MaduraWorkflowUI

User Guide

Document Revision: Revision: 1

Document Revision: Revision: 1

Table of Contents

1.Change Log..... 5

2.References..... 6

3.What is this?..... 7
3.1.What do you mean: Workflow?..... 7
3.2.So what does this do?..... 7
3.3.Running the application..... 7

4.The UI..... 17
4.1.Vaadin and Madura Vaadin Support..... 17
4.2.Security..... 17
4.3.Spring Framework..... 17
4.4.Scheduler..... 18

5.Workflow Bundles..... 19
5.1.Bundle Configuration..... 19
5.2.Bundled Forms..... 21
5.3.Bundle contents..... 23

6.Database..... 25
6.1.Workflow Database..... 25
6.2.Bundled Databases..... 27

7.Locking..... 29

8.JMX..... 30

9.Configuring for Production..... 31

10.Building Your Own..... 32

A.License..... 33

B.Release Notes..... 34

1. Change Log

2. References

[1] Weblogic JNDI
[2] Spring Security
[3] Atomikos
[4] Tomcat
[5] H2
[6] Vaadin
[7] Madura Utils
[8] Madura Objects
[9] Madura Vaadin Support
[10] Madura Rules
[11] Madura Bundles
[12] Madura Workflow
[13] Madura Workflow UI
[14] Apache Licence 2.0
[15] Spring Framework

3. What is this?

3.1. What do you mean: Workflow?

If you need an application that, say, just has a user fill in a form and save the results to a database then you do not need workflow. However, if you need an application that has a user fill in one of several forms, each of which has dynamic validation, then passes the result to another user who operates it using another form (also with dynamic validation) and then perhaps send the result to an external service that returns information that is then integrated into the result, and all this might take several days or weeks to complete, then you probably do need workflow. If, in addition, there are conditions and timeouts that route the results to a supervisor for review, and there are hundreds if not thousands of these things going through the system at once then you almost certainly do need workflow.

3.2. So what does this do?

This is a sample application showing what can be built onto Madura Workflow[12].

Madura Workflow is a workflow engine rather than an application, consequently it leaves a number of implementation decisions open. Those decisions are addressed here. The sample application is somewhere between a demo and production code. With a few tweaks it could be used in production and part of this document explores those tweaks and further ways to extend the application.

Over and above the workflow engine this application provides the following:

- A full UI implemented in Vaadin which allows users to launch and manage process instances.
- Security, based on Spring Security[2] which provides a login and associated permissions. The permissions are used to restrict users' access to certain queues and process instances.
- A way to deploy new process definitions on the fly using Madura Bundles[11]. The bundles also contain all associated resources, such as forms, messages, object definitions and custom code.
- A JPA database implementation, including a two phase commit transaction supporter (Atomikos[3]). The configuration for this is worth reviewing.
- Integration with Madura Rules[10]. This means the objects defined in the process definitions can have rules attached to them. The forms and other interactions with these objects will automatically invoke the rules. This greatly simplifies the code that you would otherwise have to write for the UI and message handling.
- Scheduling based on Spring's[15] `task:scheduler` namespace which by default uses `ScheduledThreadPoolExecutor`.
- JMX integration which allows you to monitor some lower level facilities.
- Deployment in an application server, in this case Tomcat[4], but no Tomcat-specific services are used so it ought to run on any JEE compliant server.
- A locking protocol which manages locks across the system.

Each of these is discussed in more detail below, as well as some ways to extend the application or invoke Madura Workflow in other ways.

3.3. Running the application

The README.md file in [12] gives information about deploying this, including some minor but necessary configuration.

You will need an application server. We tested this with Tomcat V7, but any application server you are comfortable with should be just fine. It also runs on VMWare's cloud server, and probably most others. Make sure you are running Java 7 or later.

When you start your application server the console log will show some warnings:

```
WARN  c.a.i.c.UserTransactionServiceImp - Slf4jLogger.java:12 No
 properties path set - looking for transactions.properties in classpath...
WARN  c.a.i.c.UserTransactionServiceImp - Slf4jLogger.java:12 Using init
 file: /home/roger/madura4/.metadata/.plugins/org.eclipse.wst.server.core/
tmp0/wtpwebapps/MaduraWorkflowUI/WEB-INF/classes/transactions.properties
WARN  c.a.jdbc.AbstractDataSourceBean - Slf4jLogger.java:12
 AtomikosDataSoureBean 'pu__workflow': poolSize equals default - this may
 cause performance problems!
WARN  org.hibernate.ejb.Ejb3Configuration - Ejb3Configuration.java:1132
 HHH000144: hibernate.connection.autocommit = false breaks the EJB3
 specification
WARN  c.a.jdbc.AbstractDataSourceBean - Slf4jLogger.java:12
 AtomikosDataSoureBean 'Workflow1-0.0.2': poolSize equals default - this
 may cause performance problems!
WARN  org.hibernate.ejb.Ejb3Configuration - Ejb3Configuration.java:1132
 HHH000144: hibernate.connection.autocommit = false breaks the EJB3
 specification
INFO  n.c.s.m.bundle.BundleManagerImpl - BundleManagerImpl.java:224 Added
 bundle: workflow1-0.0.2
```

These are all normal and do not cause problems, though the poolsize should be reviewed before going into production.

Browse to `http://localhost:8080/madura-workflow-ui/` (your server name and port may vary if you are not running a default configured Tomcat on your local machine). You should see a login page.

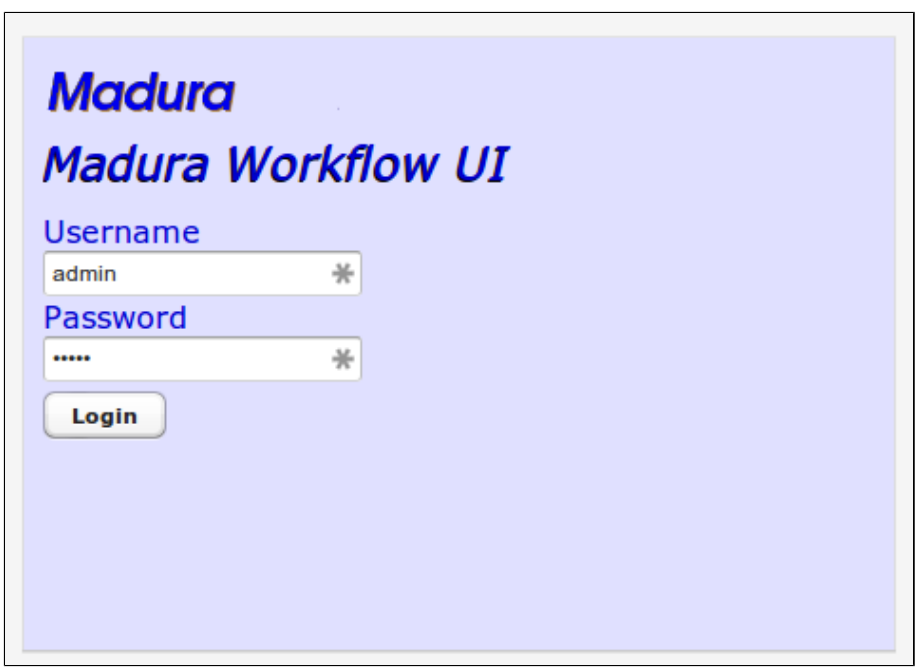

Figure (1) Login

The username/password is admin/admin.

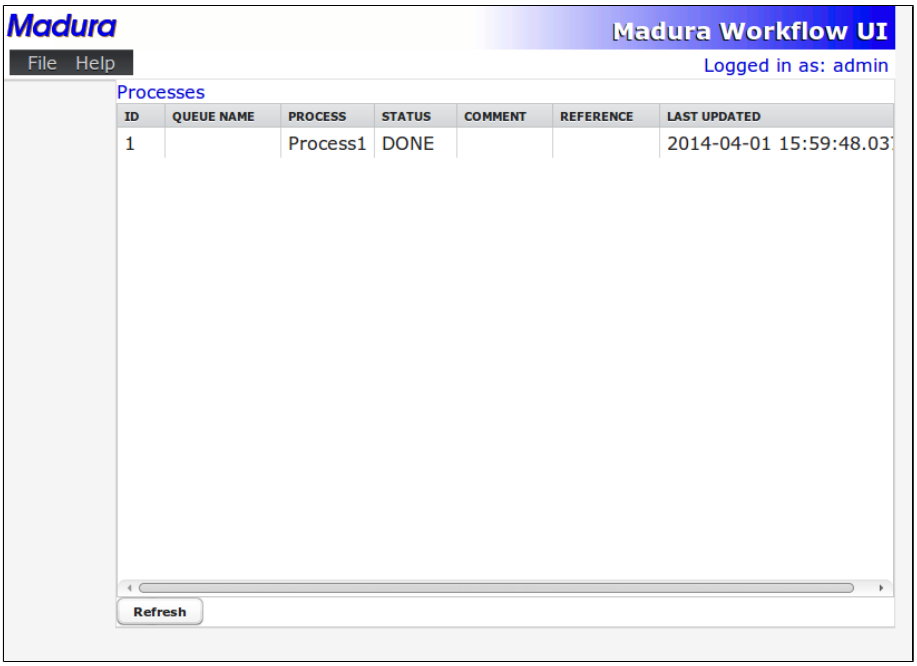

Figure (2) Home-1

This is the home screen, the one every user reaches just after login. It is mostly taken up by the Processes table which shows all the processes this user is allowed to see. Since we logged in as admin (who has ADMIN permission) we can see (but not necessarily edit) everything, and there is a process instance in the table with the status DONE, which means it ran to completion. Ordinary users only see process instances they are able to do something with so they do not see DONE processes. When you run your copy for the first time you will not see the DONE process.

The next step is to launch a new process instance. Click on File->Launch and you will see the Launch Wizard.

Figure (3) LaunchWizard

The Launch Wizard shows a list of all the process definitions in the system that this user is allowed to launch. Because this user has ADMIN permission that means all of them. Other users might see a shorter list or none at all. In fact this is not the full list of process definitions anyway. There might be multiple versions of each of these processes and only the latest versions are presented because we cannot launch old versions. Also some process definitions are not directly launchable, they can only be launched by another process so they are not included either.

Just clicking on one of the process definitions will show its launch form. Click on the first one, the Demo process.

Before going any further we should look at what this process does.

```
process: Order "Demo" "This is the demo process" launchForm=LaunchDemo
 queue="Q1"{
 try {
```

```
  message=orderMessageSender;
 }
 catch (abort) {
  compute=temperatureCompute;
 }
 form=DisplayFahrenheit queue="Q1";
}
```

This is a trivial example of a process. It starts with a launch form which we will see next. Once underway it tries to send a message to an external web service. The web service converts Fahrenheit to Celsuis, total overkill for this kind of software, but it does give us an understandable sequence to work through. If the web service fails the compute task is performed and this just runs some Java code to convert the temperature.

Finally the process shows a form that displays the converted temperature.

The two forms could be custom written in Java using Vaadin, but this takes the lazy approach and just generates the Vaadin form from the underlying object. The underlying object is, of course, a Madura Object and includes rich metadata, so this is more functional that you might suppose.

```
<bean id="VaadinLaunchDemo"
 class="nz.co.senanque.workflow.VaadinLaunchForm" scope="prototype">
 <property name="referenceName" value="orderName"/>
 <property name="fieldList">
  <list>
   <value>orderName</value>
      <value>fahrenheit</value>
     </list>
    </property>
</bean>
<bean id="VaadinDisplayFahrenheit"
 class="nz.co.senanque.workflow.VaadinLaunchForm" scope="prototype">
    <property name="fieldList">
     <list>
      <value>celsius</value>
     </list>
    </property>
</bean>
```

Using the `fieldList` property we show only the fields we are interested in.

The message definition looks like this:

```
<bean id="orderMessageSender"
 class="nz.co.senanque.messaging.MessageSenderImpl">
    <property name="channel" ref="orderChannel" />
    <property name="replyChannel" ref="orderReplyChannel" />
</bean>
```

...which looks too simple to be true, but it refers to channels in the Spring Integration configuration which is out of scope for this document. However it is worth noting that the workflow process definition is only loosely coupled to SI. The workflow definition is only vaguely aware of how the messages are handled, it just knows they get sent somehow and a response of some kind comes back, and then it can go on.

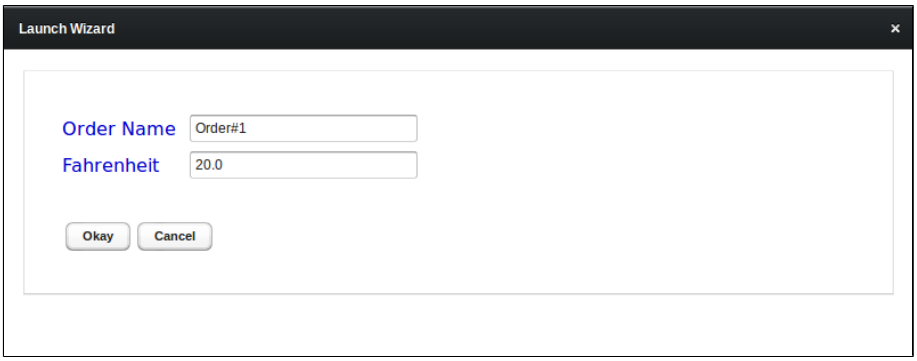

Figure (4) LaunchForm

By the time you see this form the process instance has not yet been launched. If you hit the Cancel button it never will be and there is nothing to clean up. Enter 'Order#1' into the Order Name field, accept the default value for Fahrenheit, and click Okay.

The process instance is launched and its id is displayed, you also have an opportunity to attach documents to the process instance at this point.

Figure (5) LaunchWizard2

And the Okay button will return you to the home page.

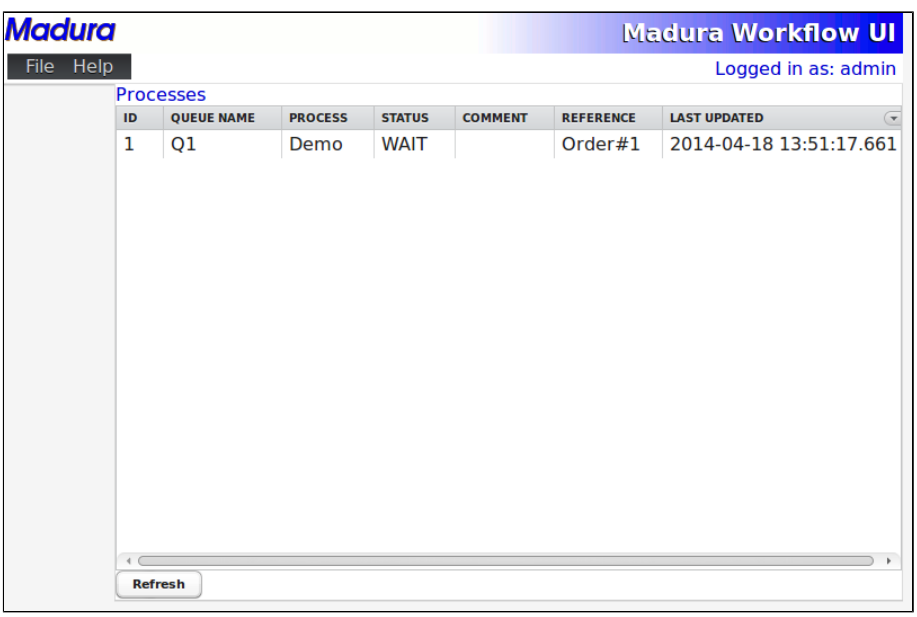

Figure (6) Home-2

The home page has an entry, which is the process instance you just launched. Yours might not have a queue name yet, so give it a few seconds and click refresh until it does. Notice that the Reference column contains 'Order#1' which is what you typed on the launch form. You can also see that it is in queue Q1 (when it appears) and that it is running process definition Demo, the definition you launched.

The process instance we launched has a queue name and that means it is waiting for a human to do something. Click on that process instance.

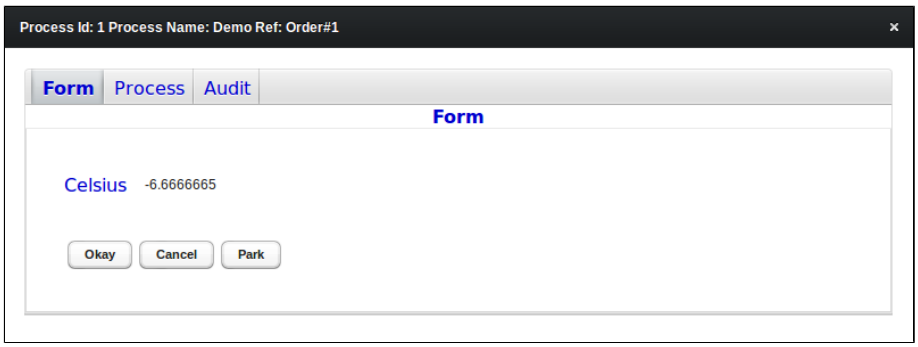

Figure (7) Process1-Form

This is the form associated with this stage of the process. It is displaying information about the object associated with this process instance, in this case an Order object. It is showing the converted temperature in Celsius. You can click Okay for it to go on to the next step, but before you do click on the Process tab.

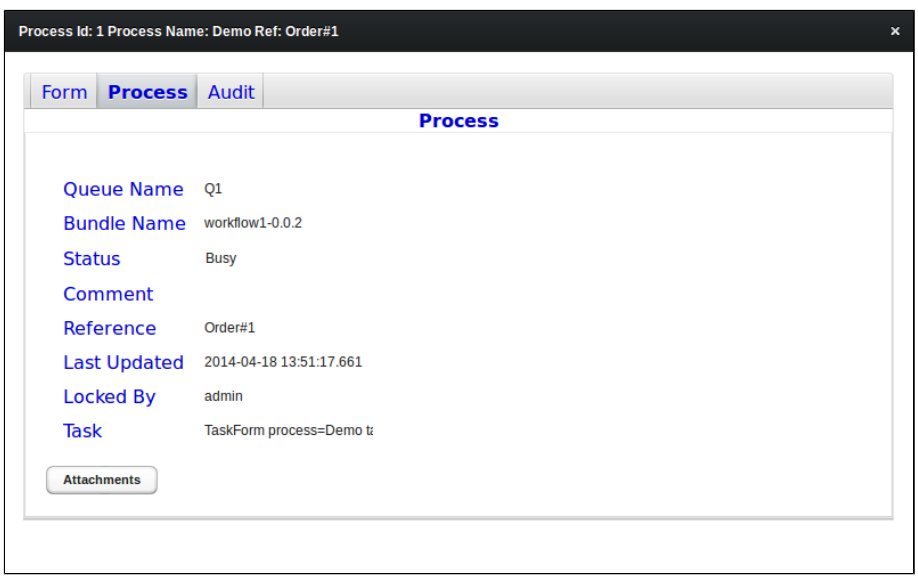

Figure (8) Process1-Process

The Process tab shows the internal details of the process instance. You can see what bundle this process was loaded from, its current status (Busy because you have got it locked) what queue it is in and so on. You can view these details but unless you have TECHSUPPORT permission you cannot change them. TECHSUPPORT can change anything you see here. There is also an opportunity to view the attachments. Now press the Audit tab.

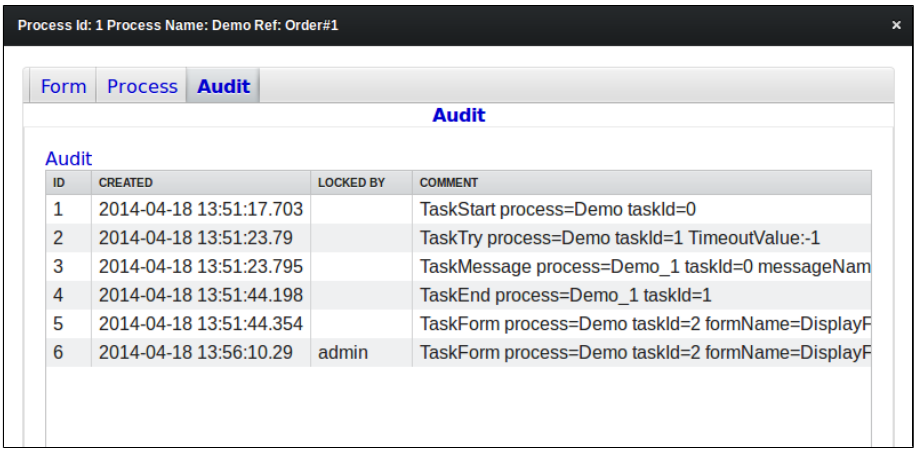

Figure (9) Process1-Audits

The Audit tab shows all the audit records generated for this process instance. You can see what tasks were run and when, and you can see who locked the process instance as well. Notice that entries 3 and 4 refer to a process instance of `Demo_1` rather than `Demo`. This is indicates there was an inner process generated from the main process' try/catch block. You can click on these entries for more detail.

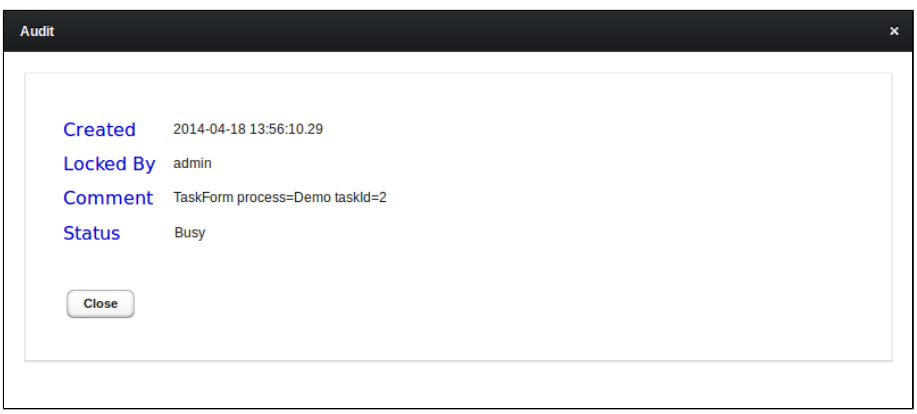

Figure (10) Process1-Audits Detail

Unlike the process instance details no one can edit an audit record, not even someone with TECHSUPPORT permission, although anyone with enough access to the database could modify this information, of course.

Now go back to the Form tab and press Okay. You might need to wait a few seconds and then press refresh to see the process instance finished.

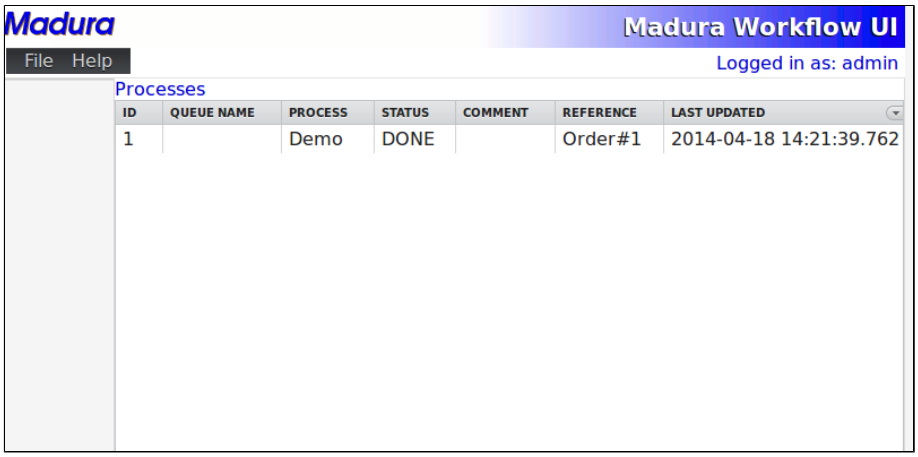

Figure (11) Home-3

As a demo this perhaps looks a little trivial. But note what happened carefully. A process instance was launched with a launch form which accepted some initial data. That process was then managed through multiple stages, including sending a message, operating a form, and running a custom compute bean.

What we did not see yet is handling of errors. Let's go around again, but this time turn off your internet connection.

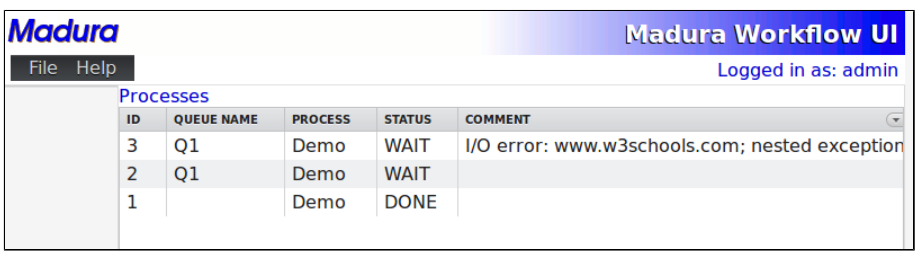

Figure (12) Home-4

The new process is id 3 and, if you give it time to try the message and then refresh you will see this. Notice the error message, that is because it could not get to the web service. So what happened then? Well, take a look at the process definition again:

```
process: Order "Demo" "This is the demo process" launchForm=LaunchDemo
 queue="Q1"{
 try {
  message=orderMessageSender;
 }
 catch (abort) {
  compute=temperatureCompute;
 }
 form=DisplayFahrenheit queue="Q1";
}
```

If there is an error in the message the catch block will be executed and the compute task will figure out the temperature. It will still land on the form, and that is where it is now. If you click on the entry you will see the form.

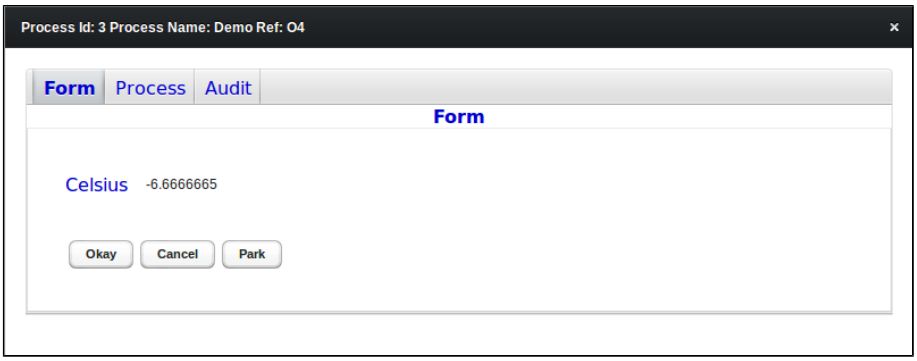

Figure (13) Process2-Form

Click on the process tab.

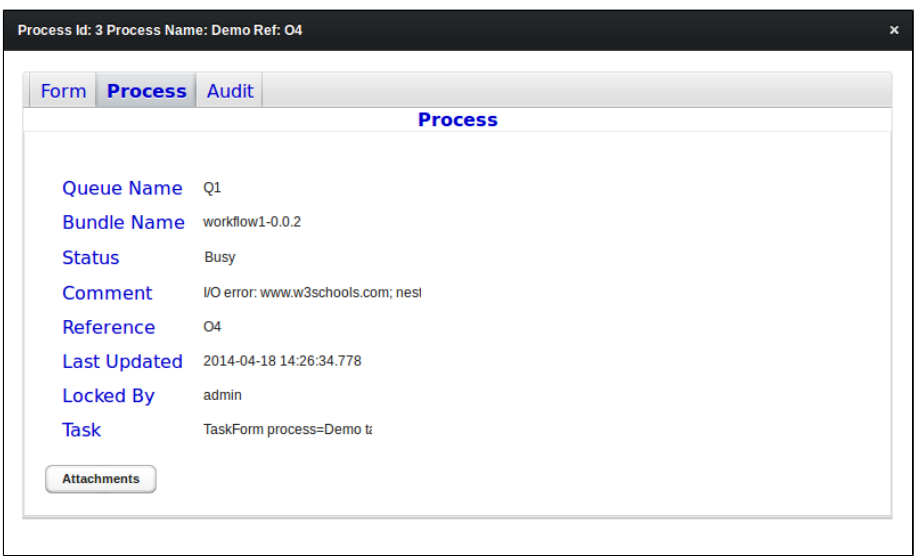

Figure (14) Process2-Process

Now you can see the details of the process, including the error. On the Audit tab you can see the trace of how the error was handled.

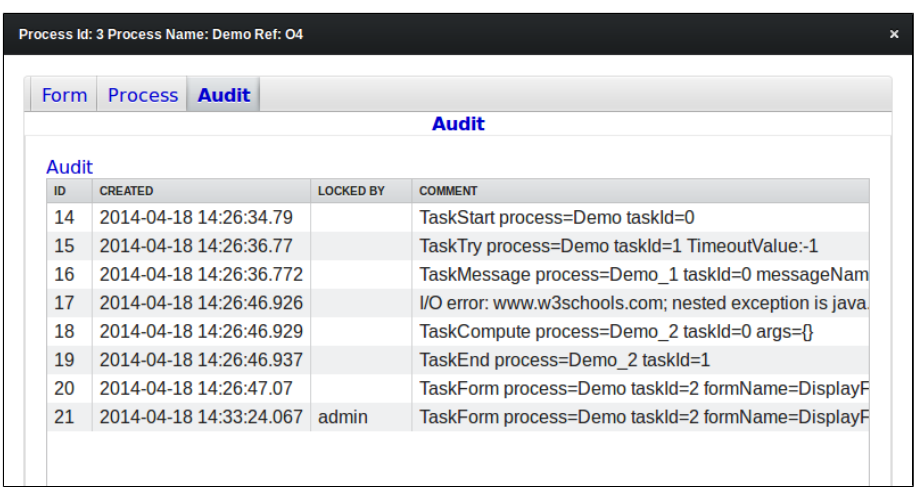

Figure (15) Process2-Audit

This shows the error from the message and the fact that the compute task was executed. Naturally the form on the first tab looks much the same with a computed value for the temperature.

The subject of attachments came up twice in that demo, once when the process was launched and once again when operating a form. Attachments are documents of any kind attached to the process instance. There are limitations on the size of the document you can attach which vary depending on the database product chosen. But any type of file can be attached. Also, attachments are used only for user reference. They do not directly influence the state of the process, though a user might make a decision based in information obtained by reading an attachment.

When the attachment button is pressed you see a window like this:

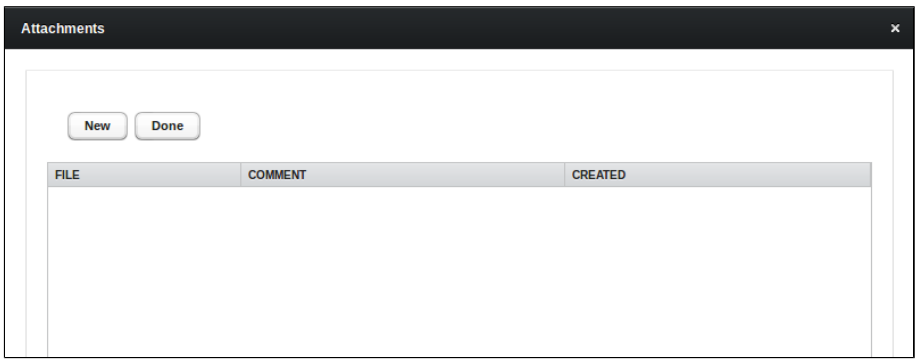

Figure (16) Attachments-1

This is the initial view of the attachments on a process instance, before any have been added. Now press the New button.

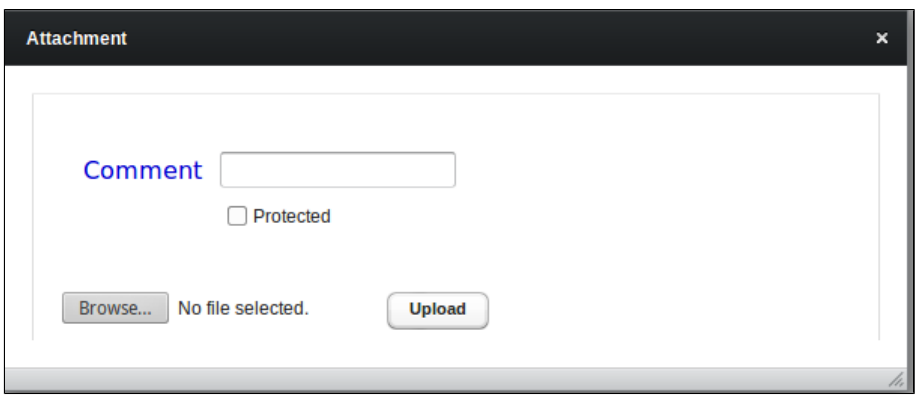

Figure (17) Attachments-2

It wants a new attachment now. There is space for a comment, a protected flag and the usual buttons to select and upload the file. The protected flag means only ADMIN and TECHSUPPORT users can see this attachment. Once an attachment is uploaded it goes back to the list of attachments.

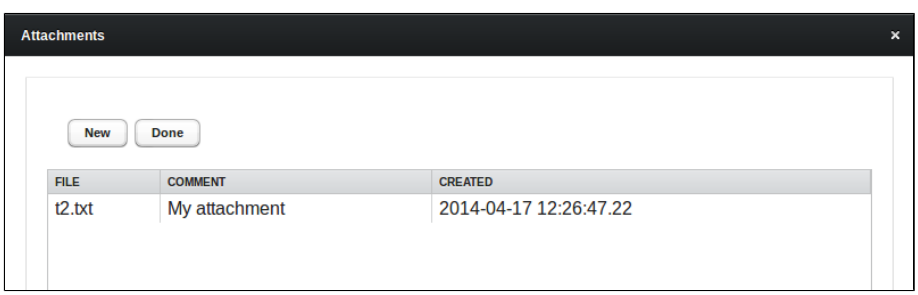

Figure (18) Attachments-3

Now the attachment just added is visible in the table. To examine it you just click on it and the browser will attempt to download it or open it, depending on your browser setting. There is no way to remove or modify attachments, that allows them to be used as a kind of audit trail.

4. The UI

4.1. Vaadin and Madura Vaadin Support

There seem to be endless ways of implementing a UI in Java so eventually you just have to pick one you like. In this application the choice is Vaadin[6] mostly because of its Swing-like API and the fact that as far as the programmer is concerned it is Java all the way down, you don't have to cope with Javascript or JSP etc. It also has a rich set of controls and all the styling is controlled by CSS. The CSS control and the 'Java all the way down' are not contradictory. Programmers should not need to worry very much about styling, that can be turned over to a *designer* and they often know CSS better than a Java programmer. Typically programmers can do a rough cut of the application styling and get the functionality in place, then get the designer to got to work on the CSS.

It is no coincidence that Madura Vaadin Support[9] handles the integration between Vaadin and Madura Objects[8]. That provides dynamic validation on the forms, I18n, and a good way to use Vaadin with Spring, including Spring Security.

So the choice here is Vaadin with a little help from Madura Vaadin Support.

4.2. Security

The security implemented in the application uses Spring Security and the configuration for that is defined in the file `security-context.xml`.

It is kept simple here: the users, passwords and permissions are hard coded in the xml file. In production you would use more elaborate Spring Security options such as fetching user profiles from a database etc. But this is sufficient to implement the login and permissions the application actually needs to run. There is further integration in Madura Vaadin Support because this provides the login form which then verifies the user against Spring Security and fetches the permissions. The permissions for the current user are then held in the PermissionManager, another class supplied by Madura Vaadin Support.

Madura Objects provides a way to attach permissions to object properties and Madura Vaadin Support honours those when the forms are displayed. That means you can say the XYZ permission only has read-only access to this property, and when it is displayed on a form it will be read-only etc. Since all forms in the application use Madura Objects in this way then any/every property may have a permission attached.

Queues also have permissions attached to them. That means that user have limited access to view or operate that queue, depending on their permission. If the current user has only read access to the queue then items in the queue cannot be opened. If they have no read access to the queue then no items from that queue will be displayed.

In an enterprise environment, and typically workflow would run in an enterprise environment, there is often an external login procedure which adds a security token to the request. Madura Vaadin Support's login form supports this, allowing a pre-logged-in user to bypass the login form.

There is also one 'super permission' called TECHSUPPORT. Users with this permission can always modify any process instance in any queue, including ones that are being worked on by another user. Naturally that capability ought to be used sparingly. The ADMIN permission allows users to view, but not necessarily change, anything. The specific names associated with these permissions are defined in the file `FixedPermissions.java`.

4.3. Spring Framework

When using Spring with Vaadin it is not normally possible to use XML to wire the display components because Spring makes most sense when wired beans are singletons and Vaadin's display objects need to be session objects. Madura Vaadin Support simplifies this by using two basic xml files, in this case `applicationcontext.xml` and `WorkflowUI.xml`. The first of these is just a normal Spring context file and most of the beans defined there are singletons as you expect with Spring. The second looks like a lot of singelton beans but they are, in fact, all session beans. This entire context is created when a new session starts and remains associated with the session until it ends. This

makes it easy to define Vaadin display objects as beans and it nicely separates the singletons from the session beans.

4.4. Scheduler

The scheduler is configured using Spring in `applicationContext.xml` like this:

```
<!-- The executor is responsible for scanning for active processes etc -->
<bean id="executor" class="nz.co.senanque.workflow.ExecutorImpl" />

<task:scheduler id="myScheduler" pool-size="10" />
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="bundleManager" method="scan"
        fixed-delay="10000" />
</task:scheduled-tasks>
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="executor" method="activeProcesses"
        fixed-delay="10000" />
</task:scheduled-tasks>
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="executor" method="deferredEvents"
        fixed-delay="10000" />
</task:scheduled-tasks>
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="executor" method="clearDeferredEvents"
        fixed-delay="60000" />
</task:scheduled-tasks>
```

The key to this is the `myScheduler` to which are added several tasks. The first is the task that invoked the bundle manager to scan the bundles directory for new bundles. The other three are workflow tasks and they invoke different methods on the `executor` bean.

The first of these scans for active processes, ie processes in tasks that can be executed off line. Such tasks include compute tasks and message tasks, but they do not include form tasks which need user input.

The other two tasks are for handling deferred events, which mostly means timeouts. When a deferred even comes due the relevant process instance has its status updated and then left for the active process handler to progress it. The `clearDeferredEvents` method is minor housekeeping which organises removing old deferred events. It does not need to run as often as the others.

If you need to deploy multiple copies of the application all using the same databases then depending on your workload profile you could consider having the workflow scheduled tasks run in only one of those copies because doing all the off-line processing from one dedicated machine is often a good idea. But multiple copies will run happily enough together.

But all copies of the application must scan the bundles directory because they must all be aware of the same bundles.

5. Workflow Bundles

By making use of Madura Bundles[11] the application supports deploying of new workflow definitions and their associated resources on the fly. A bundle is a specially packaged jar file and by copying a new jar file into the monitored directory a new bundle can be deployed or an existing bundle can be upgraded(1). In the case of new bundles the workflow definitions contained in the bundle will become visible on the list of launchable processes in the UI. For upgraded bundles the situation is more complex because there may be active process instances using the previous version (say, version 0.0.1) and now we have 0.0.2. Those active process instances will actually continue to use the workflow definition they started with, and only new launches will use the 0.0.2 version.

Bundles, as already noted, are just specially packaged jar files, which means they have some extra entries in the `MANIFEST.MF` file, including a reference to a Spring context file. So they are just ordinary Madura Bundles, and we will work through their configuration in detail below. First we need to look at the bundle configuration in the main application.

5.1. Bundle Configuration

```
<!-- <jee:jndi-lookup id="bundlesDir" jndi-name="java:/comp/env/
WorkflowUIBundlesDir" expected-type="java.lang.String" /> -->
<bean id="bundleManager"
 class="nz.co.senanque.madura.bundle.BundleManagerImpl">
<!--  <property name="directory" ref="bundlesDir"/> -->
 <property name="inheritableBeans">
  <map>
    <entry key="jpaVendorAdapter" value-ref="jpaVendorAdapter"/>
    <entry key="transactionManager" value-ref="transactionManager"/>
    <entry key="lockFactory" value-ref="lockFactory"/>
    <entry key="em-workflow" value-ref="em-workflow"/>
    <entry key="atomikosTransactionManager" value-
ref="atomikosTransactionManager"/>
    <entry key="atomikosUserTransaction" value-
ref="atomikosUserTransaction"/>
    <entry key="errorEndpoint" value-ref="errorEndpoint"/>
    <entry key="genericEndpoint" value-ref="genericEndpoint"/>
    <entry key="environment" value-ref="environment"/>
    <entry key="workflowDAO" value-ref="workflowDAO"/>
    <entry key="bundleManager" value-ref="bundleManager"/>
    <entry key="hints" value-ref="hints"/>
  </map>
 </property>
</bean>

<bean id="bundleListener"
 class="nz.co.senanque.workflowui.bundles.BundleListenerImpl">
 <property name="messageSource" ref="messageSource"/>
</bean>
```

This is from the `applicationcontext.xml` file. It defines a JNDI variable which holds the location of the directory the bundles are held in. This directory will be scanned every few (configurable) seconds for new files, and the inheritable beans map contains beans that are defined in this file that must be visible from the bundles. The bundles actually see all these beans as if they defined them themselves.

Finally the `bundleListener` bean is called by the bundle manager whenever a new bundle is added. The listener peers inside the new bundle for new process definitions and new queues and it places these into an internal map so that when the user opens the list of available processes they are

1) Since Madura Bundle 4.0.0 bundles may be deployed to a maven repository and, instead of copying the jar file to the monitored directory you copy a small text file describing the bundle instead.

already there for display. The alternative would be to scan all the bundles for this information every time it is asked for which would be inefficient.

Inside a bundle we always have a Spring context file and a bean defining the workflow manager like this:

```
<bean id="workflowManager"
 class="nz.co.senanque.workflow.WorkflowManagerImpl">
    <property name="schema" value="classpath:/OrderInstances.xsd" />
    <property name="processes" value="classpath:/OrderWorkflow.wrk" />
</bean>
```

This refers to the xsd file that defines the objects used by the workflow, and the wrk file that contains the workflow process definitions.

As well as this the bundle needs a JPA database, yes another one, see *4* for details as to why.

All the beans defined in the bundle context file are singletons unless scoped otherwise, and one or two are. Here are the rest of the essential beans:

```
<import resource="classpath:/database-nmcinstances-context.xml"/>

<context:annotation-config />
<context:component-scan base-package="nz.co.senanque.workflow.nmcrules" />
<bean id="propertyConfigurer"
 class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" /
>

<bean id="bundleName"
 class="nz.co.senanque.madura.bundle.StringWrapperImpl">
 <constructor-arg value="${bundle.name}"/>
</bean>
<bean id="permissionManager"
 class="nz.co.senanque.vaadinsupport.permissionmanager.PermissionManagerImpl"
 scope="session"/>
<bean id="maduraSessionManager"
 class="nz.co.senanque.vaadinsupport.application.MaduraSessionManager"
 scope="bundle">
    <property name="formFieldFactory" ref="fieldFactory"/>
</bean>

<bean id="fieldFactory" class="nz.co.senanque.vaadinsupport.FieldFactory"
 scope="bundle"/>

<bean id="workflowManager"
 class="nz.co.senanque.workflow.WorkflowManagerImpl">
 <property name="schema" value="classpath:/NMCInstances.xsd" />
 <property name="processes" value="classpath:/NMCWorkflow.wrk" />
 <property name="validationEngine" ref="validationEngine" />
</bean>

<bean id="contextDAO" class="nz.co.senanque.workflow.nmc.ContextJPA"/>
```

This file imports a database configuration which will be discussed in detail in *4*. There is some necessary housekeeping beans and then three session scoped beans: `maduraSessionManager`, `fieldFactory` and `hints`. These are all used by Madura Vaadin Support. The two DAO beans are used to access the two different databases, then there are two forms both with prototype scope.

5.2. Bundled Forms

If the workflow definition includes `form` tasks or launch forms then those forms must be in the bundle, they cannot be delivered by the main application because it does not know anything about the data in the workflow definition. That means the forms are configured in the bundle like this:

```
<bean id="VaadinLaunchForm"
 class="nz.co.senanque.workflow.VaadinLaunchForm" scope="prototype">
 <property name="referenceName" value="orderName"/>
</bean>
<bean id="VaadinFirstForm" class="nz.co.senanque.workflow.VaadinFirstForm"
 scope="prototype">
 <property name="fieldList">
  <list>
   <value>orderName</value>
   <value>celsius</value>
  </list>
 </property>
</bean>
```

The two forms are referred to by the workflow definition. It has an entry like this:

```
...
process: Order "Process2" "This is the second process"
 launchForm=LaunchForm {
 try {
  message=orderMessageSender;
  form=FirstForm queue="Q1";
...
```

There they are! LaunchForm and FirstForm. The name of the form in the definition is just the name of the bean in the bundle... except the beans have 'Vaadin' on the front of it so them is a little more complicated. The forms are actually delivered using the `formFactory` bean and that is auto-injected with the `environment` bean which contains the string 'Vaadin'. If it is injected with an environment `formFactory` prepends the string to the bean name. If there is no environment then it prepends nothing.

The form beans need to be scoped as prototype so that whenever they are requested by the `formFactory` a new copy is created.

Why would you ever need that extra complication? Well sometimes you might have multiple UI technologies going on at once. Some users, perhaps, are using Vaadin and others for whatever reason are using Swing and others are using Camel. Assuming you know which user is using which technology you can change the hard coded 'Vaadin' there for something like `${myenvironment}` and arrange for the correct value for that symbol to be defined for each user. Then you would define *six* beans to cover all the cases like this:

```
...
<bean id="VaadinLaunchForm"
 class="nz.co.senanque.workflow.VaadinLaunchForm" scope="prototype">
 <property name="referenceName" value="orderName"/>
</bean>
<bean id="VaadinFirstForm" class="nz.co.senanque.workflow.VaadinFirstForm"
 scope="prototype">
 <property name="fieldList">
  <list>
   <value>orderName</value>
   <value>celsius</value>
```

```
  </list>
 </property>
</bean>
<bean id="SwingLaunchForm" class="nz.co.senanque.workflow.SwingLaunchForm"
 scope="prototype">
 <property name="referenceName" value="orderName"/>
</bean>
<bean id="SwingFirstForm" class="nz.co.senanque.workflow.SwingFirstForm"
 scope="prototype">
 <property name="fieldList">
  <list>
   <value>orderName</value>
   <value>celsius</value>
  </list>
 </property>
</bean>
<bean id="CamelLaunchForm" class="nz.co.senanque.workflow.CamelLaunchForm"
 scope="prototype">
 <property name="referenceName" value="orderName"/>
</bean>
<bean id="CamelFirstForm" class="nz.co.senanque.workflow.CamelFirstForm"
 scope="prototype">
 <property name="fieldList">
  <list>
   <value>orderName</value>
   <value>celsius</value>
  </list>
 </property>
</bean>
<bean id="formFactory" class="nz.co.senanque.forms.FormFactoryImpl"/>
<bean name="environment" class="nz.co.senanque.forms.FormEnvironment">
 <property name="name" value="${myenvironment}"/>
</bean>
```

With that in place the right form will be delivered to the right user. However, so far only Vaadin forms have been fully implemented so if you do want Swing or Camel you will have more work to do than for Vaadin. It is probably fairly easy to build a simple form in another technology (depending on the technology, of course), more complex is integrating that form with Madura Objects. Without Madura Objects the forms would need to be more complicated and harder to maintain.

In practice the two Vaadin forms in the sample are very simple extensions of a base class `GenericVaadinForm` that is packaged in the main application. The base class is generic enough to simply generate a form based on the object it is bound to using all the properties it finds in it, unless a field list is supplied (as it is with the FirstForm) in which case just those fields are displayed. It also presents three buttons: OK, Cancel and Park. The first two are obvious enough but the last allows a user to save the current process instance without actually releasing it, useful if they have not completed it but want to go home for the night etc.

The generic form also has a `referenceName` property. When the form is saved the property named for this (`orderName`) in this case, is copied to the process instance reference, which means it is visible in the table of process instances.

`GenericVaadinForm` is smart enough to generate date fields from date properties, checkboxes from booleans, drop downs from enums etc and because they are backed by Madura Objects you get automatic validation of the fields. For example numeric fields will insist the user enters a number. Labels are automatically generated and they are all supported by I18n. Even better you can use Madura Rules to add rules for cross field validation and to make the forms dynamic. For example you can specify rules that will limit the drop down lists on one field depending on values entered into other fields, you can also add rules that switch fields from enabled to disabled, visible and invisible etc.

While you can use rules to control if fields are readOnly or not, you sometimes want to make a whole form readOnly. You can do that with the readOnlyForm property like this:

```
<bean id="VaadinFirstForm" class="nz.co.senanque.workflow.VaadinFirstForm"
 scope="prototype">
 <property name="readOnlyForm" value="true"/>
 <property name="fieldList">
  <list>
   <value>orderName</value>
   <value>celsius</value>
  </list>
 </property>
</bean>
```

So you can get quite a long way with just using `GenericVaadinForm`. But in a production application you would eventually need to add something else. For example if you have an Order which needs OrderItems added you would probably extend `GenericVaadinForm` but you would add more code to it to create OrderItems, attach them to the Order and so on. There would likely be more buttons involved and at least one popup window. But do remember than the OrderItems can be monitored by rules as well. So, for example, to get the order total you would write a rule and the total would be updated as OrderItems are added, deleted or their details changed.

5.3. Bundle contents

There are two working bundles supplied with the application: Workflow1 and tbundle, the most interesting one is Workflow1. They are both maven projects under the `bundles` directory. The resulting jar files are copied to that directory as well so this is where `bundlesDir` ought to point to.

The Workflow1 bundle looks like this:

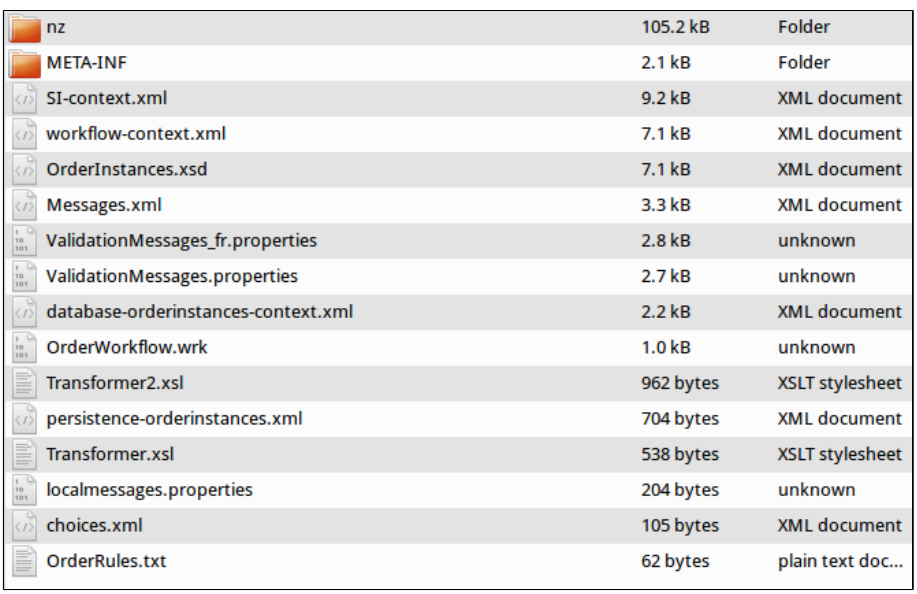

Figure (19) Workflow1 Bundle

There are two subdirectories at the top, the `nz` is the beginning of the `nz.co.senanque.workflow` structure which is detailed later. `META-INF`, as usual, contains the `MANIFEST.MF` file which in this case looks like this:

```
Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Build-Jdk: 1.7.0_21
Built-By: Roger Parkinson
Bundle-Activator: nz.co.senanque.madura.bundle.BundleRootImpl
Bundle-Context: workflow-context.xml
```

```
Bundle-Description: A sample bundle containing a workflow.
Bundle-Name: Workflow1
Bundle-Version: 0.0.2
```

This is normal for a Madura Bundle and the main thing to note is that the `Bundle-Context` specifies the `workflow-context.xml`, which is the Spring context that is to be loaded for this bundle.

After the subdirectories there are a number of files in the top directory of the jar file.

`workflow-context.xml` ,`SI-context.xml` and `database-orderinstances-context.xml` are all Spring context files. `workflow-context.xml` is the main one that imports the other two. `database-orderinstances-context.xml` contains the database definitions and `SI-context.xml` contains Spring Integration configuration, which means this bundle uses SI to send messages to external services, typically web services.

`OrderWorkflow.wrk` and `OrderInstances.xsd` are the process definitions and the definitions of the objects they refer to. The xsd file has already been used to generate the annotated POJOs but it has a runtime function as well.

There are two xsl files which are used to generate and unpack web services messages. These are referred to by `SI-context.xml`, and there are several properties files used to provide I18n translation.

Finally the `OrderRules.txt` file contains the Madura Rules that monitor the objects defined in the xsd file. This file actually has no runtime function because the rules have been generated into Java classes and placed in `nz.co.senanque.workflow.orderinstances`. `choices.xml` and `Messages.xml` are also used by Madura Rules.

Which brings us to the contents of the `nz.co.senanque.workflow` structure.

`nz.co.senanque.workflow.orderinstances` contains the POJOs generated from the xsd file and `nz.co.senanque.workflow.orderrules` contains the generated rules. At the top of the structure, ie in `nz.co.senanque.workflow` are classes for the two Vaadin forms and the two custom compute classes referred to by the workflow definition.

6. Database

Madura Workflow requires a JPA database and a transaction handler that supports two phase commit. For this application the choice is an H2[5] memory resident database and the Atomikos[3] transaction handler. Memory resident databases are, of course, not a good choice for production but very good for a demo. The H2 database, while not normally a choice for enterprise databases, has the advantage that it requires no installation, which means it is that much less setup to do to run this application out of the box. The database configuration is defined in `database-context.xml` as well as a separate configuration for each workflow bundle.

It is important to note that there are two databases or, at least, two database connections. They are both JPA and they are normally, though not necessarily the same database product. But why exactly are there two databases?

There is one for the basic workflow, that holds the process instances and so on, but nothing about the data that is being manipulated by the workflow. For example a process definition might refer to an Order object and this has to be serialized to a database between workflow tasks. But the Order is part of the process definition, not the core workflow. Process definitions are free to operate on any objects they want to so those objects cannot be dictated by the core workflow. That means that the bundle containing the process definition must contain the object definitions as well, so there are Java POJOs annotated with JPA included in the bundle, effectively defining the database tables needed for the bundle.

6.1. Workflow Database

Now it is time to look at the workflow database configuration (as opposed to the one in the bundle). This is the `database-context.xml` file:

```
...
<bean id="JDBCPool"
 class="com.vaadin.data.util.sqlcontainer.connection.SimpleJDBCConnectionPool">
 <constructor-arg index="0" value="org.h2.jdbcx.JdbcDataSource"/>
 <constructor-arg index="1"
 value="jdbc:h2:mem:workflow;DB_CLOSE_ON_EXIT=FALSE;MVCC=true"/>
 <constructor-arg index="2" value=""/>
 <constructor-arg index="3" value=""/>
</bean>

<bean id="em-workflow"
 class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
 depends-on="springJtaPlatformAdapter">
 <property name="persistenceXmlLocation" value="classpath:META-INF/
persistence-workflow.xml" />
 <property name="persistenceUnitName" value="pu-workflow" />
 <property name="dataSource" ref="dataSourceWorkflow" />
 <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
 <property name="jpaDialect">
  <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect" />
 </property>
 <property name="jpaProperties">
  <map>
   <entry key="hibernate.transaction.jta.platform"
 value="nz.co.senanque.hibernate.SpringJtaPlatformAdapter" />
   <entry key="hibernate.dialect"
 value="org.hibernate.dialect.H2Dialect" />
   <entry key="hibernate.format_sql" value="true" />
   <entry key="hibernate.connection.autocommit" value="false" />
  </map>
 </property>
</bean>
```

```
<bean id="dataSourceWorkflow"
 class="com.atomikos.jdbc.AtomikosDataSourceBean" init-method="init"
 destroy-method="close">
    <property name="uniqueResourceName" value="pu__workflow" />
    <property name="xaDataSourceClassName"
 value="org.h2.jdbcx.JdbcDataSource" />
    <property name="xaProperties">
        <props>
        <prop
 key="url">jdbc:h2:mem:workflow;DB_CLOSE_ON_EXIT=FALSE;MVCC=true</prop>
        </props>
    </property>
    <property name="maxPoolSize" value="20"/>
</bean>

<bean id="jpaVendorAdapter"
 class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
 <property name="showSql" value="false" />
 <!-- ensures new db is auto created if needed -->
 <property name="generateDdl" value="true" />
 <property name="databasePlatform"
 value="org.hibernate.dialect.H2Dialect" />
</bean>

<bean id="springJtaPlatformAdapter"
 class="nz.co.senanque.hibernate.SpringJtaPlatformAdapter">
    <property name="jtaTransactionManager" ref="transactionManager" />
</bean>
<bean id="atomikosTransactionManager"
 class="com.atomikos.icatch.jta.UserTransactionManager"
 init-method="init" destroy-method="close">
 <property name="forceShutdown" value="false" />
</bean>
   <bean id="atomikosUserTransaction"
 class="com.atomikos.icatch.jta.UserTransactionImp">
       <property name="transactionTimeout" value="300" />
   </bean>
<bean id="transactionManager"
 class="org.springframework.transaction.jta.JtaTransactionManager">
 <property name="transactionManager" ref="atomikosTransactionManager" />
 <property name="userTransaction" ref="atomikosUserTransaction" />
 <property name="allowCustomIsolationLevels" value="true" />
</bean>

<bean id="persistenceAnnotation"
 class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" /
>
```

The `JDBCPool` bean is used to display the table of relevant process instances to a user. It uses simple JDBC protocol rather than JPA because that is all the Vaadin table control needs.

`em-workflow` is the JPA entity manager for the workflow database. This specifes the persistence xml file and a unit name. It also specifies that this entity manager uses Hibernate and the H2 dialect, as well as a datasource called `datasourceWorkflow`.

`datasourceWorkflow` is the Atomikos wrapper for the H2 JDBC datasource and it specifies the url for the database location. It also specifes a unique resource name which is used by the transaction manager. In this case we just supply a constant but for the bundles this is more complex.

`jpaVendorAdapter` specifies some Hibernate switches.

The database configuration included assumes the databases will be created (in memory) when the connection is requested. In production you would more likely have SQL scripts to do this and you would run them beforehand.

The rest of the beans are all relating to transaction management. They ensure that the Atomikos transaction manager is configured properly with Spring, including annotation driven transactions.

6.2. Bundled Databases

Meanwhile the bundles define their own database connections and it is vital that both databases are kept in sync, which is why we need the 2 phase commit support that Atomikos provides. Remember there might be multiple bundles, and each bundle might define a different database. The configuration in the bundle should look like this:

```
...
<bean id="em-local"
 class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="persistenceXmlLocation" value="classpath:persistence-
orderinstances.xml" />
 <property name="persistenceUnitName" value="pu-local" />
 <property name="dataSource" ref="dataSourceLocal" />
 <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
 <property name="jpaDialect">
  <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect" />
 </property>
 <property name="jpaProperties">
  <map>
   <entry key="hibernate.transaction.jta.platform"
 value="nz.co.senanque.hibernate.SpringJtaPlatformAdapter" />
   <entry key="hibernate.dialect"
 value="org.hibernate.dialect.H2Dialect" />
   <entry key="hibernate.format_sql" value="true" />
   <entry key="hibernate.connection.autocommit" value="false" />
  </map>
 </property>
</bean>

<bean id="dataSourceLocal"
 class="com.atomikos.jdbc.AtomikosDataSourceBean" init-method="init"
 destroy-method="close">
    <property name="uniqueResourceName" value="${bundle.name}" />
    <property name="xaDataSourceClassName"
 value="org.h2.jdbcx.JdbcDataSource" />
    <property name="xaProperties">
        <props>
        <prop key="url">jdbc:h2:mem:local;DB_CLOSE_DELAY=-1;MVCC=true</
prop>
        </props>
    </property>
    <property name="maxPoolSize" value="20"/>
</bean>

<bean id="persistenceAnnotation"
 class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" /
>
```

This is simpler than the earlier configuration because the transaction beans are already defined there and shared with this configuration. So all we need here is an entity manager and a data source, and they look much like the ones we already saw.

The one key difference is the unique resource name, which is here set to `${bundle.name}`. This is the name of the bundle plus its version, so if you upgrade a bundle this unique name will still be unique, and it needs to be. The `${bundle.name}` is always set by the bundle manager so you don't have to do anything to set it.

It is worth noting that the `persistenceXmlLocation` refers to a file contained in the bundle and that, in turn, refers to annotated POJOS also contained in the bundle. Also that the reference to the `jpaVendorAdapter` bean is actually a reference to a bean in the main application that is passed to the bundle

7. Locking

While it is common to rely on database for locking the workflow often needs to lock things across transactions so it uses the locking facility from Madura Utils[7]. This has two variants, and it is easy to add more.

For this application the choice is `SimpleLocking` which relies on memory-based flags and only works if there is only one instance of the application running, albeit supporting multiple users. A more likely choice for production is `SQLLocking` which uses its own database connection to store the flags on a database table.

8. JMX

The application supports JMX. Using JMX you can monitor the status of `SimpleLocking` and, if necessary, kill rogue locks. You can also freeze and restart the executor mentioned in *4.4*.

9. Configuring for Production

The process bundles are held in a sweep directory defined as follows:

```
<jee:jndi-lookup id="bundlesDir" jndi-name="java:/comp/env/
WorkflowUIBundlesDir" expected-type="java.lang.String" />
<bean id="bundleManager"
 class="nz.co.senanque.madura.bundle.BundleManagerImpl">
 <property name="directory" ref="bundlesDir"/>
 ...
</bean>
```

You will find the above configuration in `applicationContext.xml`.

The next step is to tell your application server what value to give that JNDI name. This depends on your application server. For Tomcat you can just edit it into your `context.xml` file like this:

```
<Environment name="WorkflowUIBundlesDir" value="MY_DIRECTORY/bundles"
 type="java.lang.String" override="true"/>
```

Finally you want to actually add some bundles to that directory. There are two example bundle projects you can use right away, these are projects inside the main project's `bundles` directory and they are from child projects (simple-workflow and order-workflow). Use those as templates for your own bundles.

You can use this application in production, with appropriate configuration changes, or you can enhance it. Here is a summary of things to look at if you follow the former route.

- Database. Obviously you do not want an in-memory database for a production system. You may want to review whether you want H2 or some other database product more widely used in enterprise applications. Hibernate and Atomikos are also only options that can be replaced by alternatives you may prefer.
- Tomcat. The application is not particularly dependent on Tomcat because it only uses standard JEE facilities. The one area that might be a little tricky is if you want to use WebLogic because it has no easy way of defining a JNDI name to point to a simple string. That problem is solved by [1]. Your reconfigured application will likely make use of JNDI data sources instead of the simpler ones configured here. There are also decisions to be made around how many instances of the application, particularly how many copies of the scheduler are running.
- Security. The hard coded users in the security configuration must be reworked to use your enterprise security facilities. This usually just means adjusting the security configuration file because Spring Security is very comprehensive.
- The scheduler options configured here are probably about right, but your workload might mean they need to be tuned or tweaked, or you might just have different preferences in your enterprise.
- Locking. You will almost certainly need to move from SimpleLock to SQLLock or perhaps something else you prefer.
- The CSS definitions. You do not have to keep the defaults. You can change all the fonts, colours and images and completely rebrand this application if you know enough about CSS.
- Language translations. The application is, we believe, fully i18n compliant. You will want to look at `src/main/resources/messages.properties` and produce a translated version of that. There is already a French one there. You also need to check `localmessages.properties` in the bundles.
- Writing your own workflow definitions, forms, objects and rules. The whole reason for doing this is to get the workflow you really want, so this step is obvious. It is where, hopefully, most of the work will go to get the application where you want it.

10. Building Your Own

But there may be times when it is simpler to deploy a new application for your workflow. This might be because you don't like Vaadin as a UI or perhaps you want to deploy smaller applications to specific user groups, and perhaps you aren't even bothered about using bundles to hold the forms. Maybe you want a very cut down app that someone can run on a tablet. In that case all you really need is a way for them to scan the PROCESSINSTANCE table for records that are in WAIT state and whose Queue Name is the one they are to access. Once they find one they should do the following:

- Lock the process instance using the chosen lock mechanism.
- Change the status to BUSY and write their user name into the LockedBy field.
- Save the record.
- Release the lock.
- Fetch the context information eg the Order or whatever object structure is associated with this process definition.
- Present a form or some kind of input facility for the user to complete and have them indicate when they are done.
- Lock the process instance (again) using the chosen lock mechanism.
- Save the updated context and update the PROCESSINSTANCE status to GO and clear the lockedBy field. This should be a 2 phase commit.
- Release the lock.

That assumes you have the scheduler running in some other application, perhaps this one or a modified version of it. The scheduler application will ensure the process instances move through the process definition while keeping the above application as simple as possible.

A. License

The code specific to MaduraWorkflowUI is licensed under the Apache License 2.0 [14].

The dependent products have compatible licenses specified in their pom files. Madura Rules (optional) has a dual license to cover projects that do not qualify for the Apache License.

B. Release Notes

1.0.1

No actual changes, just a problem with tags.

1.0.0

Initial version.