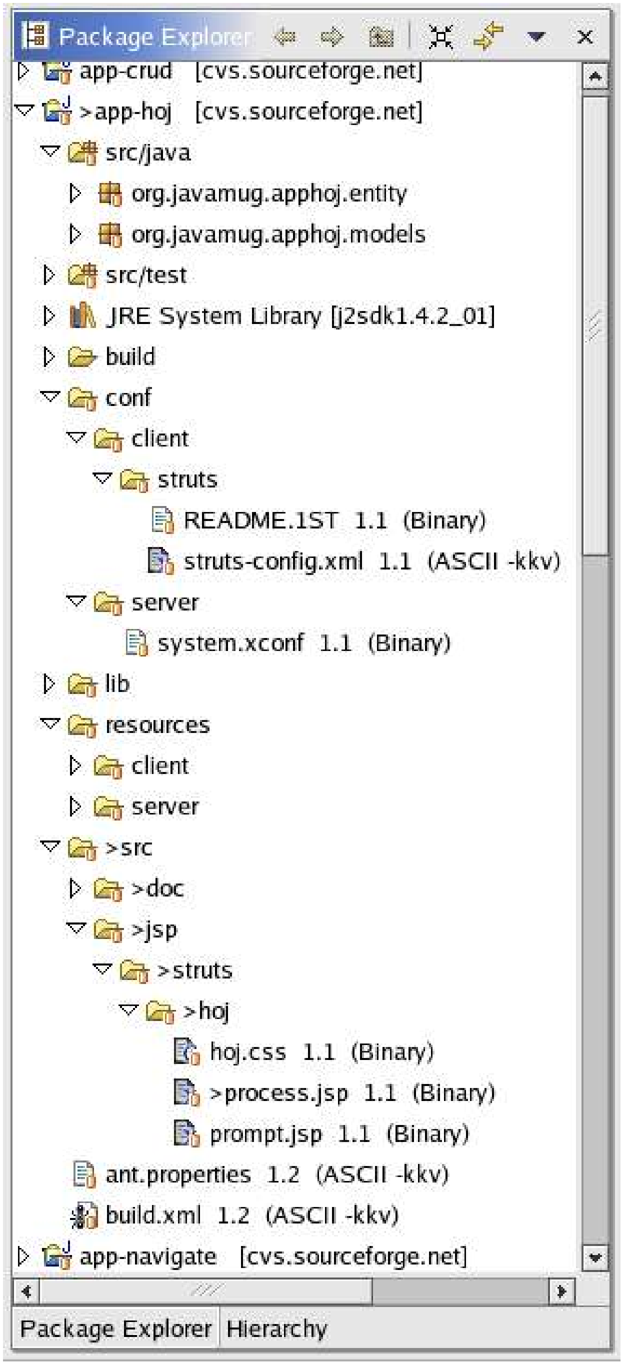
Sooner or later you will want to look under the hood to try and find things. It will help to know that using the "default" deployment, Keel will be deployed in KEEL/keel-build/deploy/jakarta-tomcat-x.y.z/wbeapps/struts.jar, which, when Tomcat runs, will be expanded to KEEL/keel-build/deploy/jakarta-tomcat-x.y.z/webapps/struts. Tomcat, Struts and Keel client-side error messages will be visible on the browser or in KEEL/keel-build/deploy/jakarta-tomcat-x.y.z/logs.

The Keel server is deployed in KEEL/keel-build/deploy/jakarta-tomcat-x.y.z/webapp/struts/WEB-INF/keel. Server-side The server-side configuration files are located in the server/conf directory, while server-side logs are to be found in server/log. All the Keel server JARs are in server/lib.

Skeletal Application

Setting-up Directory Structure

This chapter explains how to create the application structure - the hard way. However, the intent of this chapter is to introduce the structure of a Keel application in detail. Once the pieces are understood, the app-blank shortcut may be utilized, but hopefully the pieces will make more sense.



```
▷ app-crud  [cvs.sourceforge.net]
▽ >app-hoj  [cvs.sourceforge.net]
  ▽ src/java
    ▷ org.javamug.apphoj.entity
    ▷ org.javamug.apphoj.models
  ▷ src/test
  ▷ JRE System Library [j2sdk1.4.2_01]
  ▷ build
  ▽ conf
    ▽ client
      ▽ struts
          README.1ST  1.1  (Binary)
          struts-config.xml  1.1  (ASCII -kkv)
    ▽ server
        system.xconf  1.1  (Binary)
  ▷ lib
  ▽ resources
    ▷ client
    ▷ server
  ▽ >src
    ▷ >doc
    ▽ >jsp
      ▽ >struts
        ▽ >hoj
            hoj.css  1.1  (Binary)
            >process.jsp  1.1  (Binary)
            prompt.jsp  1.1  (Binary)
    ant.properties  1.2  (ASCII -kkv)
    build.xml  1.2  (ASCII -kkv)
▷ app-navigate  [cvs.sourceforge.net]
```

Figure 1- Directory Structure

To make use of Keel's supplied build system, the application must be rooted at the same level as all the other Keel modules. For this hands-on tutorial, we will create the directory structure as in the accompanying figure:

- app-hoj - This is the root directory of the application
- app-hoj/src/java – This is where Java source goes
- app-hoj/src/test – This is where unit and functional tests go
- app-hoj/build - This is where build output goes
- app-hoj/client – This where client-side configuration goes
- app-hoj/server – This is where server-side configuration goes
- app-hoj/lib – This is where JAR needed by this subsystem goes
- app-hoj/resources/client – This is where client-side resources needed by this application go
- app-hoj/resources/server – This is where server-side resources needed by this application go
- app-hoj/src/doc – This is where documentation files (Docbook, PDF, OpenOffice etc.) files go
- app-hoj/src/jsp/struts – This is where JSP files go. By convention, an additional directory struts/hoj is added to keep each applications files distinct

Directory Structure - The Easy Way(s)

Now that you know how to create the directory structure for a Keel application, it is time to know about the tools that make the job easy. First, if you use the Eclipse IDE, on Keel's SourceForge site site there is a plugin that creates the Keel directory structures for you. Secondly, in KEEL/keel-build, there is build-mdoule.xml Ant script that will allow you to create a blank application or service. Finally, the Keel distribution includes a template application, app-blank, which can be copied and modified to start your new application.

Adapting to Keel's build-system

Changing build.xml

The first thing to do is to change the application's Ant build file, build.xml. There is only a small change required, which is to change the name attribute of the project element at the very top of build.xml to the name of the application: "app-hoj".

```
<!-- Ant build file for the Hoj App Subsystem of the Keel meta-framework project
-->

<project name="app-hoj" default="Usage" basedir="../keel-build">
```

Customizing Ant targets

Keel's build system is made to adapt to a highly configurable list of modules to be included in the final deployment. The build system expects to have arbitrary applications and services to be added to Keel at each development site. The place in the build system the customization is defined is in KEEL/keel-build/custom.xml. To keep from updated Keel distributions from overwriting your customizations, copy custom.xml to local-custom.xml. Open local-custom.xml and observe that sets of applications are defined as follows:

```
    <!-- Define the sub-systems that fall in the "application" category -->
    <target name="app-set">
        <antcall target="app-crud"/>
        <antcall target="app-navigate"/>
        <antcall target="app-poll"/>
        <antcall target="app-register"/>
        <antcall target="app-security"/>
        <antcall target="app-workflow"/>
    </target>
```

Simply add the new application along with the predefined ones, as follows:

```
    <!-- Define the sub-systems that fall in the "application" category -->
    <target name="app-set">
        <antcall target="app-crud"/>
        <antcall target="app-hoj"/>
        <antcall target="app-navigate"/>
        <antcall target="app-poll"/>
        <antcall target="app-register"/>
        <antcall target="app-security"/>
        <antcall target="app-workflow"/>
    </target>
```

Scroll down further in local-custom.xml, and find the section that looks like the following:

```
    <!-- Targets for Applications subsystems -->
    <target name="app-crud" if="app.crud">
        <ant antfile="${basedir}/../app-crud/build.xml"
            target="${target-to-run}"/>
    </target>
    <target name="app-navigate" if="app.navigate">
        <ant antfile="${basedir}/../app-navigate/build.xml"
            target="${target-to-run}"/>
    </target>
```

Add the new application following the template:

```
    <!-- Targets for Applications subsystems -->
    <target name="app-crud" if="app.crud">
        <ant antfile="${basedir}/../app-crud/build.xml"
            target="${target-to-run}"/>
    </target>
    <target name="app-hoj" if="app.hoj">
        <ant antfile="${basedir}/../app-hoj/build.xml"
            target="${target-to-run}"/>
    </target>
    <target name="app-navigate" if="app.navigate">
        <ant antfile="${basedir}/../app-navigate/build.xml"
            target="${target-to-run}"/>
    </target>
```

Deployment properties

A myriad of deployment options, including which subsystems to include in a Keel deployment, is defined in what are known as deployment properties files. For this hands-on, we are going to rely on the default deployment properties, which is defined in the file KEEL/keel-build/default-deploy.properties. As before, copy default-deploy.properties to local-deploy.properties. Open local-deploy.properties and navigate to the section shown below:

```
# Now specify whether or not to include each of the following applications.
# You can also include any other custom properties required by custom.xml here
#If you have defined your own application, the property to trigger it to be
#included would be set to true here as well. E.g. app.myapp=true
#Each of the properties below includes the app from the module named "app-xyz"
#where the property is app.xyz. E.g. app.crud=true includes app-crud, etc. See
#the documentation for each module for a description of what each app does.
app.crud=true
app.poll=true
app.register=true
app.navigate=true
app.security=true
app.workflow=true
```

As before, add the new application following the template:

```
# Now specify whether or not to include each of the following applications.
# You can also include any other custom properties required by custom.xml here
#If you have defined your own application, the property to trigger it to be
#included would be set to true here as well. E.g. app.myapp=true
#Each of the proeprties below includes the app from the module named "app-xyz"
#where the property is app.xyz. E.g. app.crud=true includes app-crud, etc. See
#the documentation for each module for a description of what each app does.
app.crud=true
app.hoj=true
app.poll=true
app.register=true
app.navigate=true
app.security=true
app.workflow=true
```

Finally, the build system needs to know to use the local-custom.xml file instead of the standard custom.xml. To do that, in local-properties.xml, navigate to the section:

```
#The name of the custom ant file that specifies the Keel/Custom subsystems to
build
#If you have defined your own applications and/or services, you will need
#to make a custom version of this file (copy the original), and change this
#property to point to this custom copy. (E.g. custom.file=custom-local.xml).
custom.file=custom.xml
```

Change the custom.file property to point to local-custom.xml as follows:

```
#The name of the custom ant file that specifies the Keel/Custom subsystems to
build
#If you have defined your own applications and/or services, you will need
#to make a custom version of this file (copy the original), and change this
#property to point to this custom copy. (E.g. custom.file=custom-local.xml).
custom.file=local-custom.xml
```

Hello World in Keel

Now that we have gone through the tedious job of setting up the infra-structure for Keel, it is time to delve into some fun. The first task is to create the customary "Hello World" application.

Implementing the Model

In the best tradition of Keel, we are at first not going to worry about the user-interface of our application. Instead, the focus is on creating small chunks of business logic, called "Models" in Keel. A model's function is to accept attributes and parameters in a request, process the request, and to create a response. A response consists of outputs, commands, and inputs.

Models in Keel are classes that implement a org.keel.services.model.Model interface. There is an abstract class, org.keel.service.model.StandardModel, which implements the Model interface, as well as Avalon's Configurable interface. A further specialization of the StandardModel class is the StandardLogEnabledModel which additionally implements the LogEnabled interface. Purely for the sake of convenience, our first model is going to be an extension of the StandardLogEnabledModel.

Create the package org.javamug.hoj.models in KEEL/app-hoj/src/java. Create a new Java file, HelloWorld.java, as follows:

```
/*
 * Copyright (c) 2002, The Keel Group, Ltd. All rights reserved.
 *
 * This software is made available under the terms of the license found
 * in the LICENSE file, included with this source code. The license can
 * also be found at:
 * http://www.keelframework.org/LICENSE
 */
package org.javamug.apphoj.models;

import org.keel.services.model.ModelException;
import org.keel.services.model.ModelRequest;
import org.keel.services.model.ModelResponse;
import org.keel.services.model.StandardLogEnabledModel;

/**
 * A simple model that creates one output, named "hello",
 * which contains the string "Hello World".
 *
 * @version $Revision: 1.5 $    $Date: 2003/09/22 15:21:27 $
 * @author Schatterjee
 * Created on Jul 26, 2003
 */
public class HellowWorld extends StandardLogEnabledModel {

    /**
     * @see org.keel.services.model.Model#execute
(org.keel.services.model.ModelRequest)
     */
    public ModelResponse execute(ModelRequest request) throws ModelException {
        ModelResponse res = request.createResponse();
        res.addOutput("hello", "Hello World");
        return res;
    }

}
```

A model has only one required method, "execute" as shown in the listing. A ModelResponse object is created using the createResponse() method of the passed in ModelRequest. The required output is then added to the response, and the response returned to the caller.

Adding roles

A Keel model is, at its roots, an Avalon component that implements the Model service (role). Meta-data needs to be provided with each component that provides the Keel container with enough information to allow lookup of each component, and some hints about the expected lifecycle. To do this, we add the following meta-data in the class javadoc section:

```
*
* @avalon.component
* @avalon.service type=org.keel.services.model.Model
* @x-avalon.info name=hoj.hello
* @x-avalon.lifestyle type=singleton
*
```

The meaning of each meta-data item is as follows:

- @avalon.component - This indicates that the HelloWorld model is a component
- @avalon.service - The type attribute provides the class that implements the role that this component implements
- @x-avalon.info - The name attribute specifies the shorthand (or, hint) that this component can be looked up by
- @x-avalon.lifestyle - The type attribute provides hints about creation and deletion of the component. "singleton" means that only one instance of the component will be created, and every lookup will return the same instance. Other choices are: "transient", where a new instance will be created on every lookup; "pooled", where a pool of instances will be shared; and, "thread", where a new instance will be allocated per thread that does a lookup.

For more information on role meta-data, refer to http://66.105.113.115/vqwiki-2.3.5/jsp/Wiki?ComponentXdoclet

Adding model meta-data

All Keel components, to be looked up as a service from the container, must be defined in a configuration file. Typically called system.xconf, or a filename that ends in system.xconf, the configuration is found in every Keel modules conf/server directory. For models, this configuration can be automatically generated through the use of more meta-data. In this case, we'll add:

```
* @model.model
*   name="hoj.hello"
*   id="hoj.hello"
*   logger="hoj"
*
```

The meaning of each attribute of the @model.model tag is as follows:

- name - This refers to the model shorthand defined in the roles meta above
- id - The same model can be defined with differing configurations, and the id uniquely identifies each one

- logger - The logger category that logs will be output to

For more information on model meta-data tags, refer to http://66.105.113.115/vqwiki-2.3.5/jsp/Wiki?ModelXdoclet

Adding ant.properties file

In order for Keel to utilize xdoclet to properly deploy your application, ant will need to be instructed to do so. Be sure to have an ant.properties file with the following property in it.

has.modelmeta=true

Building

Change directory over to KEEL/keel-build, and type:

```
ant assemble-deploy
```

Ant should start the Keel build process, and will immediately ask for the deployment properties as follows:

```
$ ant assemble-deploy
Buildfile: build.xml

-test-java:
     [echo] Java version 1.4.2

-bad.java:

-test-ant:
     [echo] Ant version Apache Ant version 1.5.3 compiled on April 16 2003

-bad.ant:

-init:

assemble-deploy:

-input-deploy:
    [input] Deployment name:
```

At the "Deployment name:" prompt, type "local" (i.e. the prefix for the *-deploy.properties file) and press the enter key. Ant will proceed to build using local-deploy.properties and local-custom.xml as we have customized before. The build takes a bit of time, at the end of which you should see something similar to:

```
-deploy-client:
     [echo] Client for struts deployed.

assemble-deploy:

BUILD SUCCESSFUL Total time: 2 minutes 39 seconds
```

It takes a bit of time to assemble and deploy all the pieces of Keel. For most development, usually only one module is worked on at a time. As lomg as "ant assemble-deploy"" has been run once, some development time can be saved by only assembling the changed module. For example:

```
cd KEEL/app-hoj

ant -D"deploy.name=local" shortcut-assemble-deploy
```

Although this significantly cuts down on the build time, the shortcut must be used with caution or unexpected behavior might cause delays in a different way: use with understanding and caution.

Starting Tomcat

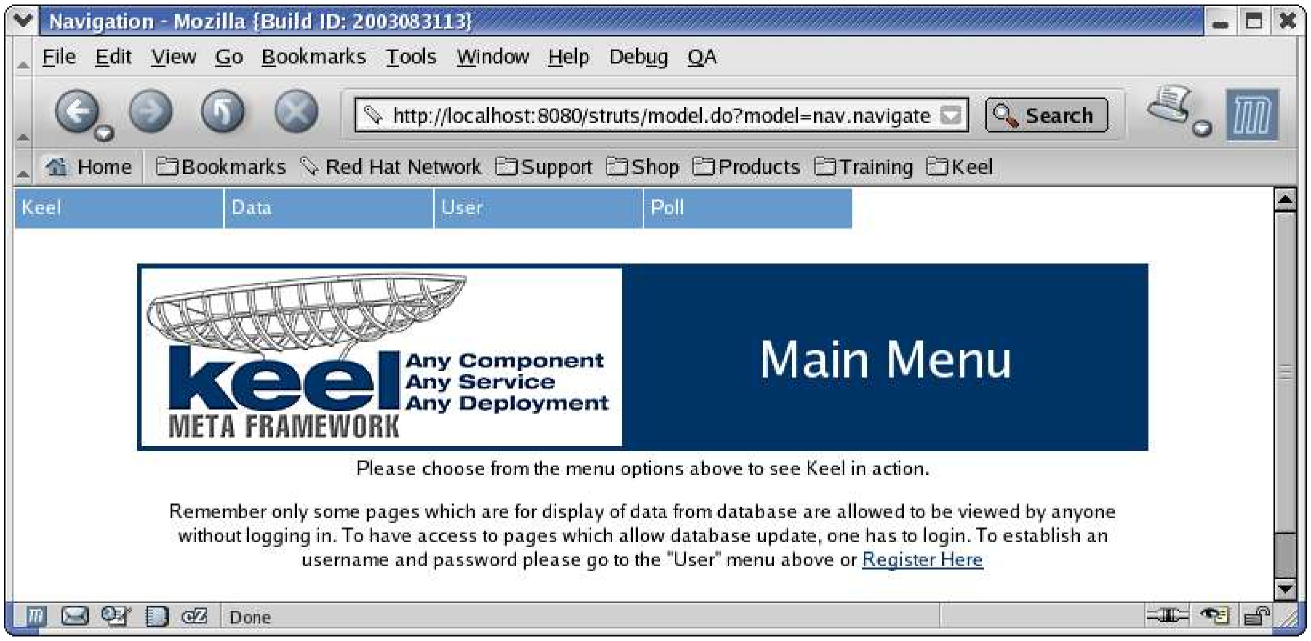
Now that the build is complete, and the WAR file deployed to Tomcat, we can start up Tomcat and see if we can get our example to work. Navigate to the KEEL/keel-build and type:

```
ant tomcat-start
```

The Tomcat logs should show:

```
Sep 27, 2003 8:39:33 AM org.apache.commons.modeler.Registry loadRegistry
INFO: Loading registry information
Sep 27, 2003 8:39:33 AM org.apache.commons.modeler.Registry getRegistry
INFO: Creating new Registry instance
Sep 27, 2003 8:39:35 AM org.apache.commons.modeler.Registry getServer
INFO: Creating MBeanServer
Sep 27, 2003 8:39:37 AM org.apache.coyote.http11.Http11Protocol init
INFO: Initializing Coyote HTTP/1.1 on port 8080
Starting service Tomcat-Standalone
Apache Tomcat/4.1.27
Sep 27, 2003 8:39:42 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='org.apache.struts.util.LocalStrings', returnNull=true
Sep 27, 2003 8:39:42 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='org.apache.struts.action.ActionResources',
returnNull=true
Sep 27, 2003 8:39:43 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='org.apache.webapp.admin.ApplicationResources',
returnNull=true
Sep 27, 2003 8:39:54 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='org.apache.struts.util.LocalStrings', returnNull=true
Sep 27, 2003 8:39:54 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='org.apache.struts.action.ActionResources',
returnNull=true
Sep 27, 2003 8:39:55 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='ApplicationResources', returnNull=true
Sep 27, 2003 8:39:55 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='CrudApplicationResources', returnNull=true
Sep 27, 2003 8:39:55 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='SecurityApplicationResources', returnNull=true
Sep 27, 2003 8:39:55 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='NavigateApplicationResources', returnNull=true
Sep 27, 2003 8:39:55 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='PollApplicationResources', returnNull=true
Sep 27, 2003 8:39:55 AM org.apache.struts.util.PropertyMessageResources <init>
INFO: Initializing, config='RegisterApplicationResources', returnNull=true
Sep 27, 2003 8:39:56 AM org.apache.struts.tiles.TilesPlugin init
INFO: Tiles definition factory loaded for module ''.
Sep 27, 2003 8:39:56 AM org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on port 8080
Sep 27, 2003 8:39:56 AM org.apache.jk.common.ChannelSocket init
INFO: JK2: ajp13 listening on /0.0.0.0:8009
Sep 27, 2003 8:39:57 AM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=33/322
config=/home/schatterjee/java/workspaces/keel/keel-build/deploy/jakarta-tomcat-
4.1.27/conf/jk2.properties
```

Once Tomcat is running properly, open your browser and access http://localhost:8080/struts/model.do?model=nav.navigate Keel starts up when the first request is made, and so the first request takes a little while. At the end of Keel initialization, you should see the main Navigation menu of Keel.

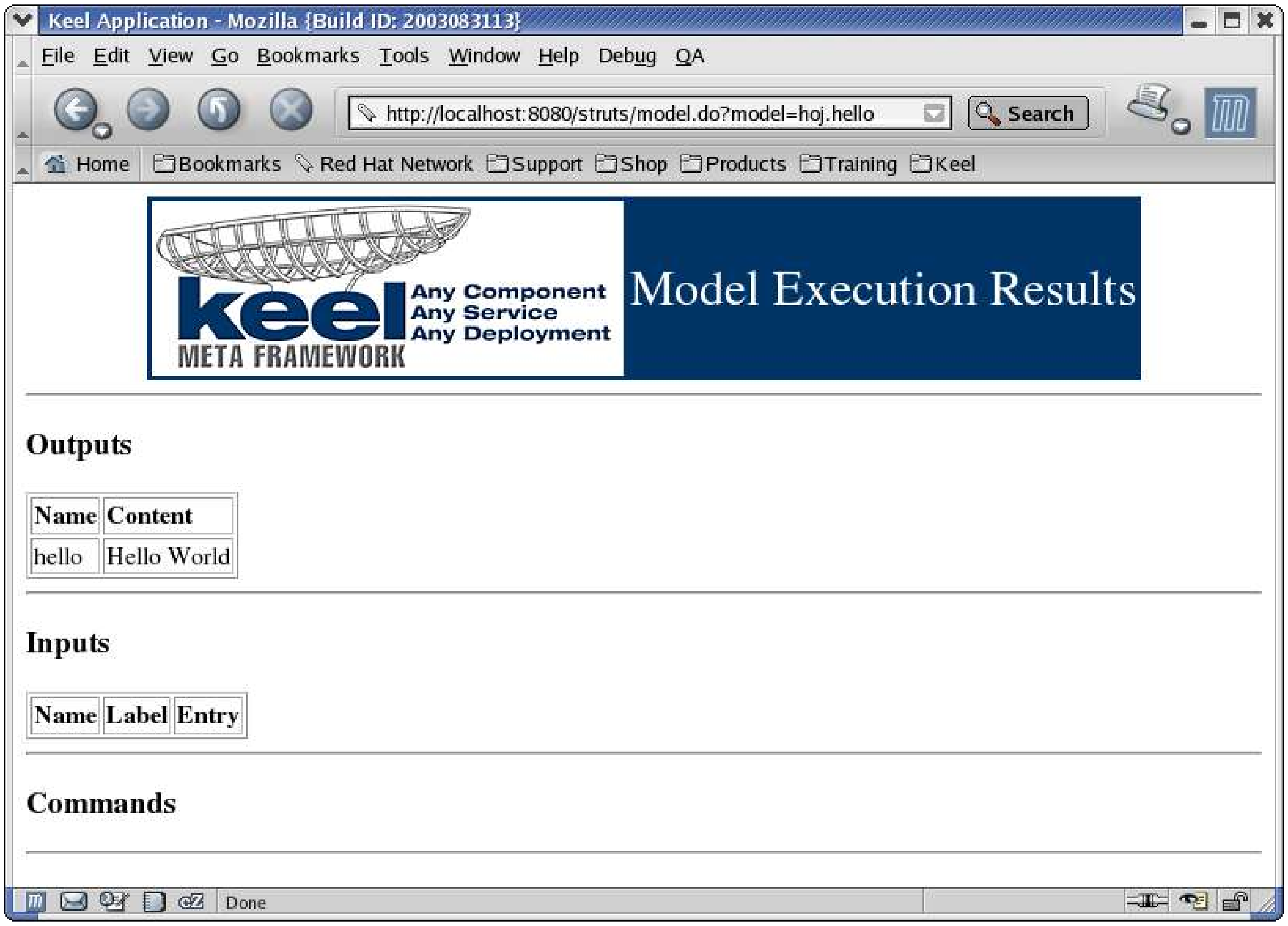

Navigation - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=nav.navigate
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
Keel Data User Poll
keel META FRAMEWORK
Any Component Any Service Any Deployment
Main Menu
Done

Please choose from the menu options above to see Keel in action.

Remember only some pages which are for display of data from database are allowed to be viewed by anyone without logging in. To have access to pages which allow database update, one has to login. To establish an username and password please go to the "User" menu above or Register Here

Figure 2- The Keel Navigation Menu

Once the main navigation menu is up, and we know Keel is setup properly, access http://localhost:8080/struts/model.do?model=hoj.hello. If everything works, we should see the fruits of our hard work:


Keel Application - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=hoj.hello
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
keel META FRAMEWORK
Any Component Any Service Any Deployment
Model Execution Results
Done

Outputs

Name	Content
hello	Hello World

Inputs

Name	Label	Entry

Commands

Figure 3- The Output of HelloWorld Model

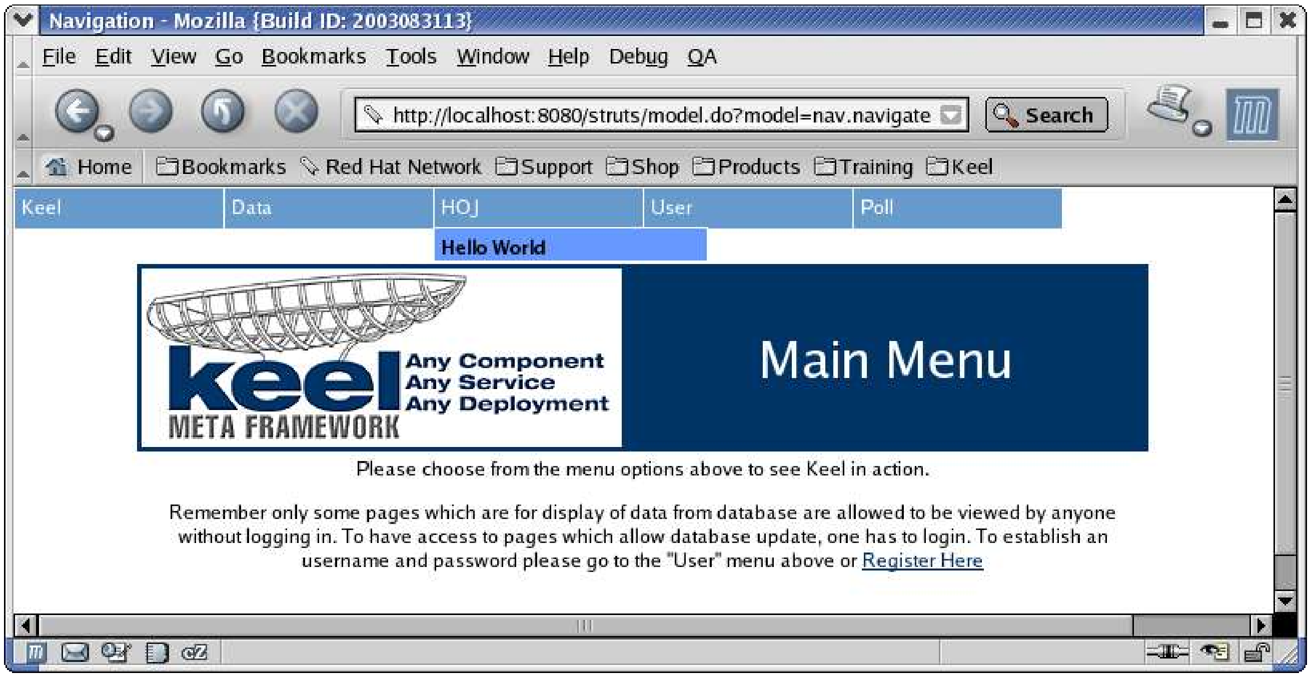
Adding to the menu

Before we finish this section, let's take one more step and add our application/model to Keel's navigation menu. So far, we have not needed to change any configuration files, but now is the time. Each application's server-side configuration goes in conf/server, and is in one ore more files that end in "system.xconf". Keel's configuration process assembles and merges all the configuration snippets into one master configuration that is then processed by the Keel container. Create KEEL/app-hoj/conf/server/system.xconf, and put the following text in it:

```
<keel>

    <nav.navigate id="nav.navigate">
        <menu id="top">
            <menu title="HOJ" id="hoj">
                <menu title="Hello World" model="hoj.hello"/>
            </menu>
        </menu>
    </nav.navigate>

</keel>
```

The nav.navigate configuration is actually defined in the system.xconf for app-navigate, in KEEL/app-navigate/conf/server/system.xconf. It might be worthwhile to open that file and see how it is defined. In this app, we have repeated the nesting of nav.navigate and the nested "top" menu elements with id attributes, Keel will merge the elements with identical id attributes together. The real change we wish to accomplish here is to add a top-level menu item with title "HOJ" and then a nested menu-option "Hello World" which, when accessed, will invoke our "hoj.hello" model.

Make sure Tomcat is stopped. Go through Keel's assemble-deploy and startup Tomcat again, as we did before. Once Tomcat is ready, access http://localhost:8080/struts/model.do?model=nav.navigate again. This time we should see our new menu option:


Navigation - Mozilla {Build ID: 2003083113}

http://localhost:8080/struts/model.do?model=nav.navigate

Main Menu

Please choose from the menu options above to see Keel in action.

Remember only some pages which are for display of data from database are allowed to be viewed by anyone without logging in. To have access to pages which allow database update, one has to login. To establish an username and password please go to the "User" menu above or Register Here

File Edit View Go Bookmarks Tools Window Help Debug QA
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
Keel Data HOJ User Poll
Hello World
keel META FRAMEWORK
Any Component
Any Service
Any Deployment

Figure 4- Hello World Added to Menu

Configuration

There are two things we are going to attempt to accomplish in this step. The first one is that we are going to demonstrate how configuration is provided to a component and how it is accessed. The second is to demonstrate the capability of Keel to have the same component be looked up using two different IDs, and behave completely different. In the first case, we'll let the model have default values, in the second case we'll provide configured values that will be used instead.

Changing the model

To show the progressive steps, instead of changing our original HelloWorld class, create a new class ConfigHelloWorld just as before. StandardLogEnabledModel/StandardModel already implement the configurable interface, so what we need to do now is to override the "configure" method, and add String variable to hold the message:

```
    private String msg = null;

    /* Override StandardLogEnabled's configure method.
      * This will be called by the Keel container when the
      * model is instantiated, and the appropriate configuration
      * from model meta-data or from *-system.xconf will be passed
      * in.
      * @see org.apache.avalon.framework.configuration.Configurable#configure
(org.apache.avalon.framework.configuration.Configuration)
      */
    public void configure(Configuration configuration)
        throws ConfigurationException {
        super.configure(configuration);

        msg = configuration.getChild("hello", true).getAttribute("value",
"Configurable Hello World");
    }
```

What we do here is simply expect an optional XML configuration element called "hello", which will have an attribute called "value". If no configuration is provided, "Configurable Hello World" will be the default value.

Altering Meta-data

Create role-meta as before, but since this is a new model provide a new shorthand for it. The @avalon.info tag should now look like:

```
 * @x-avalon.info name=hoj.confighello
```

The model meta-data must also match this new shorthand:

```
 * @model.model
 *   name="hoj.confighello"
 *   id="hoj.confighello"
 *   logger="hoj"
```

Configuring the Second Shorthand

We are going to now address the second objective for this section. We have already created our new configurable model, and set it up as we did our first model. When run, we expect the output to show

"Configurable Hello World", since we are not going to provide any altered configuration. Now, we are going to take the same class, and configure the Keel container to create another component with a different shorthand, but this time with specified configuration.

Edit system.xconf again, and add the following configuration:

```
    <hoj.confighello id="hoj.confighello2" activation="request" logger="hoj">
        <hello value="Hey, we did configure it!"/>
    </hoj.confighello>
```

What we did here was to use the same shorthand for the component as defined in the role-meta (i.e. "hoj.confighello"), but this time, we use a different id (i.e. "hoj.confighello2") from the id specified in the model-meta (i.e. "hoj.confighello"). We also provide our configuration for the hello string to print out. As an aside, can you look at the configuration above, compare it to the model-meta specified and guess what configuration the meta-data would generate?

Adding New Components to Menu

For the sake of completeness, we'll go through the optional step of adding this new model to our menu.

```
    <nav.navigate id="nav.navigate">
        <menu id="top">
            <menu title="HOJ" id="hoj">
                <menu title="Hello World" model="hoj.hello"/>
                <menu title="Configurable Hello World" model="hoj.confighello"/>
                <menu title="Configurable Hello World 2"
model="hoj.confighello2"/>
            </menu>
        </menu>
    </nav.navigate>
```

Building, Deployment, Testing

Build Keel again, deploy, and start Tomcat again. Access the Keel main menu, invoke the two new models and see if we made any headway!

Adding Security

In this section, we are going to take our model and adapt it to Keel authorization.

Create the model

We could simply change our ConfigHelloWorld class, but once more, let us just create a new class SecuredConfigHelloWorld just as before. To recap we need a class that extends StandardLogEnabledModel, overrides the configure method, and contains role and model meta-data. The new class should look like the following:

```
/*
 * Copyright (c) 2002, The Keel Group, Ltd. All rights reserved.
 *
 * This software is made available under the terms of the license found
 * in the LICENSE file, included with this source code. The license can
 * also be found at:
 * http://www.keelframework.org/LICENSE
 */
package org.javamug.apphoj.models;

import org.apache.avalon.framework.configuration.Configuration;
import org.apache.avalon.framework.configuration.ConfigurationException;
import org.apache.avalon.framework.service.ServiceException;
import org.keel.services.authorization.AuthorizationManager;
import org.keel.services.authorization.Securable;
import org.keel.services.model.ModelException;
import org.keel.services.model.ModelRequest;
import org.keel.services.model.ModelResponse;
import org.keel.services.model.StandardLogEnabledModel;

/**
 * This is the same as the ConfigHelloWorld class, except now the
 * model is secured.
 *
 * @avalon.component
 * @avalon.service type=org.keel.services.model.Model
 * @x-avalon.info name=hoj.securehello
 * @x-avalon.lifestyle type=transient
 *
 * @model.model
 *   name="hoj.securehello"
 *   id="hoj.securehello"
 *   logger="hoj"
 *
 * @version $Revision: 1.5 $    $Date: 2003/09/22 15:21:27 $
 * @author Schatterjee
 * Created on Jul 26, 2003
 */
public class SecuredConfigHelloWorld extends StandardLogEnabledModel implements
Securable {

    private String msg = null;

    private AuthorizationManager am = null;

    /**
     * @see org.keel.services.model.Model#execute
(org.keel.services.model.ModelRequest)
     */
    public ModelResponse execute(ModelRequest request) throws ModelException {
        ModelResponse res = request.createResponse();
        res.addOutput("hello", msg);
        return res;
    }

    /**
     * @see org.apache.avalon.framework.configuration.Configurable#configure
(org.apache.avalon.framework.configuration.Configuration)
     */
    public void configure(Configuration configuration)
        throws ConfigurationException {
        super.configure(configuration);

        msg = configuration.getChild("hello", true).getAttribute("value", "Secured
Hello World");
    }
}
```

To make it secure, the class must now extend the org.keel.services.authorization.Securable interface. Along with that, two new methods from the Securable interface must be implemented:

```
    /**
     * @see org.keel.services.authorization.Securable#setAuthorizationManager
(org.keel.services.authorization.AuthorizationManager)
     */
    public void setAuthorizationManager(AuthorizationManager am) throws
ServiceException {
        this.am = am;
    }

    /**
     * @see org.keel.services.authorization.Securable#getAuthorizationManager()
     */
    public AuthorizationManager getAuthorizationManager() {
        return am;
    }
```

The following imports must be added as well:

```
import org.apache.avalon.framework.service.ServiceException;
import org.keel.services.authorization.AuthorizationManager;
import org.keel.services.authorization.Securable;
```

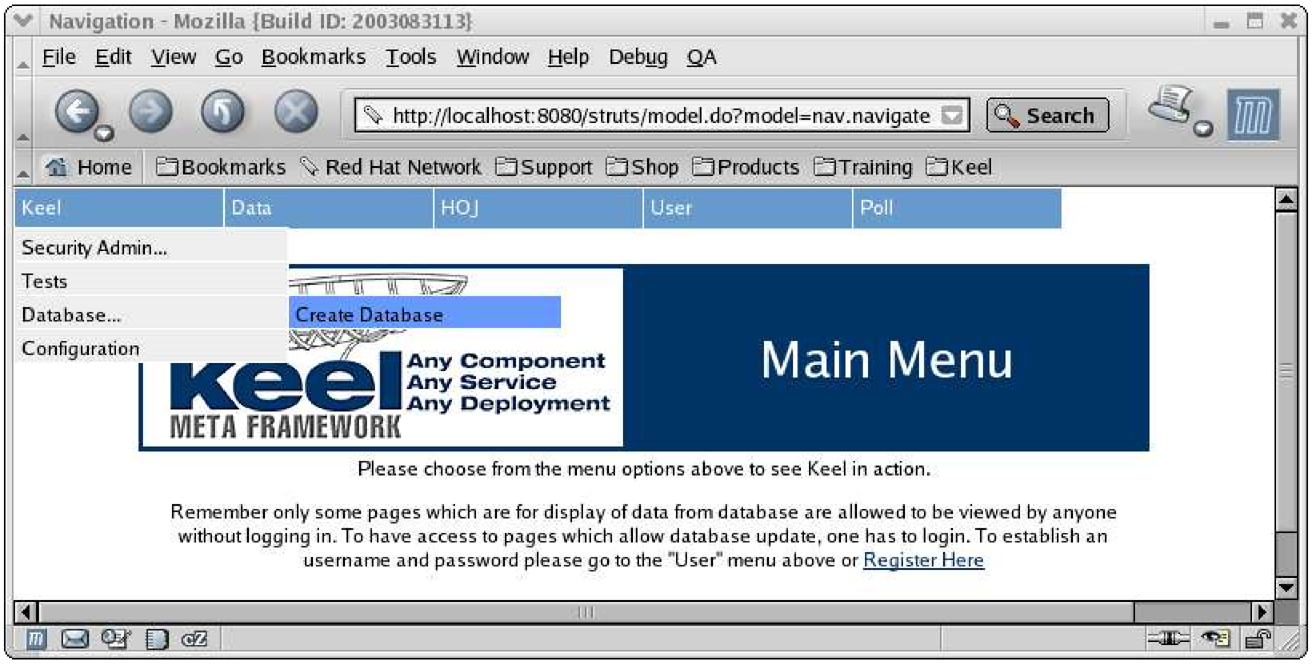
By now we know the routine of adding something to the menu.

```
                <menu title="Secured Hello World" model="hoj.securehello"/>
```

There we have it, that was all that is required to secure a model, or any other Keel component, for that matter.

Setting up security

After Keel is built and deployed again, start up Tomcat and access the main menu as before. So far, we have not done anything which requires Persistence and any of Keels database tables. Now, since the default authorization manager is based on Persistent data, we need to create the default tables. Access the main menu, and invoke Keel->Database->Create database:

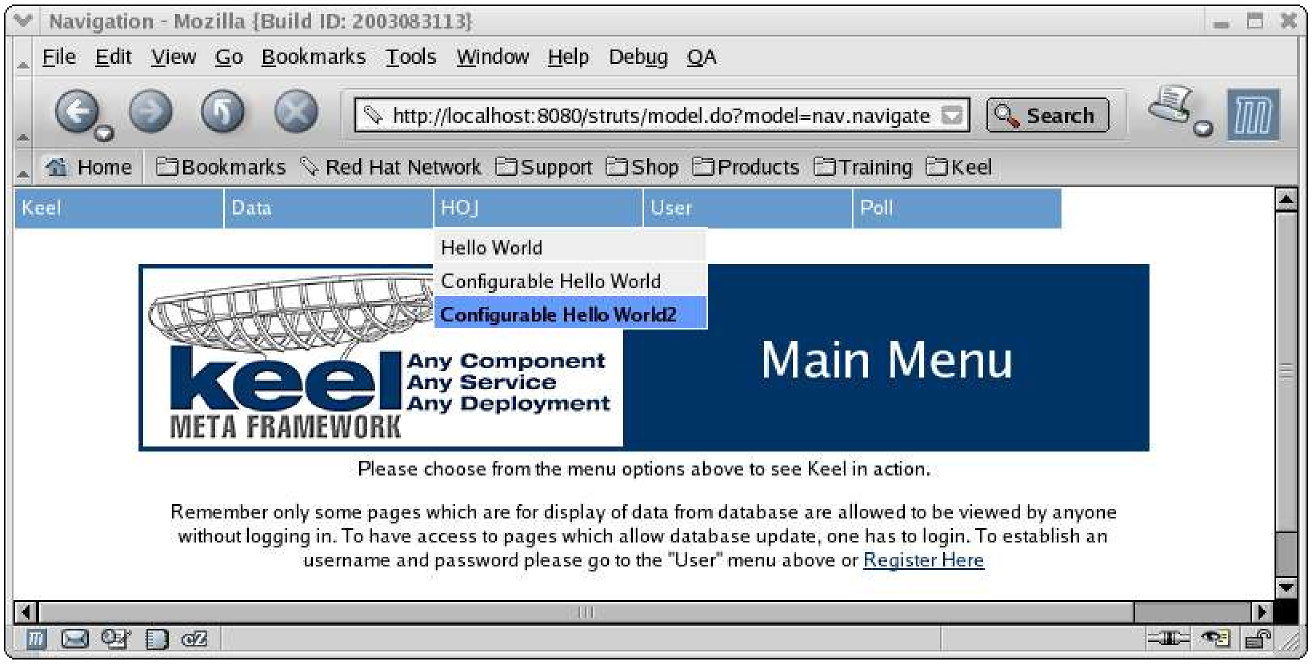

Navigation - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=nav.navigate
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
Keel Data HOJ User Poll
Security Admin...
Tests
Database...
Configuration
Create Database
Any Component Any Service Any Deployment
keel META FRAMEWORK
Main Menu

Please choose from the menu options above to see Keel in action.

Remember only some pages which are for display of data from database are allowed to be viewed by anyone without logging in. To have access to pages which allow database update, one has to login. To establish an username and password please go to the "User" menu above or Register Here

Figure 5- Creating the Default Database

Once the table creation is complete, Keel should show a page with the words "Database Creation Complete" on it. Once that is done, try the HOJ menu, and you should see that the Secured model is missing from the menu.

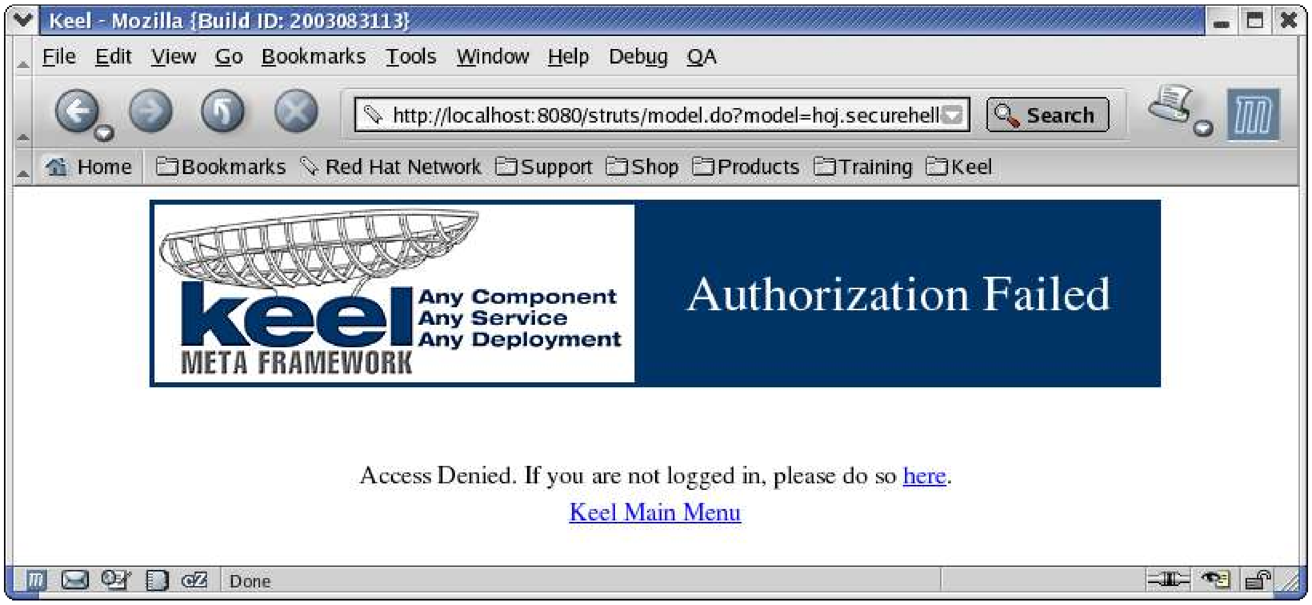

Navigation - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=nav.navigate
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
Keel Data HOJ User Poll
Hello World
Configurable Hello World
Configurable Hello World2
Any Component Any Service Any Deployment
keel META FRAMEWORK
Main Menu

Please choose from the menu options above to see Keel in action.

Remember only some pages which are for display of data from database are allowed to be viewed by anyone without logging in. To have access to pages which allow database update, one has to login. To establish an username and password please go to the "User" menu above or Register Here

Figure 6- Secured Model Absent from Menu

This is no accident. Since permissions have not been granted yet to run that secured model, it does not even show up in the menu. Next, try accessing the model direct with http://localhost:8080/struts/model.do?model=hoj.securehello. This should take you to a page telling you that access was denied.

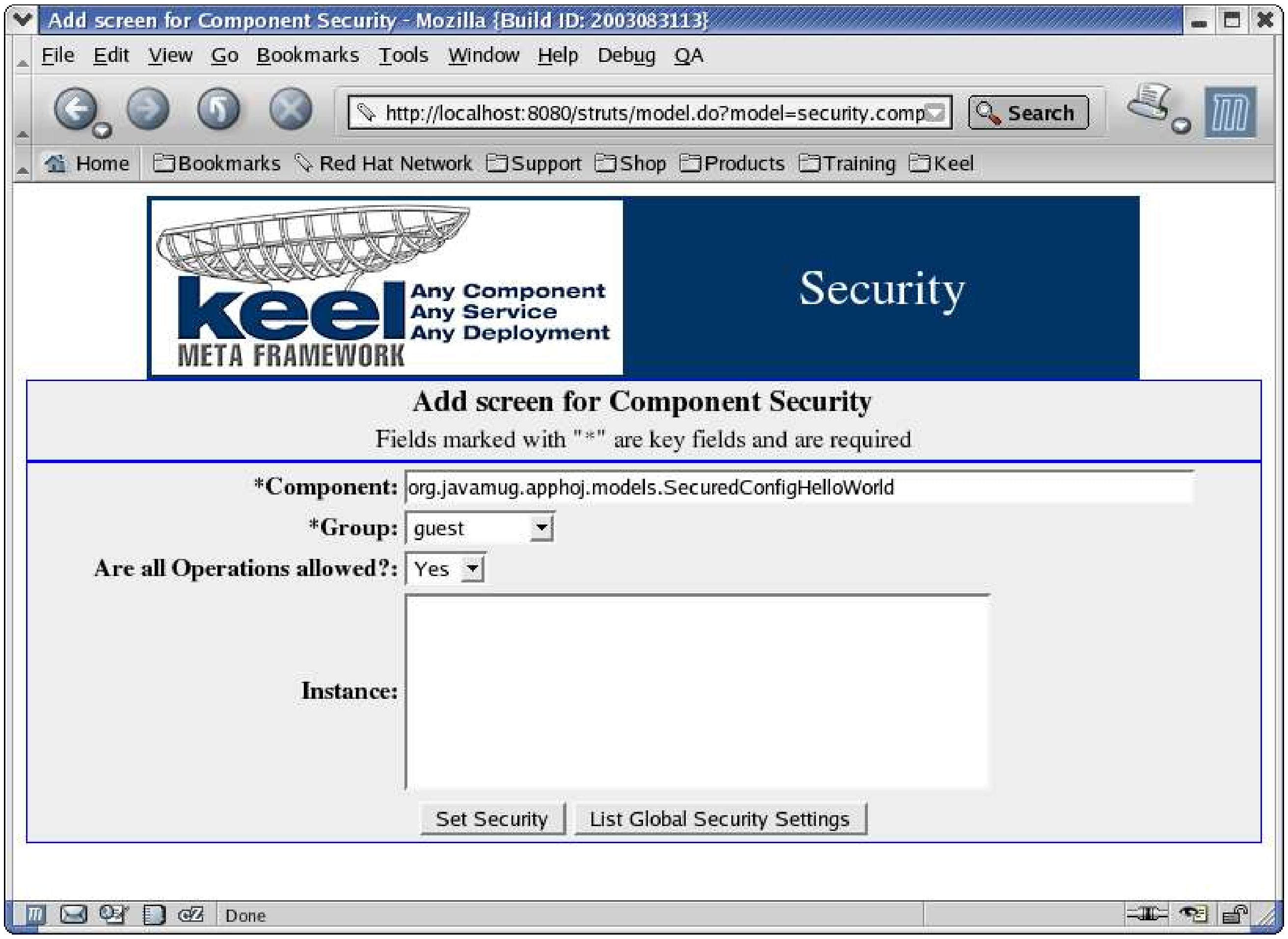

Keel - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=hoj.securehell
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
keel META FRAMEWORK
Any Component
Any Service
Any Deployment
Done

Authorization Failed

Access Denied. If you are not logged in, please do so here.
Keel Main Menu

Figure 7- Access Denied

Go back to the main menu, access User->Log In, and login with the Login "root" and Password "root". From Keel's main menu, invoke Keel->Security Admin->Component Security. In the resulting screen, put the fully qualified classname of the secured model in the "Component" field. For this example, we will allow all users in the group "guest" to have access to this model. We will allow all instances of this model to be accessible, so leave the "Instance" field blank. Alternatively, we could have made the component extend InstanceSecurable instead of Securable, and then secured by specific instance identifier. In this case, the completed form should look like the following figure. When done, click the "Set Security" button.


Add screen for Component Security - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=security.comp
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
keel META FRAMEWORK
Any Component Any Service Any Deployment
Security
Set Security
List Global Security Settings
Done

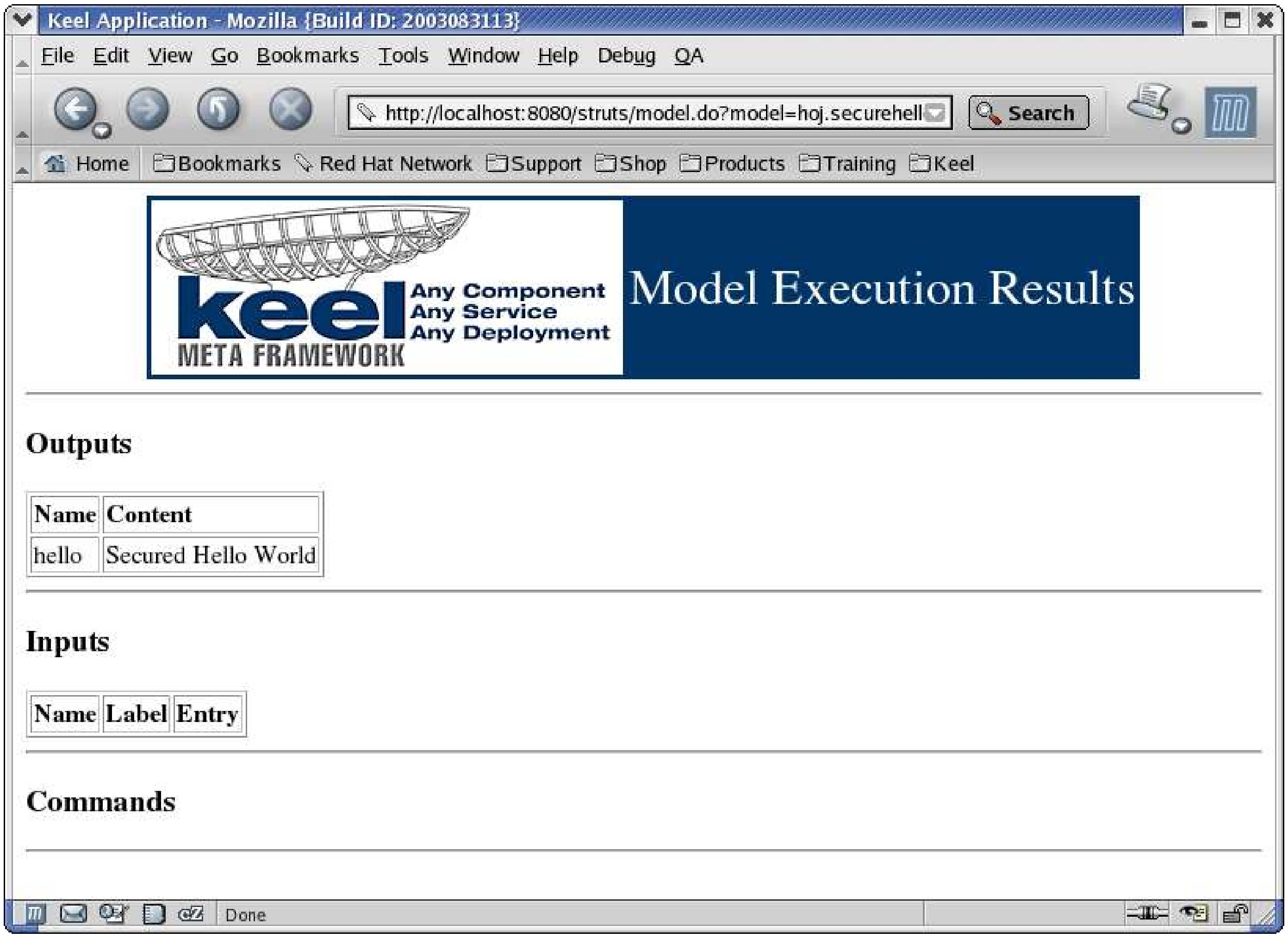
Add screen for Component Security

Fields marked with "*" are key fields and are required

*Component:	org.javamug.apphoj.models.SecuredConfigHelloWorld
*Group:	guest
Are all Operations allowed?:	Yes
Instance:	

Figure 8- Security Setup

Now go back to the Keel main navigation menu, and select User->Log In. Log in with Login "guest", password "guest", which happens to be the only user in the "guest" group. Invoke http://localhost:8080/struts/model.do?model=hoj.securehello, and this time, instead of getting access denied, the following page should display:


Keel Application - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=hoj.securehell
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
keel META FRAMEWORK Any Component Any Service Any Deployment
Done

Model Execution Results

Outputs

Name	Content
hello	Secured Hello World

Inputs

Name	Label	Entry

Commands

Figure 9- Output from Secured Model

Adding Persistence

Most applications require some sort of persistent storage. Keel has multiple implementations of its persistence service: one using it's own object-to-relational mapping, another using the popular Hibernate package, Keel also has a JDO service, using the Xorm package. In this example, we'll use Keel's native persistence implementation. However, it is important to note that, to move over to Hibernate-based persistence, or JDO-based persistence, the entity beans we are going to define are not going to change at all. The example models we'll write will use Keel's persistence service, and therefore when switching between Keel's default persistence and Hibernate, the models would not change at all either. Of course, persistence and JDO are different services, and therefore switching between the Persistent and JDO services would involve a code change.

The Entity Bean

We are going to use a very simple entity bean, which has two fields only. The first is an integer id field, which will uniquely identify each row in the table. The second filed is a string. To do that, create a class Mydata, as follows:

```
/*
 * Copyright (c) 2002, The Keel Group, Ltd. All rights reserved.
 *
 * This software is made available under the terms of the license found
 * in the LICENSE file, included with this source code. The license can
 * also be found at:
 * http://www.keelframework.org/LICENSE
 */
package org.javamug.apphoj.entity;

/**
 * Hoj Mydata
 *
 * @jdo.persistence-capable
 *
 * @hibernate.class
 *   table="mydata"
 *
 * @persist.persistent
 *   schema="hoj"
 *   name="mydata"
 *   descrip="My sample persistent"
 *   table="mydata"
 *
 * @persist.record
 *    mydataId="1"
 *    mystring="This is the first record"
 * @persist.record
 *    mydataId="2"
 *    mystring="This is the second record"
 */
public class Mydata {

    /**
     * @jdo.field
     *   default-fetch-group="true"
     */
    private Integer mydataId = null;

    /**
     * @jdo.field
     *   default-fetch-group="true"
     */
    private String mystring = null;

    /**
     * Returns the string.
     * @hibernate.property
     *   column="mystring"
     *
     * @persist.field
     *   name="mystring"
     *   db-name="mystring"
     *   type="varchar"
     *   length="80"
     *   null-allowed="false"
     *   descrip="$catDescription"
     *
     * @hibernate.property
     *   column="mystring"
     *
     * @return String The string
     */
    public String getMystring() {
        return mystring;
    }

    /**
     * Returns the mydataId.
     *
     * @hibernate.id
     *   column="mydataId"
     *   generator-class="native"
     *
     * @persist.field
     *   name="mydataId"
     *   db-name="mydataId"
     *   read-only="true"
     *   type="integer"
     *   primary-key="true"
     *   null-allowed="false"
     *   descrip="record id"
     *   read0only="true"
     *   auto-increment="table"
     *
     * @return Integer
     */
    public Integer getMydataId() {
        return mydataId;
    }

    /**
     * Sets the string.
     * @param str
     */
    public void setMystring(String str) {
        mystring = str;
    }

    /**
     * Sets the mydataId.
     * @param id
     */
    public void setMydataId(Integer id) {
        this.mydataId = id;
    }

}
```

The structure of this class is very simple. The two fields mentioned above are realized as mydataId and myString. As with any other Java bean, each field has a setter and a getter, named according to bean conventions. The interesting thing to note is that, each field and the setters/getters have meta-data annotated in Javadoc comments, and meta-data can be seen for JDO, Hibernate and Keel's default persistence. More detail about the meta-data format can be found at http://66.105.113.115/vqwiki-2.3.5/jsp/Wiki?XDocletHOWTO.

The Prompter Model

We need a model to prompt for data to be stored in the persistent table. Create a new model, PromptMydata as follows:

```
/*
 * Copyright (c) 2002, The Keel Group, Ltd. All rights reserved.
 *
 * This software is made available under the terms of the license found
 * in the LICENSE file, included with this source code. The license can
 * also be found at:
 * http://www.keelframework.org/LICENSE
 */
package org.javamug.apphoj.models;

import org.keel.services.model.Command;
import org.keel.services.model.ModelException;
import org.keel.services.model.ModelRequest;
import org.keel.services.model.ModelResponse;
import org.keel.services.model.StandardLogEnabledModel;

/**
 * This model prompts for a single string as input data
 * It also provides a command which will allow the user to
 * transition to a model which process the data that was input.
 *
 * @avalon.component
 * @avalon.service type=org.keel.services.model.Model
 * @x-avalon.info name=hoj.prompt-mydata
 * @x-avalon.lifestyle type=singleton
 *
 * @model.model
 *   name="hoj.prompt-mydata"
 *   id="hoj.prompt-mydata"
 *   logger="hoj"
 *
 * @version $Revision: 1.5 $    $Date: 2003/09/22 15:21:27 $
 * @author Schatterjee
 * Created on Jul 26, 2003
 */
public class PromptMydata extends StandardLogEnabledModel {

    /**
     * @see org.keel.services.model.Model#execute
(org.keel.services.model.ModelRequest)
     */
    public ModelResponse execute(ModelRequest request) throws ModelException {
        ModelResponse res = request.createResponse();
        res.addInput("mystring", "Enter some text");

        // The action to take for data submission
        Command a = res.createCommand("hoj.process-mydata");
        a.setName("add");
        a.setLabel("Add Data");
        res.add(a);
        return res;
    }

}
```

Until now we have only used an Output object in responses from the model. We now introduce two new objects. An Input object is an abstraction of a data entry field, which will be returned as a parameter in the request processed by the model that processes that request. A Command object associates a model tat will be transitioned to when that command is chosen. Responses can have multiple Inputs, Outputs and Commands. Commands and Inputs can be nested within Output objects, in which case Outputs are used as nesting containers.

In the prompting model above, we create a text field for entry of the string which will be stored in the persistent table. There can only be a single transition after the data is entered, to the model which will process the entered data and store it into the persistent table.

It is important to note, that the model's inputs, outputs and commands are specified with complete disregard for the actual user-interface technology that will eventually be used. Do not ever assume that the UI is going to be a webapp, or a CLI, or Struts, Cocoon, or anything else at all. Abstraction of the UI is key to Keel's power.

The Processing Model

The next thing we need is a model that will accept the submitted data and store it in our persistent table. To do that, create the final class we need in this tutorial, ProcessMydata. The class should look as follows:

```
/*
 * Copyright (c) 2002, The Keel Group, Ltd. All rights reserved.
 *
 * This software is made available under the terms of the license found
 * in the LICENSE file, included with this source code. The license can
 * also be found at:
 * http://www.keelframework.org/LICENSE
 */

package org.javamug.apphoj.models;
import org.keel.services.model.Command;
import org.keel.services.model.ModelException;
import org.keel.services.model.ModelRequest;
import org.keel.services.model.ModelResponse;
import org.keel.services.model.StandardLogEnabledModel;
import org.keel.services.persist.PersistenceException;
import org.keel.services.persist.Persistent;
import org.keel.services.persist.PersistentFactory;


/**
 * <Replace with description for ProcessMydata>
 *
 * @avalon.component
 * @avalon.service type=org.keel.services.model.Model
 * @x-avalon.info name=hoj.process-mydata
 * @x-avalon.lifestyle type=singleton
 *
 * @model.model
 *   name="hoj.process-mydata"
 *   id="hoj.process-mydata"
 *   logger="hoj"
 * @model.parameter
 *   name="mystring"
 *   required="true"
 *
 * @version $Revision: 1.5 $    $Date: 2003/09/22 15:21:27 $
 * @author Schatterjee
 * Created on Jul 27, 2003
 */
public class ProcessMydata extends StandardLogEnabledModel {

    /**
     * @see org.keel.services.model.Model#execute
(org.keel.services.model.ModelRequest)
     */
    public ModelResponse execute(ModelRequest request) throws ModelException {
        ModelResponse res = request.createResponse();

        // Persistents are created from a factory, the domain
        // is what the user logs into from the login screen
        // The default domain happens to be using Keel's default
        // persistence, but it could be Hibernate just as well.
        PersistentFactory pf =
            (PersistentFactory) request.getService(
                PersistentFactory.ROLE,
                request.getDomain());

        try {
            // This is the actual persistent, see the Mydata entity
            Persistent m = pf.create("hoj.mydata");

            // Set the field we are interested in
            m.setField("mystring", request.getParameter("mystring"));
            // Save to DB
            m.add();

            // Just some confirmation
            res.addOutput("Added data");

            // The transition after confirmation - prompt again
            Command a = res.createCommand("hoj.prompt-mydata");
            a.setName("prompt");
            a.setLabel("Prompt For Data");
            res.add(a);
        } catch (PersistenceException e) {
            throw new ModelException("Error accessing table", e);
        }
        return res;
    }

}
```

The code is a great example of Keel's simplicity. Persistents are created from a PersistentFactory. As the comments show, multiple persistent factories, could be present. Each could be a different database, on same or different hosts. At the same time each domain could be built using one of Keel's different persistence implementations. All that complexity is hidden from application programs by the hint (here, the domain name) provided when the persistence service is looked up from the Keel container.

Once the persistent is created by the factory, it is simply a matter of filling in the data in each field, and then calling the add method. A query is not shown here, but it would consist of setting the key values in appropriate fields and then calling the find method.

Auto-increment configuration

If you look at the meta-data for the mydataId field, it contains a "auto-increment" attribute. That says that the mydataId is to be automatically incremented every time a new record is added to the Mydata table. The auto-increment table must be defined, and the index seeded to start from some number. The required configuration in system.xconf is as follows:

```
     <table id="mydata" big-decimals="false" block-size="1" table="ids"
            key-table="mydata" activation="request">
            <dbpool>keel-dbpool</dbpool>
     </table>

    <default-persistent id="default">
        <schemas id="default">
            <schema id="ids">
                <persistent id="ids">
                    <default-data id="ids-default">
                        <record table_name="mydata" next_id="1"/>
                    </default-data>
                </persistent>
            </schema>
      </schemas>
    </default-persistent>
```

Here's how this works. The persistent meta-data refers to "table" in the auto-increment attribute. The "table" element in the configuration above defines an "ID-generator" which is a feature of the datasource component from Avalon/Excalibur. The ID-generator, in turn refers to a key-table called "mydata". The second set of elements then defines a row in the table of IDs for the "mydata" table, and seeds the starting index at "1".

Adding default data

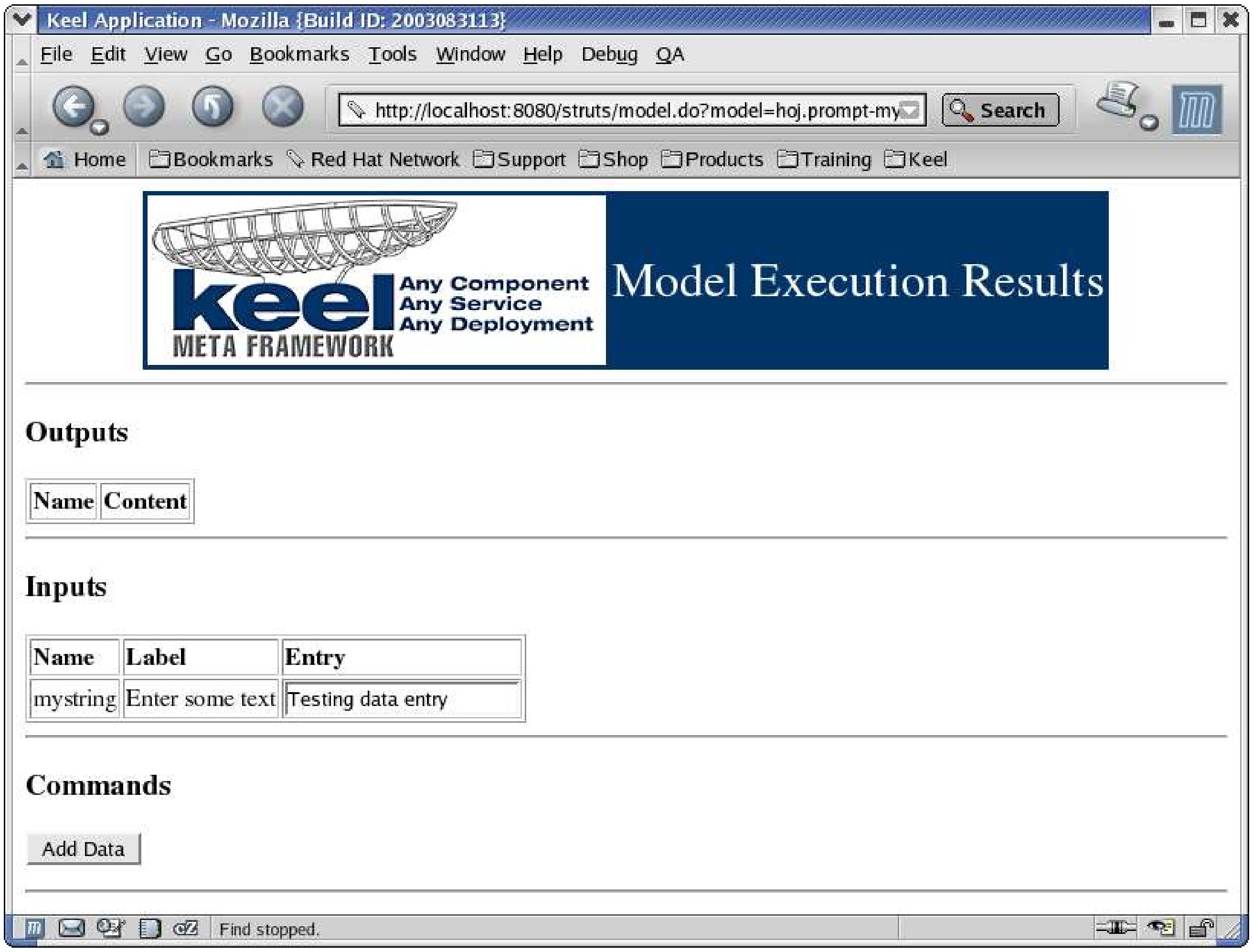
We are not going to do it, but, if needed you could seed the persistent with some default data. This could be done with meta-data in your persistent entity bean, as follows:

```
 * @persist.record
 *   mydataId="1"
 *   mystring="This is the first predefined record"
 * @persist.record
 *   mydataId="2"
 *   mystring="This is the second predefined record"
```

Using the Persistent

We have everything that is needed to test our persistent and associated models. Build Keel, start Tomcat, and access the main navigation menu once again. We have to create our database tables, the id-generators, and seed the predefined data if any are specified. Access Keel->Database...->Create Database as before to create and initialize the required tables. This step needs to be run every time new tables are added in your application(s).

Now invoke the prompt model from the HOJ menu. You should now see a page which displays the "mystring" input created by the model. The screen should look like:


Keel Application - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=hoj.prompt-my
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
keel META FRAMEWORK
Any Component
Any Service
Any Deployment
Find stopped.

Model Execution Results

Outputs

Name	Content

Inputs

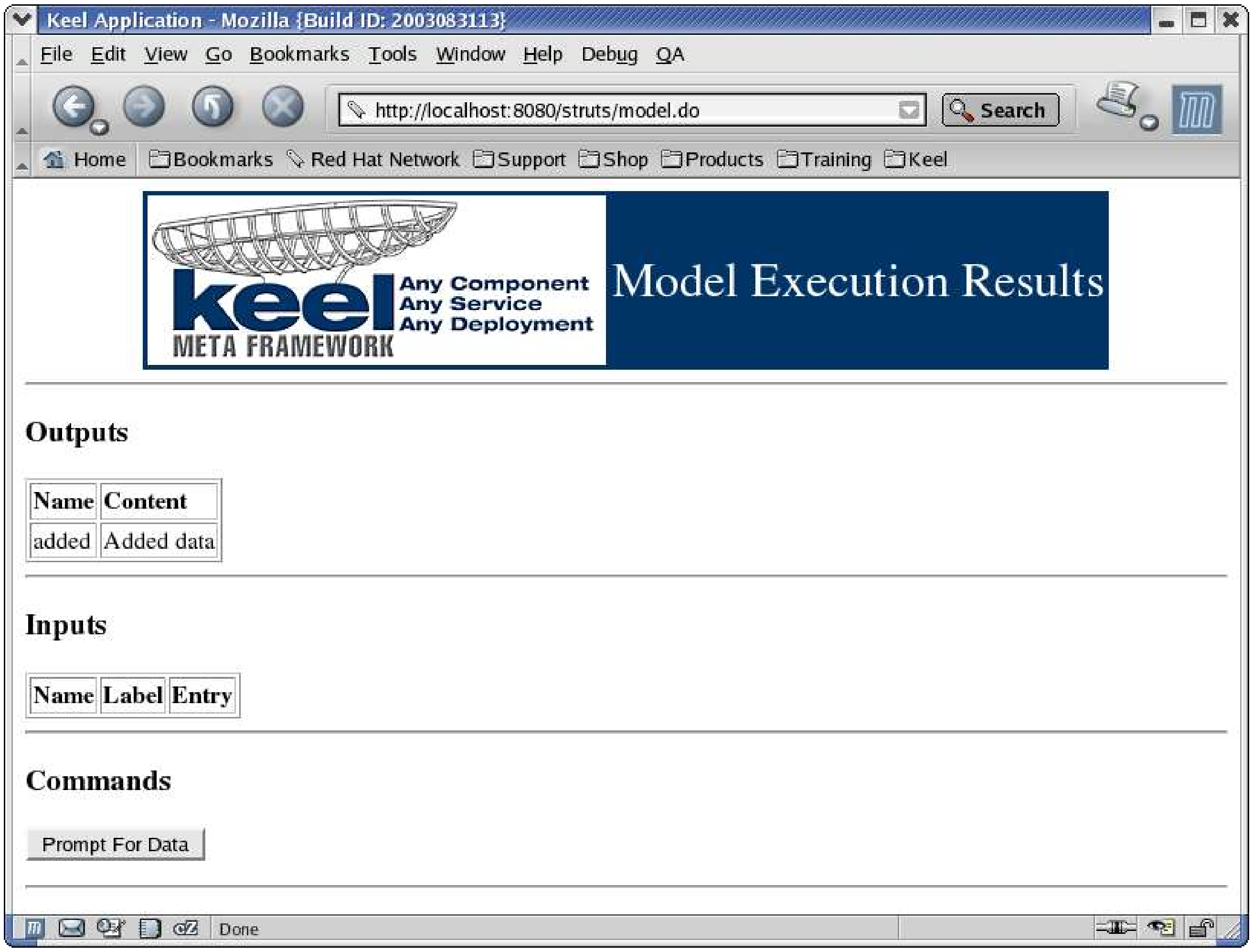
Name	Label	Entry
mystring	Enter some text	Testing data entry

Commands

Add Data

Figure 10- Output from Prompt Model

Pressing the "Add Data" button after entering some test data like "Testing data entry" should add the record in the DB and show the following page:


Keel Application - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
keel META FRAMEWORK
Any Component
Any Service
Any Deployment
Done

Model Execution Results

Outputs

Name	Content
added	Added data

Inputs

Name	Label	Entry

Commands

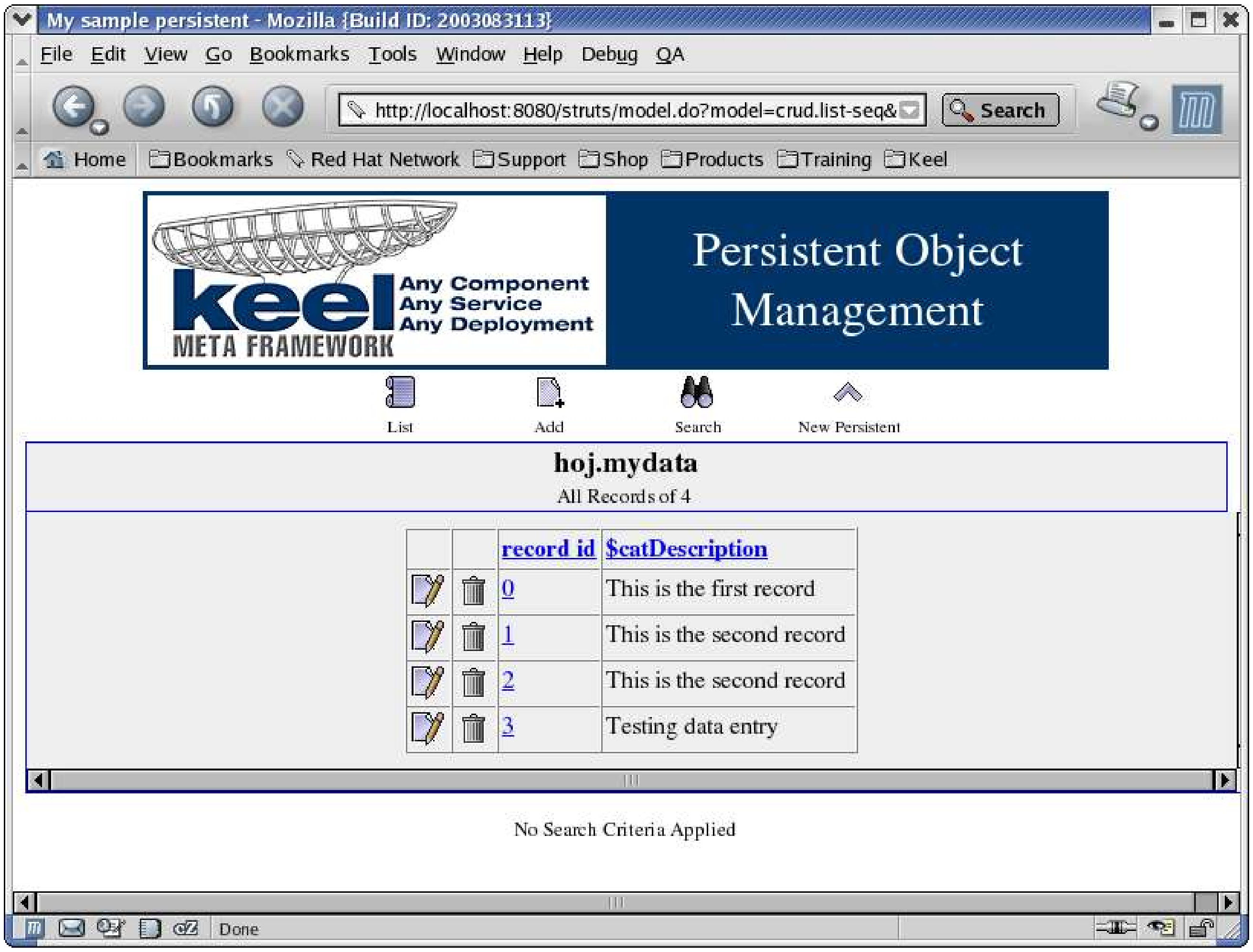
Prompt For Data

Figure 11- Output from Process Model

Pressing the "Prompt For Data" button should take you back to the prompt page again.

Verifying the data

To verify that the data was indeed added, go back to the Keel navigation menu and access Data->Select Schema. From the resulting screen, select "hoj" in the "Choose Schema" dropdown, and press "Select Schema". Click on "hoj.mydata" and you should see the data you entered. This was your first introduction to the CRUD application (app-crud).


My sample persistent - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=crud.list-seq&
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
keel META FRAMEWORK
Any Component
Any Service
Any Deployment
Persistent Object Management
List
Add
Search
New Persistent
No Search Criteria Applied
Done

hoj.mydata
All Records of 4

		record id	$catDescription
		0	This is the first record
		1	This is the second record
		2	This is the second record
		3	Testing data entry

Figure 12- Data Shown by CRUD

Adding a Struts User Interface

So far we haven't worried about putting up a pretty face, instead concentrating solely on the "business" end of things and building up the nuts and bolts of the server-side application. It's now time to see how to add in a Struts based front-end to our application. We will now add JSP front-ends to our prompt/process models we just got through exercising.

Where to go next?

Still on the abstract side of things, each model can have meta-data/configuration which provides a hint to Keel on "where to go" after the model has run. The decision is based on some attributes in the meta-data. Attributes called "forward" and "stylesheet" are used for this function. The Keel "forward" attribute (on the Keel server side) is provided to the Struts adaptor for Keel (on the Keel client side), which then uses it to find the Struts "forward" of the same name.

In the meta-data for the PromptMydata model, add the following definition:

```
 * @model.attribute
 *   name="forward"
 *   value="hoj.prompt-mydata"
```

In the meta-data for the ProcessMydata model, add the following definition:

```
 * @model.attribute
 *   name="forward"
 *   value="hoj.process-mydata"
```

Mapping to JSPs - Struts Forwards

The struts forward is defined as snippets of XML configuration, in KEEL/app-hoj/conf/client/struts/struts-config.xml. It should contain the forward definitions for each one of your models.

```
     <!-- Forwards for the HOJ models, supplied by the app-hoj project-->
     <forward   name="hoj.prompt-mydata"              path="/hoj/prompt.jsp"/>
     <forward   name="hoj.process-mydata"              path="/hoj/process.jsp"/>
```

Adding the JSPs

The JSPs are located in KEEL/app-hoj/src/jsp/struts/hoj. The Keel build process copies the app-hoj/src/jsp/struts directory to the context root directory of the webapp being created, such that the "hoj" directory will show as a child of the context-root directory.

The JSP source is included below. However, the content is straightforward. Each Input, Output and Command object added by the model is available as a bean to the JSP. Standard Struts bean and html tags are used to render the fields. The keel:command tag is a custom tag that makes it convenient to display Command objects as buttons; but standard Struts bean properties could be used to extract the parameters as well.

Here is the content of the prompt JSP:

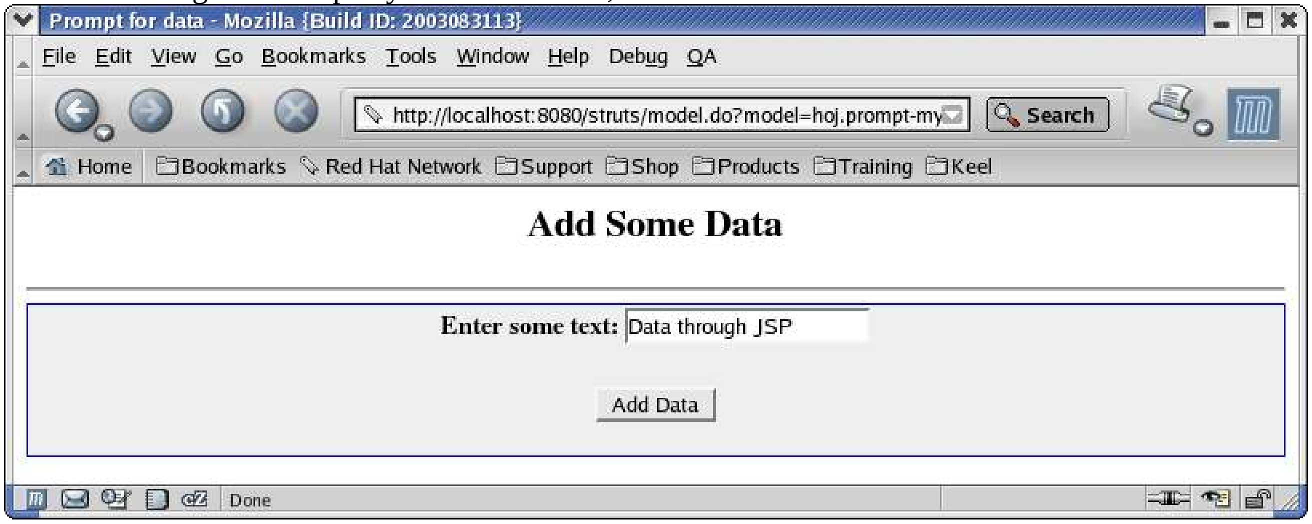
```
<%@ page language="java" %>

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-template.tld" prefix="template" %>
<%@ taglib uri="/WEB-INF/keel.tld" prefix="keel" %>

<html:html locale="true">
<head>
    <link rel="stylesheet" type="text/css" href="<%=request.getContextPath()%
>/hoj/hoj.css">
    <meta http-equiv="content-type" content="text/html;charset=iso-8859-1">
    <meta http-equiv="Content-Style-Type" content="text/css">
    <title>Prompt for data</title>
</head>

<html:base />

<body class="default">
    <p class="pageheader">Add Some Data</p>
    <font color="red">
    <keel:errors />
    </font>
    <hr>
    <html:form action="model" method="post">
        <div align="center">
            <table cellspacing="0" cellpadding="2" border="0">
                <tr>
                    <td class="reqlabel">
                        <bean:write name="mystring" property="label"/>:
                    </td>
                    <td>
                        <html:text name="default" property="mystring"/>

                    </td>
                </tr>
               <tr>
                   <td colspan="2">&nbsp;</td>
               </tr>
                <tr>
                    <td colspan="2">
                        <center>
                            <logic:present name="add">
                            <keel:command name="add"/>
                            </logic:present>
                        </center><br>
                    </td>
                </tr>
            </table>
        </div>
    </html:form>
</body>
</html:html>
```

The process JSP looks like:

```
<%@ page language="java" %>

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<%@ taglib uri="/WEB-INF/struts-template.tld" prefix="template" %>
<%@ taglib uri="/WEB-INF/keel.tld" prefix="keel" %>

<html:html locale="true">
<head>
    <link rel="stylesheet" type="text/css" href="<%=request.getContextPath()%
>/hoj/hoj.css">
```

```
    <meta http-equiv="content-type" content="text/html;charset=iso-8859-1">
    <meta http-equiv="Content-Style-Type" content="text/css">
    <title>Confirmation</title>
</head>

<html:base />

<body class="default">
    <p class="pageheader">Data Has Been Processed</p>
    <font color="red">
    <keel:errors />
    </font>
    <hr>
    <html:form action="model" method="post">
        <div align="center">
            <table cellspacing="0" cellpadding="2" border="0">
                <logic:present name="added">
                <tr>
                    <td>
                        <bean:write name="added"/>
                    </td>
                </tr>
                </logic:present>
               <tr>
                   <td>&nbsp;</td>
               </tr>
                <tr>
                    <td>
                        <center>
                            <logic:present name="prompt">
                            <keel:command name="prompt"/>
                            </logic:present>
                        </center><br>
                    </td>
                </tr>
            </table>
        </div>
    </html:form>
</body>
</html:html>
```

The result of running the PromptMydata model now, is as follows:


Prompt for data - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do?model=hoj.prompt-my
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
Done

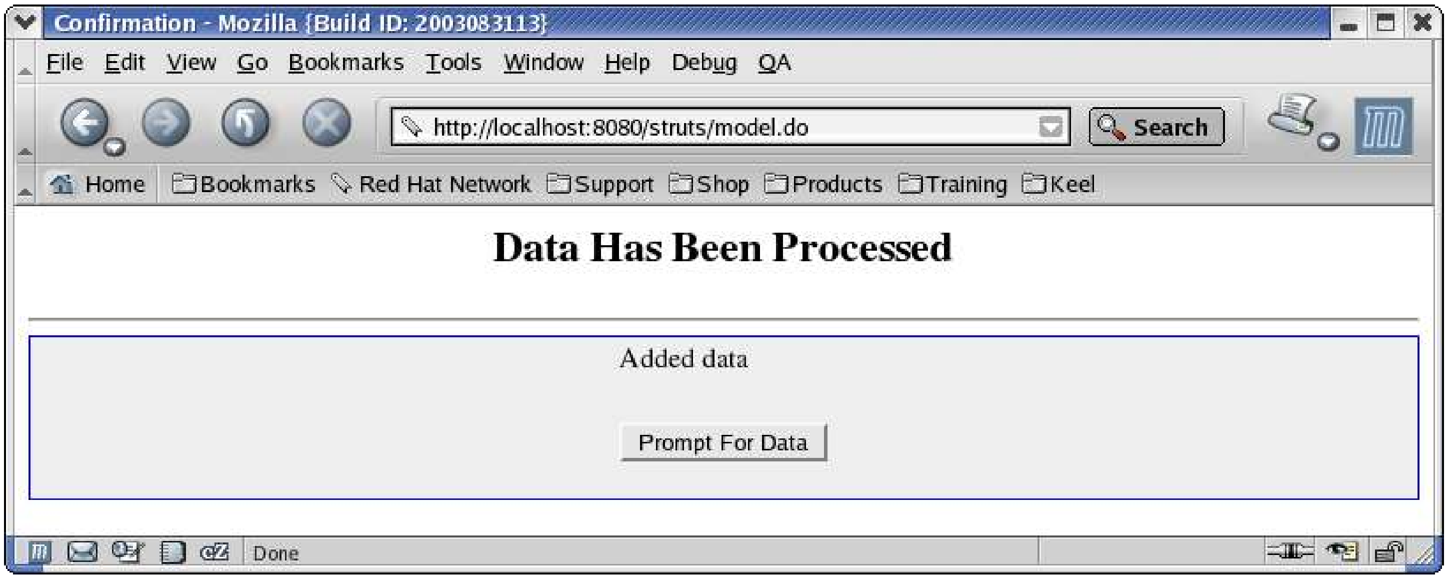
Add Some Data

Enter some text: Data through JSP

Add Data

Figure 13- JSP Output from Prompt Model

The result of running the ProcessMydata model now, is as follows:


Confirmation - Mozilla {Build ID: 2003083113}
File Edit View Go Bookmarks Tools Window Help Debug QA
http://localhost:8080/struts/model.do
Search
Home Bookmarks Red Hat Network Support Shop Products Training Keel
Prompt For Data
Done

Data Has Been Processed

Added data

Figure 14- JSP Output from Process Model

More To Come...

Congratulations for coming this far. This tutorial has shown the process of using Java and Ant on the system, as well as downloading and installing Keel Meta-framework and associated tools. The tutorial has shown how to build a sample application's models, connect it to a view, and then to enhance the model with security and persistence. However, this tutorial has barely scratched the surface of the capabilities of Keel. In fact, almost none of the optional available services in Keel were mentioned. As well, left alone were Keel capabilities such as Cocoon and Velocity views, or distribution of the client and server pieces using JMS, deployment in containers such as JBoss and many other features. The intent of this tutorial was to introduce the reader to working with Keel, in the near future upcoming articles will introduce more advanced features.

References

- Java Frameworks and Components - Accelerate Your Web Application Development, Michael Nash, Cambridge University Press, ISBN 0-521-52059-2
- Keel Meta-framework - http://www.keelframework.org
- Keel - The Next Generation Meta Framework - http://www.theserverside.com/resources/article.jsp?l=Keel
- Keel Documentation Wiki - http://66.105.113.115/vqwiki-2.3.5/jsp/Wiki?StartingPoints
- Keel Mailing Lists - http://www.keelframework.org/mail-list.shtml
- Apache Avalon Project - http://avalon.apache.org/
- Apache Tomcat - http://jakarta.apache.org/tomcat/index.html
- Apache Struts - http://jakarta.apache.org/struts/index.html
- Apache Ant - http://ant.apache.org/