MaduraWorkflow

User Guide

Document Revision: Revision: 1

Document Revision: Revision: 1

Table of Contents

1. Change Log

Author	Date	Comment
roger.parkinson	Wed May 14	[maven-release-plugin] prepare release madura-workflow-0.0.4

2. References

[1] Madura Objects
[2] Madura Rules
[3] Madura Utils
[4] Madura Workflow UI
[5] Spring Framework
[6] Spring Integration
[7] Apache Commons Beanutils

3. Requirements

- Provide a structured long running process environment, for example an order might run through various stages from initial submission through payment through delivery. This process might require various steps (tasks) handled by different parties over weeks or months. It might require sub-processes to handle specifics of different components of the order. For example an order might consist of a modem and a broadband plan, sub-processes would handle delivery of the modem and provisioning the broadband plan, respectively.
- Some stages of the process require user interaction and this is handled either simply and generically or by adding customised code. There is a UI application[4] which can monitor and control the currently active processes, depending on privileges. The UI application is fully functional and can be extensively customised, or used an example and replaced altogether.
- Process state is saved between tasks for data integrity. If servers go off line while processes are active their state is retained.
- Multiple servers can service the process queues to ensure good throughput.
- Some stages of the process require messages to be exchanged with external systems, for example sending stocking requests to the warehouse system. These messages are handled by a messaging bus provided by Spring Integration[6]. The messages are defined outside the process so the process does not have to be concerned with the transport details, nor with the retry issues. This keeps the processes simpler to understand.
- Processes can define sections that are monitored for elapsed time. A section is one or more tasks. So a process might enter a section that will, after a time, escalate and send a message to a supervisor alerting them to a hold up. If the task is finished within the time defined the escalation does not happen. This feature can also be used to monitor SOAs.
- Each process instance logs what happened to it for audit purposes.
- The objects involved in a process instance are optionally monitored by Madura Rules[2]. This simplifies the decisions in a process because all decisions use simple booleans rather than complex expressions. Rules will fire to set the appropriate flags and the process only needs to query those flags. This means that the business rules are only in one place rather than some being in the process and some being in custom code etc. That ensures the same rules apply to this object regardless of where it is used.
- Process definitions can be updated on the fly without bouncing the servers. This is managed carefully to ensure that existing process instances continue with the same environment (process definition, rules, objects) while new process instances use the new environment. Thus you can have two or more environments running simultaneously.
- Task Instances that require user interaction are placed into queues. These can be monitored by users with the correct privileges and actioned with the appropriate form, or possibly by some custom program that updates the database status correctly.

4. Terminology

To save confusion we need to define some terms.

Process Definition	This is the static definition of a process eg the Order Process. It may be overidden by a new version and old, unused versions may be dropped. It is normally tied to a version of the rules and objects.
Process Instance	An actual running process as opposed to its definition, for example and order. This is always tied to a process definition.
Task	An step in the process definition. Tasks may be one of several types: a Form, which presents a form to a user for interaction, a Compute step which invokes some custom code and a Message which handles sending a message and getting a response back. Further customisations of the generic Task can be added.
Task Instance	An actual running task, as opposed to its definition.

5. Process Definitions

A process definition looks something like this:

```
queue: name="Q1" permission="ORDERCLERK";
queue: name="Q2" permission="STOCKCLERK" read-permission="ORDERCLERK";
queue: name="Q3" permission="SUPERVISOR" read-permission="ORDERCLERK";

process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   if (decisionField) {
      try {
         form=SecondForm queue="Q2";
         compute=orderCompute log="some message";
      }
      catch (timeout=10 HOURS) {
         form=SupervisorForm queue="Q3";
         retry;
      }
      if (rejected) {
         abort "Rejected this order"
      } else {
         message=stockManagementSender;
      }
   }
   message=orderProvisioningSender;
}
```

This defines a process called `AcceptOrder` that focuses on an `Order` object. An order clerk, ie any login who has the permission ORDERCLERK, is able to launch the process by filling in the `InitialForm`. The details of the form may vary depending on what software the order clerk is running to launch the process, but we can assume there is always some sort of form which gathers information about the order.

In this case the Order object is monitored by Madura Rules and there is a rule which fires under certain conditions that sets the `decisionField` to true. So once the process is launched the `decisionField` is examined by the 'if' clause and the fairly obvious logic happens.

If the `decisionField` is true the `SecondForm` is sent to `Q2`. Anyone with permission STOCKCLERK can see the process waiting for the `SecondForm` and update it. The order clerk can see it but cannot update it.

Once the stock clerk has completed the update the process executes a piece of custom computing and moves on to the second 'if' clause.

The second 'if' clause tests the `rejected` and aborts the process if it is true. Otherwise it sends a message to the stock management system and waits for a reply.

Finally it sends a message to the order provisioning system.

The stock clerks are busy and they don't always check their queue so if the process waits too long for them to complete the `SecondForm` it will invoke the 'catch' clause and place the `SupervisorForm` into Q3. Again, the order clerk can still examine the process, but only someone with SUPERVISOR permission can update it. Essentially this is escalating the order. Once the supervisor has taken a look the order is retried, in this case sent back to the stock clerk's queue.

Because we are focused on an `Order` object, as specified in the first line, any field references (`decisionField, rejected`) are understood to mean `order.decisionField` and `order.rejected` etc. The forms used here present the order in a form to the user, with some optional customisation.

The compute task is a Spring bean[5] that implements `nz.co.senanque.process.instances.ComputeType`.

In two places there is a decision which queries a field on the order. The field may have been set by a rule based on other fields in the order or in objects attached to it, or it might be set by a Form. The rejected flag, or example, might be a checkbox.

Finally there are two message tasks which translate the order into the target message format and send it. The target message defines which fields from the order it needs and where they need to be formatted into the message, it also knows where to send it, how to handle retries and communications errors.

Between each task in the process the process instance and its context (the order in this case) is saved to a database and requeued. This means that the two forms mentioned are two distinct steps, even if they end up being done by the same user, and they might be done weeks apart. But that is not strictly true, some rationalisation happens depending on the particular tasks. For example two compute tasks next to each other will actually be executed one after the other without saving to the database. It is best to assume that saving to the database is done between tasks though.

This is the complete list of task types:

5.1. Sub processes and Loops

Sometimes we might have an order with many order items attached and we want to launch a subprocess on each order item and wait until all are completed (or one is cancelled).

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   do {
      if (inStock) {
         form=ReserveStockForm queue="Q2";
      } else {
         form=BackorderStock queue="Q2";
      }
   } for orderItems;
   form=FinalizeOrder queue="Q1";
   message=orderMessageSender;
}
```

In the next example the process is split in two. The `AcceptOrder` process has an initial form, and in that the user would enter the order header information as well as the order items. In the second task we launch the `OrderItemProcess` for each order item and wait for *all* the subprocesses to complete before continuing.

Subprocesses can be in-line like the above example or separated out, the functionality is the same.

```
process: OrderItem "OrderItemProcess" {
   if (inStock) {
      form=ReserveStockForm queue="Q2";
   } else {
      form=BackorderStock queue="Q2";
   }
}
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   do "OrderItemProcess" for orderItems;
   form=FinalizeOrder queue="Q1";
   message=orderMessageSender;
}
```

Note that in both cases the parent process always waits until all the children are complete. There is a special case where one or more of the children abort. In that case all children are aborted and so is the parent process. Child processes need to take care to catch abort signals and handle them where at all possible.

There is a third option with subprocesses called the `fork`.

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   fork "subprocess1","subprocess2";
   form=FinalizeOrder queue="Q1";
   message=orderMessageSender;
}
```

This will launch two subprocesses: subprocess1 and subprocess2 and wait for both to complete. An important difference between this and the 'do' syntax is that the subprocesses run with the same context as the parent process, so in this case all three have the `Order` as their context, rather than the `OrderItem`.

Finally there are two more variations on `do` namely the `do-while` and `do-until`.

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
 do {
  form=SecondForm queue="OrderSupervisor";
 } while (decisionField);
 compute=orderCompute log="final";
}
```

In this case the loop will continue while the `decisionField` is true. The field must be a boolean and it must, of course, be a field on the Order. In this case we assume that the initial form sets the `decisionField` to true, or perhaps it defaults to true, and that the SecondForm has a way of setting it to false so that the loop can exit. Be careful here, Java programmers will be tempted to think a boolean expression can be the condition. This is not the case, it can only be a boolean field.

If you really need the negative expression eg `!decisionField` you can use the `do-until` like this:

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
 do {
  form=SecondForm queue="OrderSupervisor";
 } until (decisionField);
 compute=orderCompute log="final";
}
```

In this case the field starts out as false and the form must set it to true at some stage to exit the loop.

Both of these can accept in-line tasks, as shown, or they can refer to a separate sub-process as shown in the earlier examples.

Loops like these are useful when you have a need to pass through a lot of approvers with the option of passing the order back to people who have already looked at it.

5.2. try/catch

Exceptions can be caught using the try/catch syntax and we already saw an example of catching a timeout. The other thing a process can catch is an abort.

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   try {
      form=SecondForm queue="Q2";
      form=ThirdForm queue="Q1";
   }
   catch (abort) {
      compute=orderCompute;
```

```
   }
   message=orderMessageSender;
}
```

In this case the abort might come from the `SecondForm` form, possibly the user pressed a button marked 'Abort' and that triggered the abort condition. All the form has to do is write 'ABORTING' to the process instance status field. In this case the process traps the abort and executes a compute task, perhaps to fix things up, and then optimistically proceeds to the `orderMessageSender` message. If only the real world were that simple!

There are three other options for the catch, in terms of proceeding anyway. The catch can retry, which we already saw with the timeout example. The retry will go back to the task that generated the abort and run it again. In this case the `SecondForm` task instance will re-appear in the stock clerk's queue. A retry looks like this:

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   try {
      form=SecondForm queue="Q2";
      form=ThirdForm queue="Q1";
   }
   catch (abort) {
      compute=orderCompute;
      retry;
   }
   message=orderMessageSender;
}
```

Or the catch might continue, which means it executes the statement *after* the one that caused the abort. So if the `SecondForm` aborted the process would execute `orderCompute` and then go to the `ThirdForm`.

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   try {
      form=SecondForm queue="Q2";
      form=ThirdForm queue="Q1";
   }
   catch (abort) {
      compute=orderCompute;
      continue;
   }
   message=orderMessageSender;
}
```

The final choice for the process is to explicitly abort like this:

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   try {
      form=SecondForm queue="Q2";
      form=ThirdForm queue="Q1";
   }
   catch (abort) {
      compute=orderCompute;
      abort;
   }
   message=orderMessageSender;
}
```

The catcher just ensures `orderCompute` runs before doing the abort which stops the process and the `orderMessageSender` message will not be sent. It is, fairly obviously, a bad idea to retry an abort.

All of these options are available with the timeout catcher, the only difference is that you must supply a time value (eg number of days etc) before the timeout happens. The time should be regarded as approximate, ie to within a few minutes. It is also valid to have multiple catchers at once like this:

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   try {
      form=SecondForm queue="Q2";
      form=ThirdForm queue="Q1";
   } catch (timeout=10) {
      form=SupervisorForm queue="Q3";
      retry;
   } catch (abort) {
      compute=orderCompute;
      abort;
   }
   message=orderMessageSender;
}
```

And they can be nested to any depth.

The timeout accepts a figure in milliseconds which is not always convenient when you want to specify, say, 3 weeks. So there is an optional qualifier of time units:

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   if (decisionField) {
      try {
         form=SecondForm queue="Q2";
         compute=orderCompute;
      } catch (timeout=3 WEEKS) {
         form=SupervisorForm queue="Q3";
      }
   }
}
```

Valid time units are: SECONDS, MINUTES, HOURS, DAYS, WEEKS.

But the timeout value can be much more complex if you are prepared to write a custom class to calculate it. The custom class must implement `nz.co.senanque.process.instances.TimeoutProvider` and you could write one, for example, to figure how long until next Easter and have the timeout happen then. The syntax looks like this:

```
process: Order "AcceptOrder" "Accept an order" launchForm=InitialForm
 queue="Q1" {
   if (decisionField) {
      try {
         form=SecondForm queue="Q2";
         compute=orderCompute;
      } catch (timeout=com.mydomain.TimeToEaster) {
         form=SupervisorForm queue="Q3";
      }
   }
}
```

Here is a summary of the options:

retry	Go back to the task that caused the problem and run it again.
continue	Go to the task *after* the one that caused the problem.
abort	Send an abort exception which will be caught by any surrounding try/catch.

If you use none of these options the next task after the catch block will be run, ie it will simply fall out of the catch block to the rest of the process in the same way Java code does.

6. Compute

This is the simplest task to add. All that is required is a Java class that implements `nz.co.senanque.process.instances.ComputeType` Here is a simple example:

```
public class OrderItemComputeClass implements ComputeType<OrderItem> {

   private static final Logger log = LoggerFactory
      .getLogger(SomeComputeClass.class);
   public void execute(ProcessInstance processInstance,
         OrderItem context, Map<String, String> map) {
      String abort = map.get("abort");
      log.debug(map.get("log"));
      if (abort != null && abort.equals("true")) {
         throw new RuntimeException("generated abort");
      }
   }
}
```

The execute method is called by the workflow to execute the task. It passes the process instance, the context (the `OrderItem` from our examples) and a map of the arguments in the argument list. There is no need to do any database fetches or updates, these are done outside the class. The full definition of the task in the process, including the argument list, looks like this:

```
compute=orderItemCompute log="final" abort="false";
```

Using the argument list allows you to write generic compute classes and use them in different situations. Also notice that this one throws an exception. Any uncaught exception thrown from a compute class is treated as an abort by the workflow process.

The actual `orderItemCompute` is defined as an ordinary Spring bean in your beans file, something like this:

```
<bean id="orderCompute" class="mydomain.OrderComputeClass" />
```

That's all you need. Workflow will find the configured bean and use it in the process definition. Because it is a bean you don't necessarily need to use the argument list because, of course, you can inject values. But if you want to use the *same* bean in different places in your process you might then find the argument list useful.

7. Forms

We have already seen several examples of processes that use forms and now we can look at them in more detail.

```
process: Order "Process1" "my description" launchForm=MyLaunchForm
 queue="Q1" {
   try {
      form=SomeOtherForm queue="Q2";
      compute=orderCompute log="#1";
   } catch (abort) {
      compute=orderCompute log="#2";
      continue;
   }
   compute=orderCompute log="#3";
}
```

This example shows two different forms, and they are both used differently so let's look at them one at a time.

The first form is `MyLaunchForm`. This is a launch form and the idea is to present this form *before* the process starts. The form may signal an abort or cancel and if it does the process will not be started, which means there is nothing to clean up. If it signals okay the process will be started.

The queue associated with a launch form is somewhat misleading because the form does not actually go into a queue like the in-process forms. But queues are a handy way to assign permissions so by specifying a queue we can tell what users are allowed to launch this process.

Launch forms are optional. Some processes can be launched without a form. This always applies to subprocesses.

The second use of forms is as in-process forms. In the example the `SomeOtherForm` is an in-process form. An in-process form is an actual task, or step, in the process unlike the launch form which is used before the process starts.

Once the process runs an in-process form it places it in the queue and then waits for some user to complete the form and save the result. To be clear about just what is going on, the process instance is just a database row and it has a status field that is set to 'wait' and a queue name field that it sets to, in this case, 'Q2'. Thereafter when the workflow is looking for work by scanning the table it ignores this until its status changes.

The users must operate an application of some kind, such as the UI, that allows them to scan for forms they can operate, ie which they have permission to operate.

It is important that the workflow process definition does not care what kind of technology is being used to define the forms. All it knows about the form is that it has a name. The rest is the responsibility of the UI. There are a lot of UI technologies, probably too many, that offer ways to deliver a UI to a user. You get to choose whatever you want, and you may have particular requirements. The best we can do is offer a good example of one and suggest ways to adapt it to another environment. The example we offer is built in Vaadin which provides the excellent combination of a web based application with a Swing-like programming model. It also helps that the integration work to transparently link Vaadin with Madura Objects and Madura Rules.

Whatever the technology chosen it must do the following:

- Provide some kind of security token that identifies the users and their assigned permissions.
- Use those permissions to present the users with a list of process instances they are allowed to see (ie the permissions match)
- Also present the users with a list of processes they are allowed to launch.
- When users indicate they want to launch a process locate the launch form name and present something that is relevant to that, gather the data for the context object and save both that and the ProcessInstance record to the database.

- When users indicate they want to operate a waiting form locate the form name, the ProcessInstance and context records and present something relevant to the form name. When the form is complete save the updated, the new status should be GO.
- Deliver some kind of 'form' or list of fields that can be assigned values when requested for each form name. Save the result to the relevant database object.
- Ensure the locking of process instance records is pessimistic.
- Ensure any business rules relating to the context object are enforced correctly.

The reason the process never defines the form itself, it just provides its name, is to allow the possibility of there being multiple UI technologies active on the same process at the same time. A simple scenario might be that the order clerks all use a browser interface and are quite happy with Vaadin for that. But some external constraints prevent the stock clerks from using a browser on their machines so they cannot use the same application. Instead the convenient solution is to deploy a Swing program to their machines which provides the same functions. When the Swing application requests a form by name it needs to get a Swing form, not a Vaadin form, and process that. Since the process itself does not specify what *kind* of form you might have two forms of the same name but for different technologies.

The sample UI, which does all this, is documented in [4].

There is a requirement stated earlier*1* that processes must be able to be updated without shutting down the workflow system. Actually this especially applies to the user interface. In a high volume/high availability application it is not acceptable to take the application off line to do upgrades. It implies that the way the processes are packaged must be self contained. For example the forms must be defined in the process package that uses them so that both the form and the process can be updated the same way. This appears to conflict with the scenario just raised with the stock clerks and the Swing program. The answer is that the process has to be packaged with *all* the forms for all the technologies in use, so they are all upgraded together and always consistent.

In the sample UI these issues are addressed using Madura Bundles and the packaging model is described there.

8. Attachments

The database has room for attachments. There is an Attachment table which has a blob field for the attachment. Attachments are made to the process instance and they remain with the process instance throughout its life, unless they are deliberately deleted. However attachments are not used directly by the processes, there are there for users to refer to during manual tasks.

There may be restictions around the size of individual documents depending on what database product you choose to store the data in. Check how your database handles blob fields for specifics.

9. Messaging

In simple terms a message task forms a message of some kind, typically a web services message. All of our examples here assume Spring Integration[6] because that is what we have tested with. Other similar products might be used because the integration points with Madura Workflow are few. For now, however, we are quite happy with Spring Integration.

```
message=orderMessageSender;
```

This is an example of the message task. The message is formed from the context (eg the Order). Details of how the message is formed and how it is unpacked are of no concern to the process designer who only wants to know that a message is sent at this point in the process and a response of some kind came back and was unpacked appropriately. Of course that means a lot has to happen in between.

The first thing to understand is that the `orderMessageSender` is a Spring bean, defined in the beans file. It is some class that implements `nz.co.senanque.messaging.MessageSender`. Here is an example:

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

Actually the `MessageSenderImpl` class is fairly generic. It transforms the context into an XML document and sends it to the Spring Integration channel named `orderChannel`. It is actually a little more complex than that. If there is a marshaller defined in the beans file then it will use it to marshal to some format, eg XML. If there is none defined it will make a message out of the current context, eg the Order object. For these examples we will assume the marshaller is defined and translates to an XML document.

9.1. Simple Spring Integration Example

The usual case is that the message is sent through some transport like Web Services or JMS and a response comes back which we may or may not be interested in. But it is likely we want to unpack the response in some way, and especially we want to know it tells us about an error.

The order channel might be configured like this:

```
<int:chain input-channel="orderChannel">
   <!-- The delayer ensures the message is sent in a different thread from
 the sender -->
   <int:delayer id="delayer" default-delay="3000" />
   <int-xml:xslt-transformer
      xsl-resource="classpath:nz/co/senanque/workflow/
WorkflowRetryTransformer.xsl"/>
   <int-ws:header-enricher>
      <int-ws:soap-action value="http://tempuri.org/FahrenheitToCelsius" /
>
   </int-ws:header-enricher>
   <int-ws:outbound-gateway uri="http://www.w3schools.com/webservices/
tempconvert.asmx" />
</int:chain>
```

In this example we are using Spring Integration to do the following:

1. Add a delay. This ensures the message is processed in a different thread from the one that ran the message sender. That means the first thread will release the process instance and context and wait for the response message without holding onto anything.
2. Transform our original message (which was just the order) into the SOAP message the target system requires using XSL.
3. Add the soap action to the message.
4. Send the message to the target URL.

Note that Spring Integration is a rich product and has many options. There are lots of other things you can add to this simple example.

The response will be delivered to the `replyChannel` which might be configured like this:

```
<int:chain input-channel="orderReplyChannel">
   <int-xml:xslt-transformer
      xsl-resource="classpath:nz/co/senanque/workflow/
WorkflowRetryTransformer2.xsl"/>
   <int:service-activator ref="genericEndpoint"
 method="issueResponseFor" />
</int:chain>

<bean id="genericEndpoint"
 class="nz.co.senanque.messaging.GenericEndpoint" />
```

This transforms the incoming message to look something like this:

```
<Order>
   <celsius>32.2</celsius>
</Order>
```

The `GenericEndpoint` can unpack this format into the context, simply setting each field into the context object (eg the order). It doesn't care what the name of the first element is. There is more detail on this format in *9.4*

In practice you would definitely want to add some complication to these channel configurations, notably you would want to add some retry logic to both the outgoing and the incoming message.

Why the incoming message? This is to handle the situation where the process instance is locked just at the moment the reply arrives. Rather than give up and crash the retry logic will back off and retry several times.

9.2. Adding a Retry Template

While this is not a manual of Spring Integration it is worth setting out one way to organise the retry. First you need a template like this:

```
<bean id="retryTemplate"
 class="org.springframework.retry.support.RetryTemplate">
   <property name="retryPolicy">
      <bean class="org.springframework.retry.policy.SimpleRetryPolicy">
         <constructor-arg index="0" value="10" />
         <constructor-arg index="1">
            <map>
               <!-- retryable (assuming target is down) -->
               <entry
 key="org.springframework.integration.MessagingException" value="true" />
               <entry
 key="nz.co.senanque.messaging.WorkflowRetryableException" value="true" />
            </map>
         </constructor-arg>
      </bean>
```

```
   </property>
   <property name="backOffPolicy">
      <bean
 class="org.springframework.retry.backoff.ExponentialBackOffPolicy">
         <property name="initialInterval" value="1000" />
         <property name="multiplier" value="5" />
      </bean>
   </property>
</bean>
```

This template will try a maximum of 10 times and back off for 5 seconds, then 25 seconds and so on. It will only retry the two exceptions named: `MessagingException` and `WorkflowRetryableException`. It is possible one template with your selected values will do for all your needs, but you might need different ones for different kinds of messages.

Why did we say MessagingException is retryable, aren't we dead then? Actually no. You get this when there is some networking problem or if the target system is down. In these cases the answer is to fix the network and let the message go through rather than send an error to the process instance. The only kinds of errors the process instance ought to hear about are errors that can never be recovered from.

To use this template in our earlier messages we need to add an advice handler to the outbound gateway and the service activator respectively like this:

```
<int-ws:outbound-gateway uri="http://www.w3schools.com/webservices/
tempconvert.asmx" >
   <int-ws:request-handler-advice-chain>
      <bean
 class="org.springframework.integration.handler.advice.RequestHandlerRetryAdvice">
         <property name="recoveryCallback">
            <bean

 class="org.springframework.integration.handler.advice.ErrorMessageSendingRecoverer">
               <constructor-arg ref="recoverChannel" />
            </bean>
         </property>
         <property name="retryTemplate" ref="retryTemplate" />
      </bean>
   </int-ws:request-handler-advice-chain>
</int-ws:outbound-gateway>
```

And this:

```
<int:service-activator ref="genericEndpoint" method="issueResponseFor" >
   <int:request-handler-advice-chain>
      <bean
 class="org.springframework.integration.handler.advice.RequestHandlerRetryAdvice">
         <property name="recoveryCallback">
            <bean

 class="org.springframework.integration.handler.advice.ErrorMessageSendingRecoverer">
               <constructor-arg ref="recoverChannel" />
            </bean>
         </property>
         <property name="retryTemplate" ref="retryTemplate" />
      </bean>
    </int:request-handler-advice-chain>
</int:service-activator>
```

But, as you can see, we added something called the `recoverChannel`, which is another message channel that needs to be configured something like this:

```
<int:chain input-channel="recoverChannel">
   <int:service-activator ref="errorEndpoint"
 method="processErrorMessage">
      <int:request-handler-advice-chain>
         <bean
 class="org.springframework.integration.handler.advice.RequestHandlerRetryAdvice">
            <property name="recoveryCallback">
               <bean

 class="org.springframework.integration.handler.advice.ErrorMessageSendingRecoverer">
                  <constructor-arg ref="recoverChannel" />
               </bean>
            </property>
            <property name="retryTemplate" ref="retryTemplate" />
         </bean>
      </int:request-handler-advice-chain>
   </int:service-activator>
</int:chain>

<bean id="errorEndpoint" class="nz.co.senanque.messaging.ErrorEndpoint" />
```

The `ErrorEndpoint` always sends an abort to the process instance. Like the other channels it uses the retry template because it too can encounter a lock condition.

9.3. Further Steps

The above examples barely scratch the surface of Spring Integration. In a production system you would probably want to add a message store to ensure that messages that are still retrying survive a system restart and also support multiple server instances running efficiently. These kinds of decisions hinge around what kinds of transports you are using, for example JMS already stores messages.

There are also ways to monitor and adjust the configuration using JMX that ought to be considered in a production environment. These kinds of options are covered in the Spring Integration documentation and are out of scope of this document.

9.4. GenericEndpoint format

`GenericEndpoint` is designed to handle incoming XML messages and unpack them into the current context. The simplest case is that the message needs to update one or more fields in just one object, the context object. This is an example of the format:

```
<Order>
   <celsius>32.2</celsius>
   <decisionField>true</decisionField>
   <junkField>something</junkField>
</Order>
```

The outer tag name, Order, is ignored. We always assume we are updating the current context of the process whatever it is. What we actually use are the inner names like celsius, decisionField and junkField. We assume these are fields on the current context object and we set the values. In the case of junkField there is no such field on the Order object, which does not cause an error, it just ignores those. Similarly problems of type casting are ignored.

The next level of complication is to add xpath attributes. Before explaining that you need to understand that to set the values the code just calls `org.apache.commons.beanutils.PropertyUtils.setProperty(context, name, value)` [7] and that allows some more complex semantics in the name than just the field name. For

example the call will take an xpath expression like this: `"orderItems[3].approved"` to access a field in the attached list of orderItems. To make use of this you use an xpath attribute like this:

```
<Order>
   <celsius>32.2</celsius>
   <decisionField>true</decisionField>
   <junkField>something</junkField>
   <ignored xpath="orderItems[3].approved">true</ignored>
</Order>
```

The tag `ignored` is not used if there is an xpath attribute, but it needs to be unique in this level of the document to conform to XML.

The final variant is when the incoming message needs to signal an error:

```
<Order error="Not a valid request">
   <celsius>32.2</celsius>
   <decisionField>true</decisionField>
   <junkField>something</junkField>
   <ignored xpath="orderItems[3].approved">true</ignored>
</Order>
```

Because of the error attribute the rest of the document is ignored. The error message is turned into an abort and passed to the process instance.

The external services will not, of course, be interested in delivering a format compatible with `GenericEndpoint`. Instead you provide an XSL file which translates from their format to this one. The other choice is to ignore `GenericEndpoint` and write your own endpoint in Java or, at least, the unpacking part of it.

To do that you write an implementation of `nz.co.senanque.messaging.MessageMapper` and inject it into the `GenericEndpoint` like this:

```
<bean id="genericEndpoint"
 class="nz.co.senanque.messaging.GenericEndpoint">
   <property name="messageMapper">
      <bean class="mydomain.MyMessageMapper">
   </property>
</bean>
```

The message mapper you implment just has to unpack the message into the context however you want. If you decide the message describes an error then throw an exception. The locking, fetching and retry logic are handled before the message mapper is called.

10. Locking and Database

The database requirement is done by JPA, so any valid implementation of JPA should be fine. But locking is done outside the database mostly because we need a pessimistic locking system that survives database transactions.

The Madura Locking project, part of Madura Utils[3], supplies the framework for the locking system, but the implementation of the locks are your choice. Details on customising the locking are gven in the Madura Utils documenation.

10.1. Locking Strategy

Because there can be multiple concurrent processes attempting to operate on different levels of the context in a process instance the locking must be managed carefully. Locking a process instance is straightforward because it is only one record. Child processes have to update their parent process when they complete, but they do this through a deferred process which only locks one process instance, ie the parent. This avoids the possibility of deadlocks because each thread only ever holds a lock on one process instance.

Locking the context is more complex because of the following scenarios:

- One process which wants to update the top of its context, for example the process controls an Order and wants to update it. This is not a problem at all and we can just about get away without using a lock if we assume that the process instance has to be locked before anyone acesses the Order.
- The process launches a subprocess for the three orderItems using a 'do'. The main process is waiting until the subprocesses are done so it doesn't matter. The three subprocesses have separate contexts. A convention that you can update your own context and below means that, as above, you can just about get away without locking.
- The process launches two processes using a fork. This will leave the parent process waiting for the subprocesses, making it not a problem. But the two subprocesses are a problem because, using the conventions above, they are free to update the same part of the context whenever they like. To solve this we add a second lock to every scenario which locks the context as well. Before a task can run it must obtain a lock on the process instance as well as a lock on the context.

The locks actually refer to database rows and they are always formed from a string that names the class that represents the object type and the id, like this `nz.co.senanque.workflowtest.instances.Order@25`. To avoid deadlocks the order of locking is important, the process instance is locked first and the context second. The unlock happens in the *reverse* order.

10.2. SQL Script

The Workflow project needs a JPA database which is defined in the XSD file and it generates JPA Entities from that. Using those Entities the test code is able to use Hibernate to generate a scratch database. But for production you usually need an SQL file to create your database. There is an SQL file suitable for an H2 database distributed with the project that is generated using the SchemaTranslatorMavenPlugin like this:

```
<plugin>
 <groupId>nz.co.senanque</groupId>
 <artifactId>schema-translator-maven-plugin</artifactId>
 <version>0.0.1</version>
 <executions>
  <execution>
   <id>sql-generate</id>
   <phase>package</phase>
   <goals>
    <goal>sql</goal>
   </goals>
```

```
  </execution>
 </executions>
 <configuration>
  <persistenceFile>src/main/resources/META-INF/persistence-workflow.xml</
persistenceFile>
  <persistenceUnit>pu-workflow</persistenceUnit>
  <dialect>org.hibernate.dialect.H2Dialect</dialect>
 </configuration>
 <dependencies>
  <!-- Need this because the entity classes refer to items in
 MaduraObjects -->
  <dependency>
   <groupId>nz.co.senanque</groupId>
   <artifactId>madura-objects</artifactId>
   <version>2.2.2</version>
  </dependency>
 </dependencies>
</plugin>
```

This plugin simply wraps Hibernate's ability to generate databases from the JPA entites. In this case it outputs not the database itself but the SQL script it would use to create it. By default the output is to `target/${project.artifactId}-${project.version}.sql` but if you may want to override the default name or location using destdir and destFile.

Finally the drop parameter, which defaults to false, can be used to include the drop statements in the resulting SQL file.

11. Permissions

These are only relevant when we have on-line users operating forms. Permissions therefore only apply to form tasks. More precisely they apply to queues.

```
...
queue: name="Q3"
   permission="SUPERVISOR"
   read-permission="STOCKSUPERVISOR";

process: Order "AcceptOrder" {
 form=InitialForm queue="OrderClerk";
   if (decisionField) {
      try {
         form=SecondForm queue="Q2";
         compute==orderCompute;
      } catch (timeout=10) {
         form=SupervisorForm queue="Q3";
         retry;
      }
      if (rejected) {
         abort "Rejected this order"
      } else {
         message=StockManagement;
      }
   }
   message=orderMessageSender;
}
```

This tells the system that the queue named Q3 can be operated with any user who has SUPERVISOR permission. It can also be displayed by any user that has STOCKSUPERVISOR permission. The actual implementation of this is done in the UI[4], it is not enforced at all in the core workflow.

A. License

The code specific to MaduraWorkflow is licenced under the Apache License 2.0 .

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

B. BPEL

This is not the only product of its type around and the others mostly, but not all, implement a standard called BPEL (Business Process Execution Language), which is an XML based process definition language. For example SAP, Oracle and Microsoft have products that work that way. We've used jBPM extensively in the past and it uses BPMN (Business Process Model and Notation) to present a graphical representation of a process and an underlying BPEL definition.

There are other choices too, such as YAWL, which dispenses with BPEL.

We think the various extensions to BPEL and the alternatives to it weakens its position. The goal for having a standard workflow language needs re-evaluating. There are two reasons for it:

- The ability to re-use process definitions across different products making migration between workflow engines simpler.
- To make it easier to people to write definitions for different engines, ideally business analysts rather than programmers can define the definitions.

In practice the various extensions the individual products add to their implementations preclude migrating them in any useful way and engineering all but the simplest business process is too specialised for a business analyst to handle. The BPEL engines implement messaging to a greater or lesser extent and that adds to their complexity which, in turn, adds to the extensions they need to implement. Also BPEL lacks explicit user interactions, which we think is a good thing in that it keeps it simpler, but it does mean the different implementations have to cover this requirement with more extensions, compromising portability again.

In contrast the Madura approach is to keep the messaging and user interactions external. This simplifies the process definitions, not to mention the amount of code we have to write.

For messaging we use Spring Integration, with a very loose coupling so that it could be replaced with something else. If it were replaced the process definitions would not change.

For user interraction we leave the options open. There is a full example of implementing user interraction using a mix of Vaadin and Madura Objects, which is our preferred technology for UI, but the final choice is up to the implementor. There are so many UI technologies that locking on to a specific one would be too constraining.

Finally, the process definitions look as much like Java as possible on the understanding that Java programmers are easy to find and they ought to be able to learn the environment in a very short time, possibly minutes. For the same reason we resisted the idea of making the processes graphical. In our experience adding graphical representation just limits the flexibility the process designer requires, and often makes it more rather than less complex.

C. Release Notes

0.0.4

Changes to support Eclipse plugin. These all relate to parser behaviour.

0.1

Initial version.