****

**MDW Tomcat Cookbook – Table of Contents**

**Prerequisites:**

[**Eclipse with the MDW Plug-In**](#eclipse)

Get Started with BPM Cloud Development.

[**Java Container (Apache Tomcat 7.x)**](#java)

Run Locally for Advanced Capabilities.

**Local Cloud Development:**

[**1: Create a Local Cloud Project**](#local_1)

Use the MDW wizard to create a workflow-faceted Java project.

[**2: Build a Workflow Process**](#local_2)

Create a basic process definition with Dynamic Java Code.

[**3: Run Your Process**](#local_3)

Execute your process and debug your custom code.

[**4: Consume a Web Service**](#local_4)

Use the Document Web Service adapter activity to invoke a SOAP web service.

[**5: Expose Your Process to External Systems**](#local_5)

Register an MDW service-oriented Event Handler to launch your process.

**Advanced Cloud Development:**

[**1. Build a Web Page to Capture User Input**](#advanced_1)

Launch your process from a custom web page.

[**2. Expose an Entire Custom WebApp**](#advanced_2)

Override MDW's built-in MDWHub WebApp.

[**3. Design a Report and Access it through the Web**](#advanced_3)

Create a simple BIRT report based on an MDW template.

**MDW Tomcat Cookbook – Eclipse Setup**

[Contents](#contents) [Next](#java)

**Eclipse with the MDW Plug-In**

**Set Up Eclipse on Your PC:**

* Make sure you have Java Runtime 7 or 8 installed on your computer:
  + [**http://www.oracle.com/technetwork/java/javase/downloads**](http://www.oracle.com/technetwork/java/javase/downloads)
* Install Eclipse Mars (4.5.x) for JavaEE Developers:
  + [**https://www.eclipse.org/downloads**](https://www.eclipse.org/downloads)
* Launch Eclipse. Here’s an example command line that includes appropriate JVM memory settings for running and debugging:

**C:\eclipse\_4.5.2\eclipse.exe -vm C:\jdk1.7.0\_45\bin\javaw.exe -vmargs -Xms512m -Xmx1024m**

* Install the latest version of the MDW Plug-In via Eclipse Software Updates (Help > Install New Software > Add > **http://lxdenvmtc143.dev.qintra.com:6101/MdwPlugin** > Install).
* Make sure Eclipse has a Java 7 or Java 8 Installed JRE available in your workspace (Window > Preferences > Java > Installed JREs). ***For deploying on Tomcat you will need to install JDK 7 or 8.***

**Optional Plug-Ins:**

* Groovy Eclipse Plug-In (Groovy script syntax highlighting, auto-complete, etc.)

**http://dist.springsource.org/release/GRECLIPSE/e4.5**

* Quantum DB Plug-In (Database querying)

**http://quantum.sourceforge.net/update-site**

**MDW Tomcat Cookbook – Container**

[Previous](#eclipse) [Contents](#contents) [Next](#cloud)

**Java Containers**

You can perform many cloud development activities using a remote workflow project. However, there are certain advantages to being able to deploy locally. The differences between local and remote development are described in later sections of this tutorial. To be able to develop locally you need one of the following containers installed. Chances are if you're deploying in Cloud Mode you'll choose Apache Tomcat.

**Supported Java Containers:**

* Apache Tomcat:
  + [**http://tomcat.apache.org/download-70.cgi**](http://tomcat.apache.org/download-70.cgi)
* Apache ServiceMix:
  + [**http://servicemix.apache.org/downloads.html**](http://servicemix.apache.org/downloads.html)
* JBoss Fuse 6.1.0:
  + [**http://www.jboss.org/products/fuse/download**](http://www.jboss.org/products/fuse/download)

**MDW Database:**

* Starting with version 5.5, MDW saves the workflow assets you create on your local file system until you commit them to a version control repository such as Git. Runtime data is stored in a MySQL or Oracle database. Supported versions are listed in the MDW [Compatibility Matrix](http://cshare.ad.qintra.com/sites/MDW/Releases/Compatibility). Generally for cloud development you'll point to a pre-existing central database. If you want to host your own database, you'll need to configure an instance of MySQL with the MDW db schema. The SQL scripts for installing the MDW schema are available here:
  + [**http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/MDW%20Database**](http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/MDW%20Database)

**Local Cloud Development**

[Previous](#java) [Contents](#contents) [Next](#local_2)

**1. Create a Local Cloud Project**

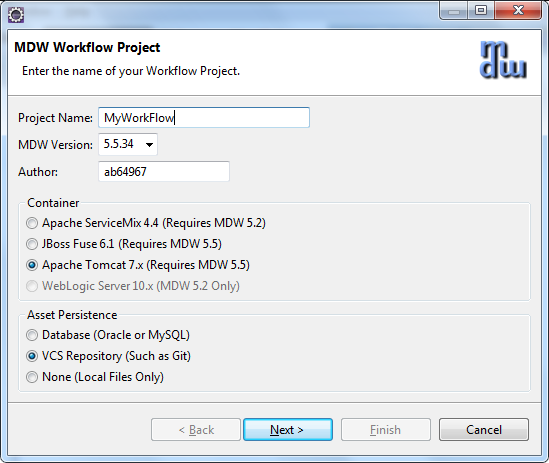
A local cloud project is useful if you want to debug your custom Java source code and Groovy scripts. The standard MDW war file is deployed as part of the steps outlined in this tutorial.

**Open the Designer Perspective in Eclipse:**

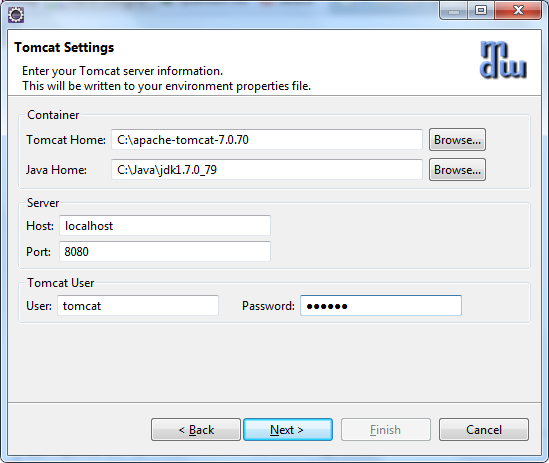
* Launch Eclipse (with the MDW Plug-In installed).
* From the menus select Window > Open Perspective > Other > MDW Designer.
* For detailed documentation covering Designer, refer to the User Guide:
  + [**http://qshare.ad.qintra.com/sites/MDW/User%20Documentation/MDW%20Designer%20User%20Guide.doc**](http://qshare.ad.qintra.com/sites/MDW/User%20Documentation/MDW%20Designer%20User%20Guide.doc)

**Launch the Local Cloud Project wizard:**

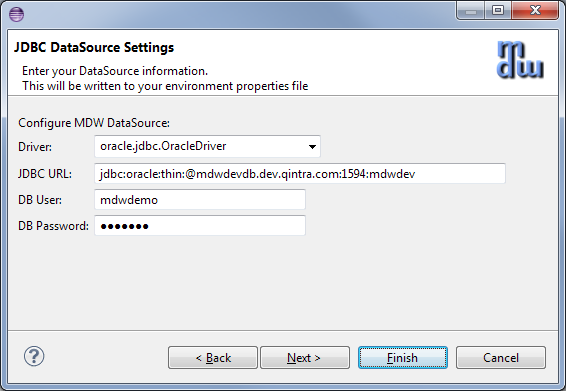
* Right-click inside the blank Process Explorer view and select New > Local Cloud Project. Select Apache Tomcat as your Java container. For Asset Persistence choose VCS (MDW will create a local Git repo where it'll store the workflow metadata for your assets).



* Click Next. Enter information about your Tomcat installation. If you don't know what your Tomcat User password is, enter "tomcat".



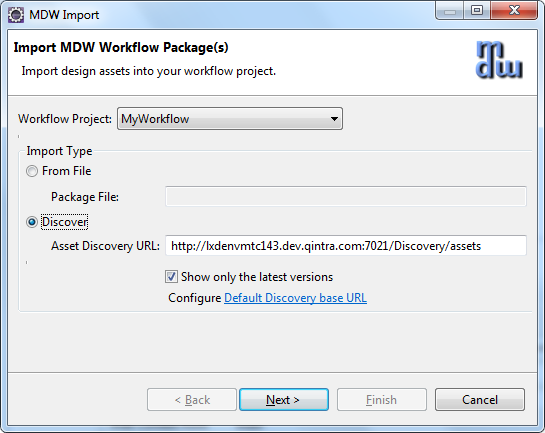
* Click Next again and enter your database connection info. The password for the default mdwdemo database is "mdwdemo".



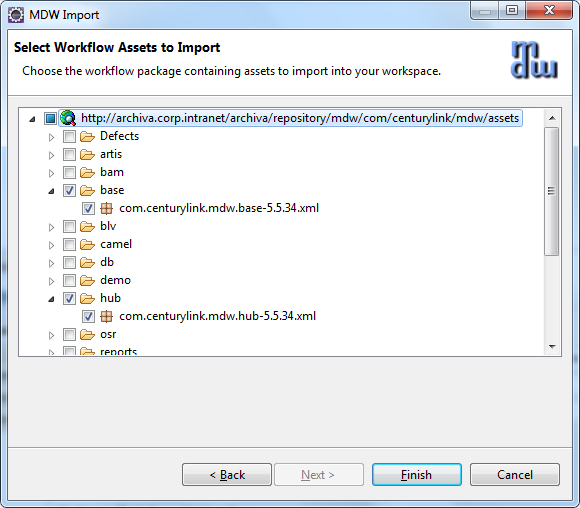
* Click Finish to generate your local cloud project and download MDW into your Tomcat webapps directory.

**The MDW Base Package:**

* The design artifacts in MDW are known as workflow assets. When you create processes or other assets these are organized into workflow packages, which are different from Java packages in that they can contain assets in a wide variety of formats. Much of the MDW framework's core functionality is itself delivered as workflow assets. The essential assets required by MDW are included in the packages "com.centurylink.mdw.base" and "com.centurylink.mdw.hub". The first step in setting up your workspace is to import these packages locally from the MDW repository.
* Expand your newly-created workflow project in Process Explorer and you'll see that it currently contains no workflow packages. Right-click on the project and select Import > Package. Choose the "Discover" option and leave the repository location as the default.



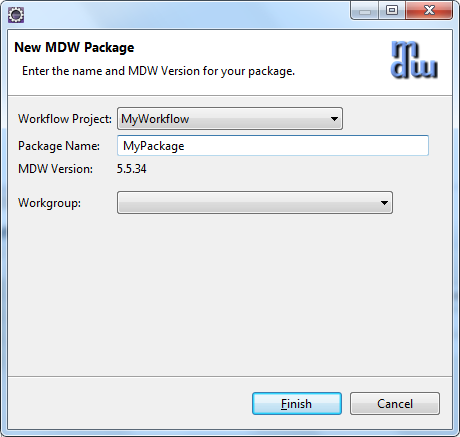
* After you click Next it'll take a few moments for Designer to locate the available packages. Once these are displayed, expand both the base package and the hub package and select the same MDW version as you did when creating the project.



* Click Finish, and the packages will be downloaded and become visible in your Process Explorer project tree.

**Create a Workflow Package:**

* The top-level branches in the project tree represent workflow packages. Your work should be incorporated in a dedicated package, which will be used for managing resources and for insulating your work from that of other users. For further details refer to the Eclipse Cheat Sheet (Help > Cheat Sheets > MDW Workflow > Importing, Exporting and Versioning).
* Create your workflow package by right-clicking on your project and selecting New > MDW Package. Note: make sure your package name complies with Java package naming requirements (eg: no spaces) since it will contain dynamic Java resources. Leave the Workgroup dropdown blank. If you don’t see MDW Package when you right click your project, click File > New > MDW Package from the top menu bar.



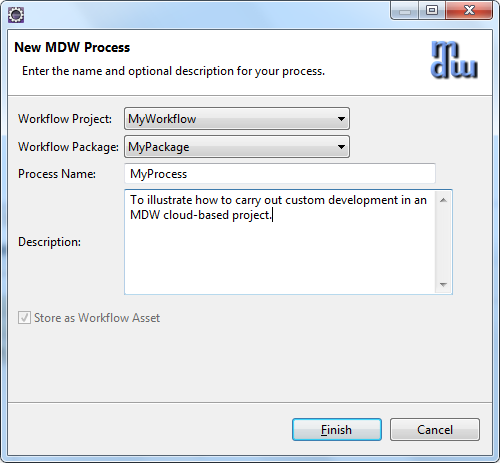
**Local Cloud Development**

[Previous](#local_1) [Contents](#contents) [Next](#local_3)

**2. Build a Workflow Process**

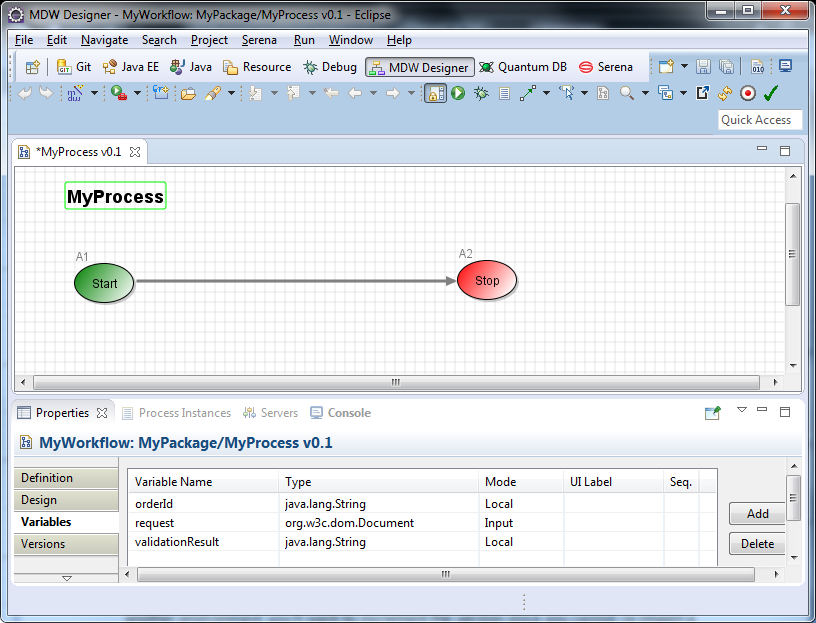
**Create the Process:**

* Right-click on your new package in Process Explorer and select New > MDW Process. Enter the process name and description and click Finish. If you don’t see MDW Process when you right click your new package, click File > New > MDW Process from the top menu bar.

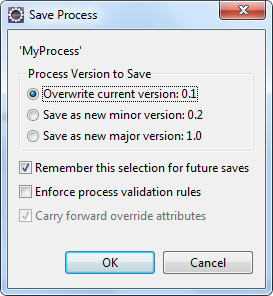


**Add some Process Variables:**

* Double-click on the process title or on a blank area somewhere in the canvas to display the Properties View. Select the Variables property tab and add an input variable (request) and two local variables (orderId and validationResult) with types as depicted below.

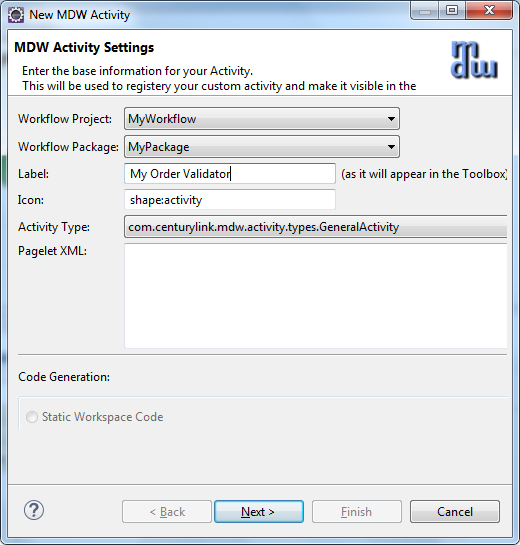
****

* Save your process design by selecting File > Save from the menu (or by clicking the disk icon in the Eclipse toolbar, or by typing ctrl-s). Elect to overwrite the current version and to keep the process locked after saving. During iterative development for convenience you'll sometimes overwrite the existing version of a process definition. However once you've exported to another environment you'll want to increment the version since you cannot re-import a changed process with the same version number. Details are covered under Help > Cheat Sheets > MDW Workflow > Importing, Exporting and Versioning.

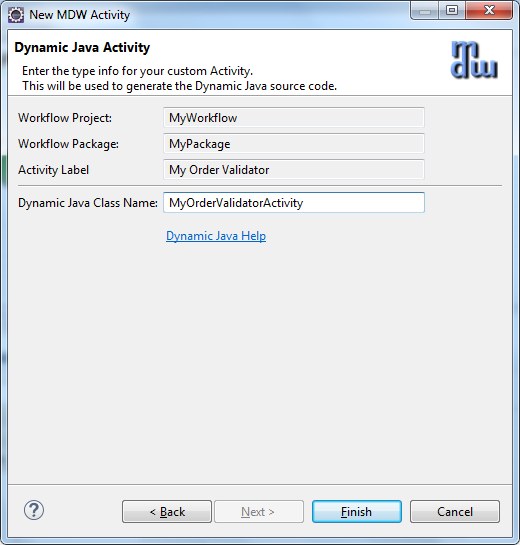


**Create a Dynamic Java Custom Activity:**

* Right-click on your package in Process Explorer and select New > Activity > General Activity. On the first page of the wizard, enter a label to identify your activity in the Toolbox view.

****

* Click Next and enter a class name for your activity implementor. The Java package name is the same as your workflow package name.



* When you click Finish the Java code for a skeleton implementation is generated. You'll also see the Java class under your package in Process Explorer. This source code resides under src/main/workflow and is known as a Dynamic Java workflow asset. It's dynamic because it can be changed without needing any kind of application deployment. Naturally there are rigorous controls in place to prevent unauthorized modifications. In step 1 you were granted permissions in the MDW Demo environment to create and modify workflow assets. With Dynamic Java, as with all types of workflow assets, MDW provides facilities for versioning, rollback and import/export for migrating between environments.
* Update the generated Java source code to resemble the following:

**package** MyPackage;

**import** com.centurylink.mdw.common.utilities.logger.StandardLogger.LogLevel;

**import** com.centurylink.mdw.common.utilities.timer.Tracked;

**import** org.w3c.dom.Document;

**import** org.w3c.dom.Node;

**import** com.centurylink.mdw.activity.ActivityException;

**import** com.centurylink.mdw.model.value.activity.ActivityRuntimeContext;

**import** com.centurylink.mdw.workflow.activity.DefaultActivityImpl;

/\*\*

\* MDW general activity.

\*/

@Tracked(LogLevel.***TRACE***)

**public** **class** MyOrderValidatorActivity **extends** DefaultActivityImpl {

/\*\*

\* Here's where the main processing for the activity is performed.

\*

\* **@return** the activity result (aka completion code)

\*/

@Override

**public** Object execute(ActivityRuntimeContext runtimeContext) **throws** ActivityException {

loginfo("Validating order...");

Document request = (Document) getVariableValue("request");

Node orderIdNode = request.getFirstChild().getFirstChild().getNextSibling();

String orderId = orderIdNode.getFirstChild().getNodeValue();

setVariableValue("orderId", orderId);

**boolean** valid = **true**;

String msg = "Success";

**if**(!orderIdNode.getLocalName().equals("orderId")){

msg = "Missing order ID.";

}

valid = msg.equals("Success");

setVariableValue("validationResult", msg);

**return** valid;

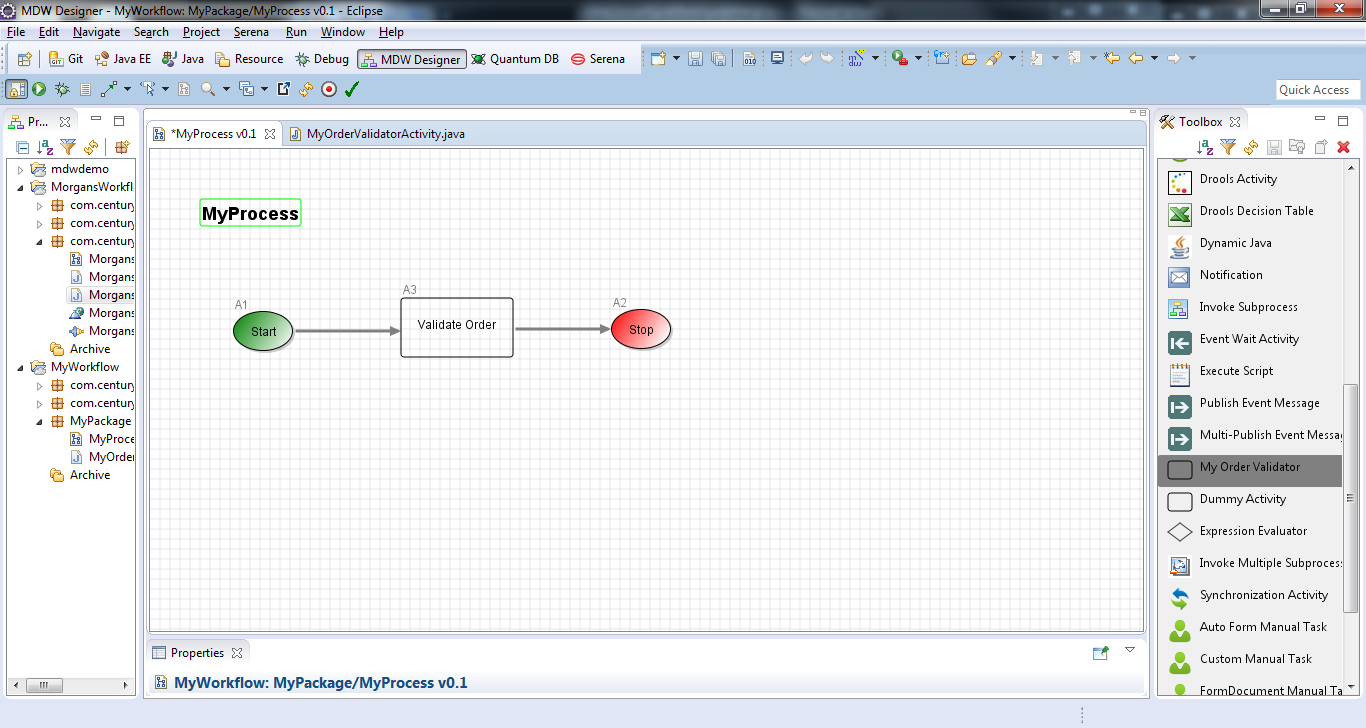
}

}

* Now if you switch back to your process the new activity should appear in the Toolbox View. From the toolbox, drag your activity onto the canvas and insert it into your process flow between the Start and Stop activities.

Tip: To draw a link (or *transition* in MDW terminology) between activities on the designer canvas, hold down the Shift key on your keyboard, Click on the upstream activity, and continue holding down the mouse left click button while dragging the cursor to the downstream activity ("*shift-click-drag*")*.*

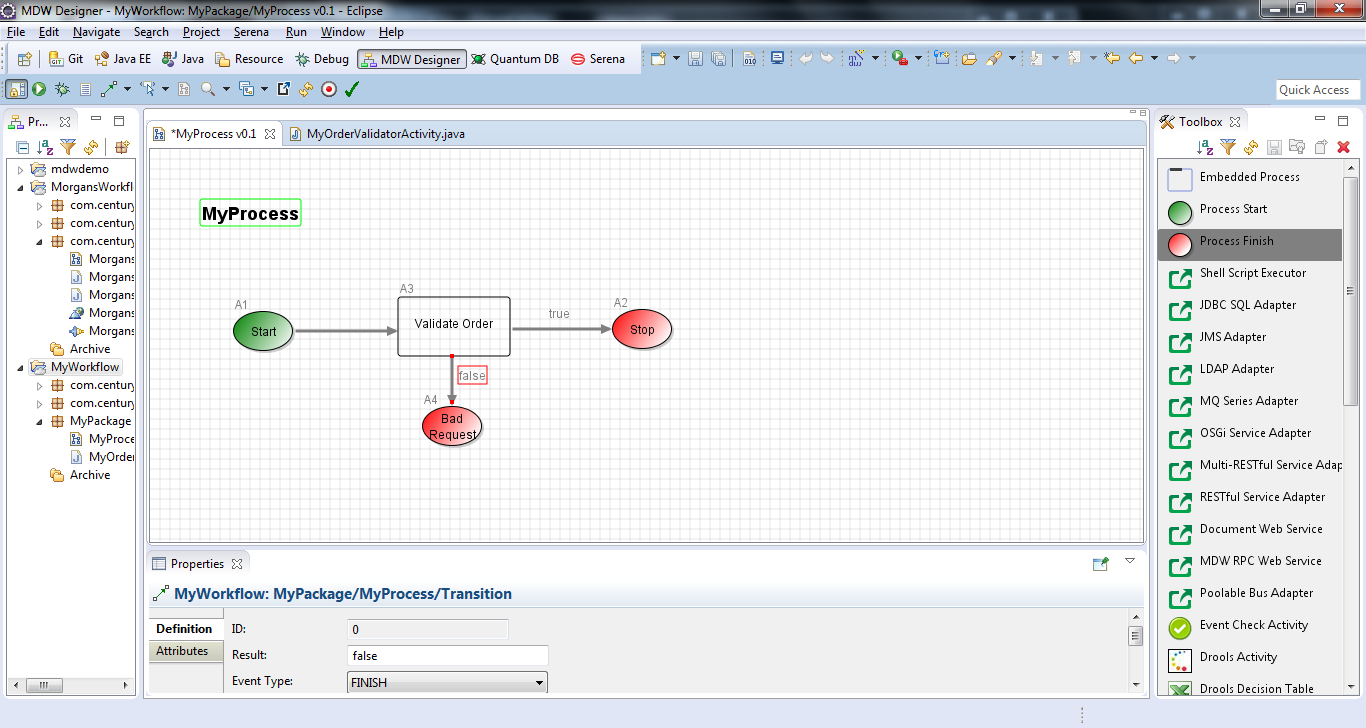
* Your activity can be dragged like this and used in other processes designed by other users. Actually the proper term in MDW for this reusable element in the Toolbox is *activity implementor*. This conveys the idea that it's actually a template to be dragged and configured as an *activity* in the canvas, and it also conveys the fact that it always corresponds to a Java class. To take this reuse concept a step further, your activity implementor can be made discoverable so that it can easily be imported into other environments and reused across domains. If you click on the light bulb icon at the top of the Toolbox you'll get an idea how items in the palette can be imported from a file or discovered in the corporate repository.
* Double click the activity in the canvas, and in its Definition property tab change the label to something like "Validate Order". When you click back on the canvas the activity reflects its new label.

****

* Note: If you select the Design property tab for your activity you'll see that it's blank. A non-trivial activity would allow certain aspects (such as endpoint URLs) to be configurable, so that it could readily be reused. For example, take a look at the Design tab for the Start activity. You control what appears on the Design tab through the pagelet XML for the activity implementor. In the creation wizard we left the pagelet XML blank, so the Design tab for our activity is empty. But to continue with the example of the start activity, find the Process Start icon in the Toolbox and view its Design tab (for the implementor, not the activity on the canvas). This gives you an idea of how the pagelet XML relates to the fields on the Design tab for the activity user. Since we're on the subject you may be interested to know how you can customize the icon for your activity implementor. On the Definition tab you can choose one of the built-in shapes, or more flexibly choose any GIF, JPG or PNG asset that you can easily add to your workflow package.

**Add Multiple Activity Outcomes:**

* Drag a Process Finish activity from the Toolbox, and add another outbound transition from "Validate Order". Assign Result Code values of "true" and "false" to the respective transitions as illustrated below. Save your process definition. The value passed in setReturnCode() in your activity's execute() method dictates which of these two paths will be traversed at runtime.



**Local Cloud Development**

[Previous](#local_2) [Contents](#contents) [Next](#local_4)

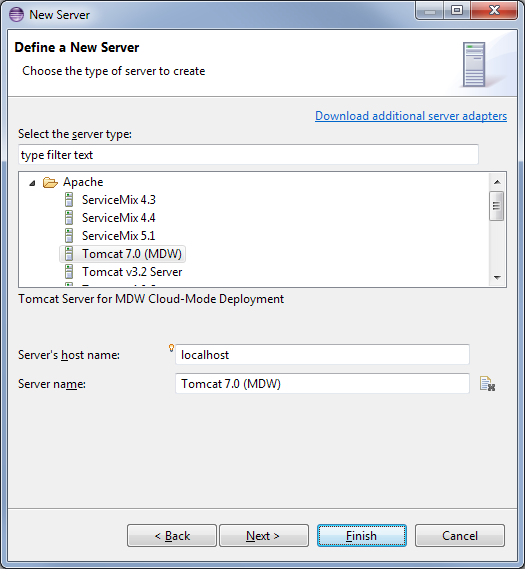
**3. Running Your Process**

**Ensure Permissions:**

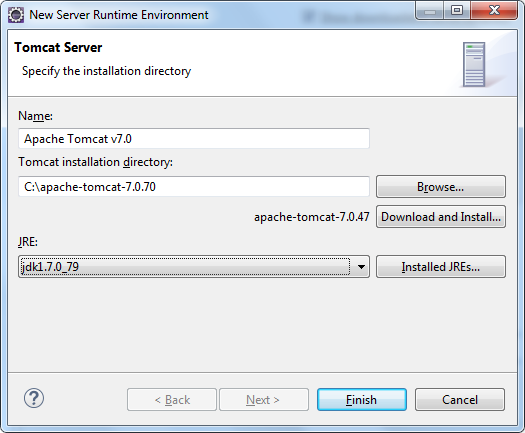
* To follow the remaining steps in this tutorial you need to be granted the appropriate roles in the MDW demo database (unless you've installed a database locally). An administrator can grant you appropriate access using the MDWHub webapp. For a detailed discussion of this topic, refer to the "Roles and Permissions" section of the [MDW Designer User Guide](http://qshare.ad.qintra.com/sites/MDW/User%20Documentation/MDW%20Designer%20User%20Guide.doc).

**Create a Tomcat Server:**

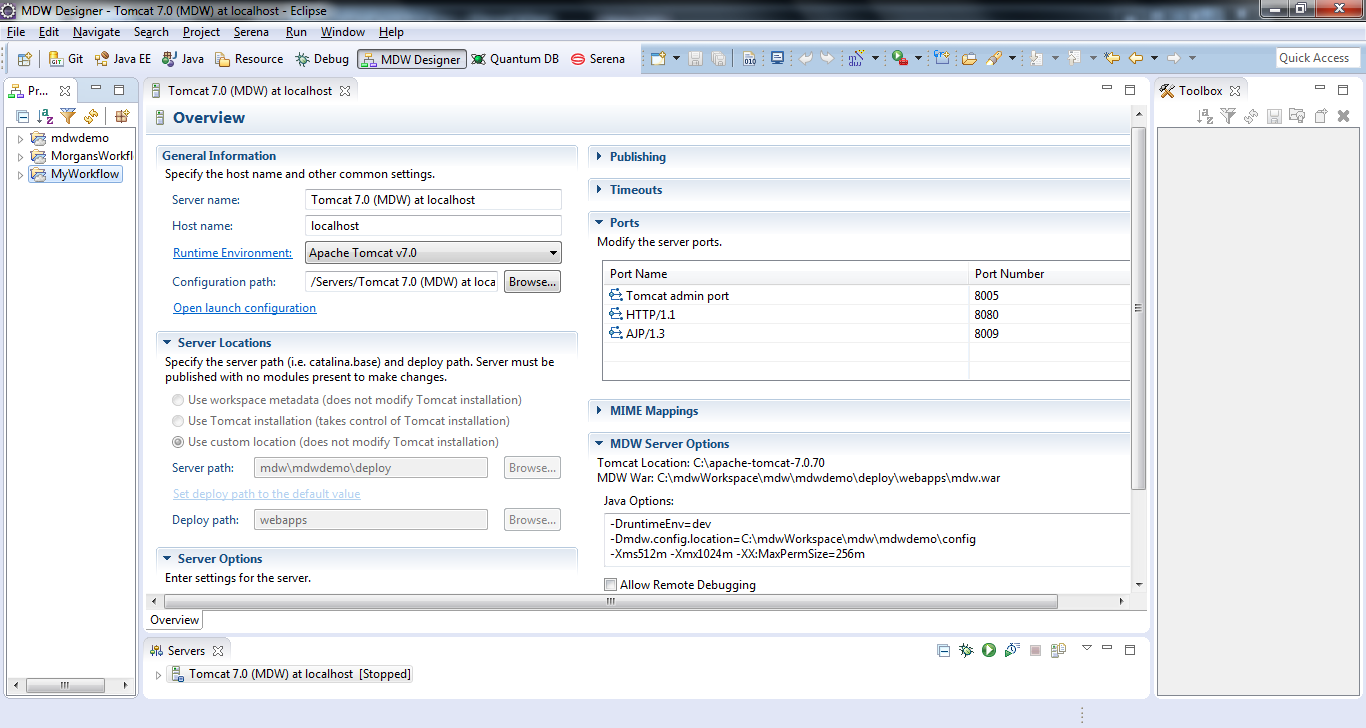
* To execute a workflow process you need a server running with MDW deployed. For debugging in Eclipse the easiest way to set this up is through a Web Tools Platform server instance. From the menu select File > New > Other > Server > Server. Click Next and select Apache > Tomcat 7.0 (MDW) from the options. To be able to debug your Dynamic Java it's important that you select ***Tomcat 7.0 (MDW)*** instead of the standard ***Tomcat 7.0 Server***. The server name is arbitrary, so you can make is something friendlier than the default.



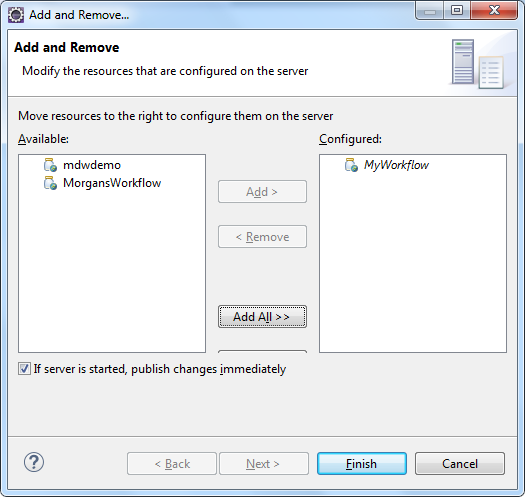
* If you've not previously used a Tomcat 7 runtime in Eclipse, clicking Add takes you to a page where you specify your Tomcat location. Make sure to select a JDK to compile the code and that the selected JDK is Java 1.7.x



* The final page of the New Server wizard is where you designate your workflow project to be deployed on the server. After that, click Finish to create the server instance.
* If the Servers view is not visible in your current perspective, from the menu select Window > Show View > Other > Server > Servers. You should see your Tomcat 7 server in this view. You can double-click the server to edit its configuration. Expand the Timeouts section, change the start timeout value to 300 seconds, and hit Ctrl-S to save your changes. Then close the editor.



* Before you start the server, you will need to add your MyWorkflow project to your server instance. Right click the server instance, select Add and Remove… and select the MyWorkflow from the left pane and click the Add to move it to the right pane.



**Run Tomcat:**

* Now that you've created the WTP server instance, the Servers view gives you a handy way to start and stop Tomcat. And output is directed to the Eclipse Console view, where you can click on stack traces to open associated source code (including MDW Framework code and Dynamic Java). Start your server in debug mode by right-clicking on it and selecting Debug (or use the icon in the Servers view toolbar).
* The first time you start your server Tomcat explodes the mdw.war file in your deploy/webapps directory and caches the deployable content. This can sometimes take a minute. With the server running you should see MDW output in the Eclipse Console view. You can safely ignore any Dynamic Java compilation errors unless they pertain to the custom activity you created in Step 2.

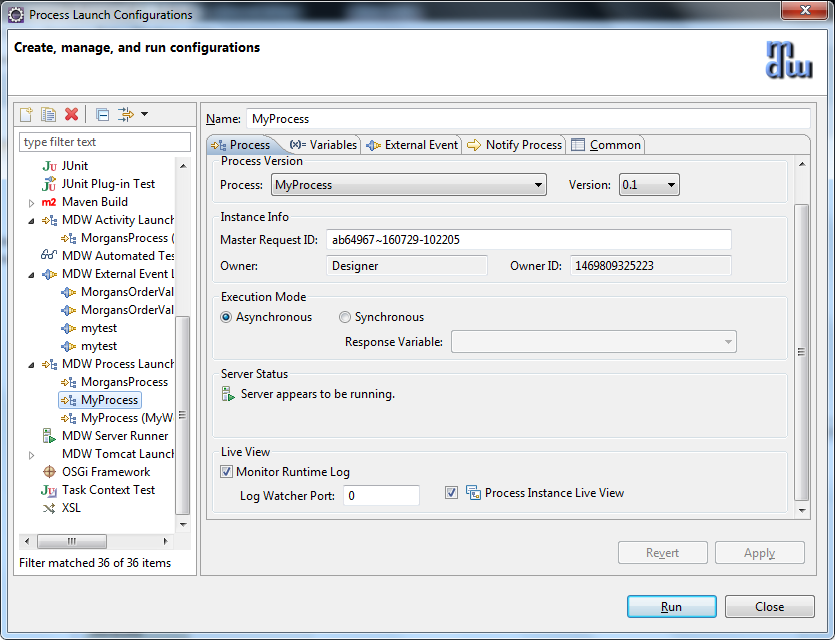
Tip: When you upgrade to a new MDW build version in Eclipse, Designer automatically downloads the corresponding mdw.war file into your deploy/webapps directory. If at any time you want to clean out the MDW deployment and start fresh, you can delete mdw.war and the exploded mdw directory (and for a very thorough cleansing you can even delete the Tomcat cache under deploy/work/Catalina/localhost/mdw). Then you can deploy from scratch from Package Explorer view by right-clicking on your workflow project and selecting MDW Update > Update Framework Libraries.

* Make sure your project is added to your Java Build Path/Source. You will need to do this from a Java or Resource perspective.
* You can confirm that MDW was successfully deployed by accessing MDWHub in your browser:

<http://localhost:8080/mdw>

**Open the Process Launch Dialog:**

* Right-click on your process that is under your workflow package in Process Explorer view and Select Run. You can also right-click on your Designer and Select Run. Designer will present the launch dialog and open a connection to the server to confirm that it's running (required for launching a process).
* On the Process tab in the launch dialog, select "Monitor Runtime Log" and "Process Instance Live View" to get a feel for how you can watch your process flow in real time.



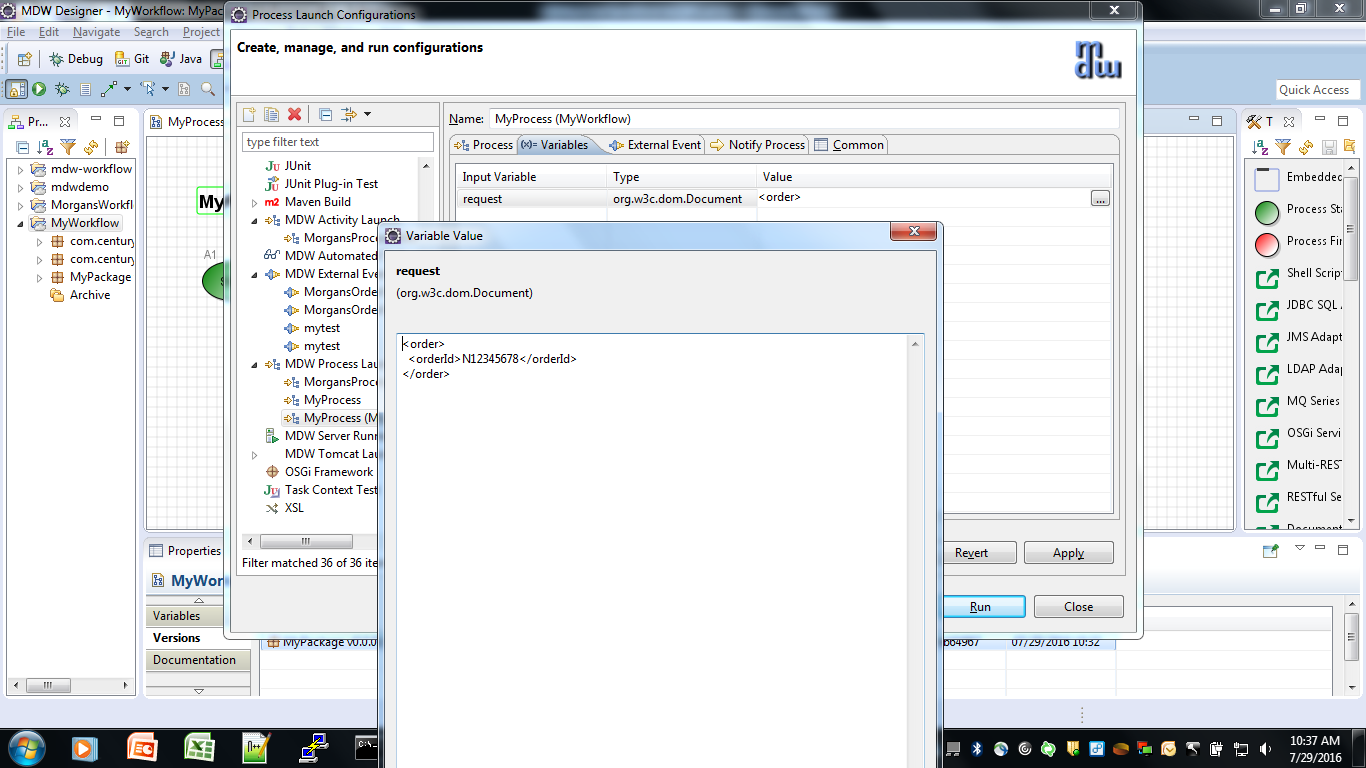
**Populate the Input Variable:**

<order>

<orderId>N12345678</orderId>

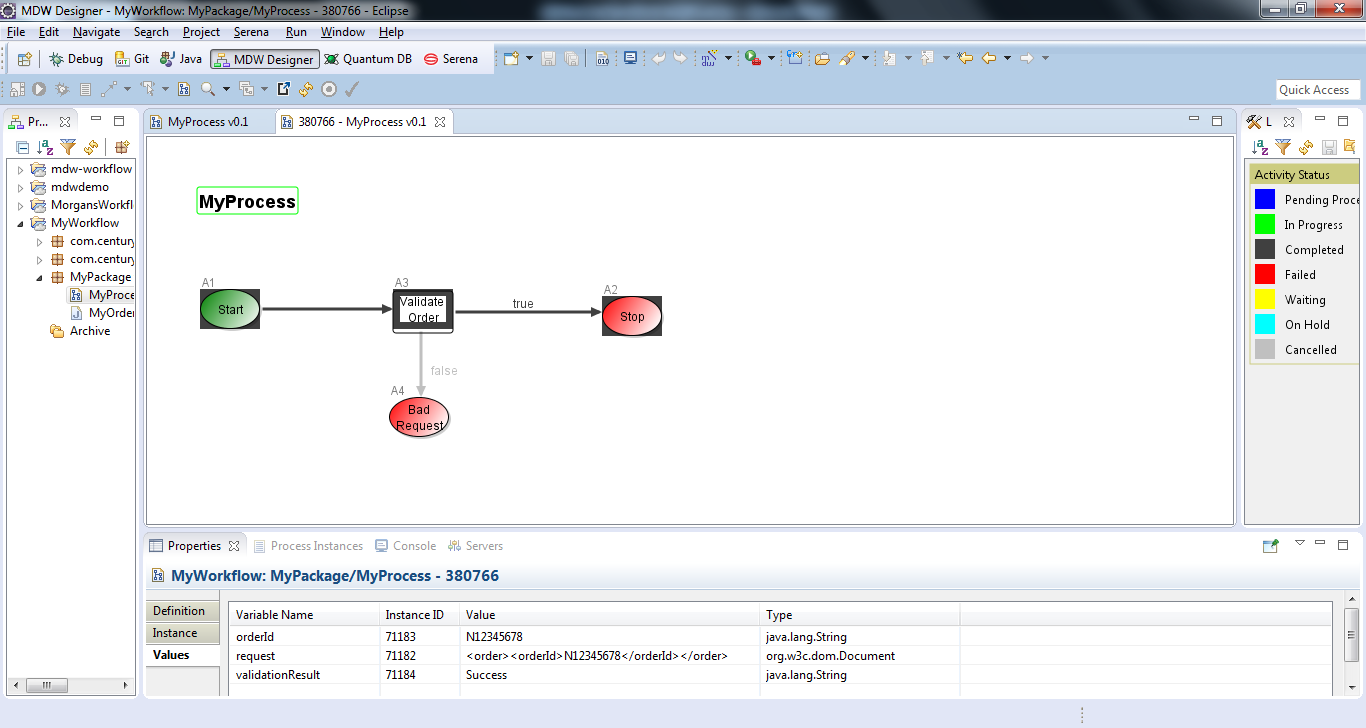
</order>

* Select the Variables tab in the launch dialog, and populate the request variable with the following content.



**Launch and View an Instance:**

* Click Run on the launch dialog to run an instance of your process. In the Live View you should see the new instance progress down the happy path with the Validate Order outcome equal to 'true'. For processes not displayed in Live View, you can open an instance manually by right-clicking on your process in Process Explorer view and selecting View Instances. The latest instance will appear at the top of the Process Instances list, and you can double-click to open its runtime view.
* In Designer Perspective when a process instance is visible, a legend appears showing what the borders surrounding the activities mean. To inspect the runtime variable values for the instance, click the Values property tab.



**Change Java Code and Rerun with a Breakpoint:**

* Change the Java source so that validation expects an order number that begins with a digit:

**if** (!orderIdNode.getLocalName().equals("orderId"))

msg = "Missing order ID.";

**else** **if** (!Character.*isDigit*(orderId.charAt(0)))

msg = "Order ID must begin with a digit.";

* Save your changes and run your process again to confirm that this time it fails validation with the appropriate validationResult message. Note: In the real world Order IDs would likely be unique for each request, so you may want to change the XML input on the process launch Variables tab to something other than the value remembered from the last launch.
* Let's pretend like we don't know why validation is failing, so we'd like to debug our Dynamic Java source code. Set a breakpoint on the line with the *if* condition by double-clicking on the marker bar on the left side of Eclipse's Java editor.
* Run your process again, but this time uncheck "Monitor Runtime Log" on the Process tab in the launch dialog so that Live View doesn't steal focus while you're debugging. After clicking Run, switch to the Debug perspective in Eclipse by selecting Window > Open Perspective > Debug. When process flow reaches your validator activity, you should see the usual green highlighting in the editor. Here you can step through the code and evaluate variables in the usual way as described in [this section](http://help.eclipse.org/luna/topic/org.eclipse.jdt.doc.user/tasks/task-stepping.htm?cp=1_3_6_1_4) of the online Eclipse help docs.
* When you're done debugging, continue execution to let the process complete. You can view the new instance by right-clicking on the process in Process Explorer and selecting View Instances. Double-click on the top instance row to confirm that this second instance took the Bad Request path. Make sure that your tomcat server is up to view the Process Instances.

**Local Cloud Development**

[Previous](#local_3) [Contents](#contents) [Next](#local_5)

**4. Consume a Web Service**

MDW comes with the Document Web Service Activity for consuming document-style services hosted by external providers. In this exercise we'll invoke the GetEmployee service hosted in the MDW Demo environment (and this service itself is implemented as an MDW workflow process; the sections ahead describe how to create and expose a service process).

**Use the Document Web Service Activity:**

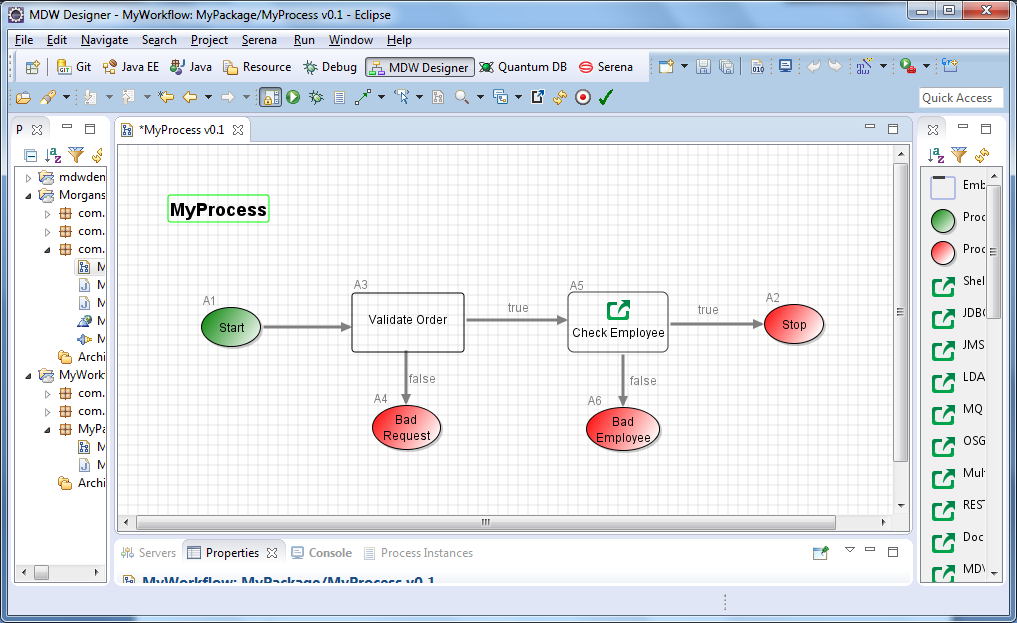
* Open the same process definition you started building in the sections above. Add another String variable called employeeId. Edit the code in your order validation activity to set employeeId from the request:

String employeeId = orderIdNode.getNextSibling().getNextSibling().getFirstChild().

getNodeValue();

setVariableValue("employeeId", employeeId);

* Drag the Document Web Service activity onto the design canvas and insert it downstream of Perform Validation. Label the web service activity "Check Employee", and give it two separate outcomes corresponding to true and false, just like the validation activity.



On the Design tab of the service, set the Endpoint/WSDL URL to: <http://lxdenvmtc143.dev.qintra.com:8989/mdw/Services/SOAP>

.

**Add Pre and Post Script:**

* To customize the behavior of the Document Web Service activity, you could extend the framework class in a custom Dynamic Java activity as we did for the validator. However, for service invocation activities (known in MDW as *adapter* activities), you can also associate script to be executed before and after the service call. This can sometimes be a quick alternative to creating your own custom activity. Double-click on the Check Employee activity and select the Script property tab. Edit the prescript, adding the Groovy code below to return a request that includes employeeId (notice that in your script you can refer to variables directly by their name):

**return** '''<GetEmployee>

<sapId>''' + employeeId + '''</sapId>

</GetEmployee>''';

* Add new process variables – click the Process on the canvas and click Variables tab that is on the left navigation under the Design tab - to hold the service request (name=employeeServiceRequest, type=com.centurylink.mdw.model.StringDocument) and response (name=employeeServiceResponse, type=org.w3c.dom.Document). On the Design tab of the Check Employee web service activity, select these new variables in the Request Variable and Response Variable dropdowns respectively. Edit the postscript as follows (if you've installed the Groovy plugin you'll get syntax highlighting and autocomplete):

**import** org.w3c.dom.Node;

**import** org.w3c.dom.NodeList;

NodeList nodes = employeeServiceResponse.getFirstChild().getChildNodes();

String firstName = **null**;

String lastName = **null**;

**for** (**int** i = 0; i < nodes.getLength(); i++) {

Node node = nodes.item(i);

**if** ("firstName".equals(node.getLocalName()))

firstName = node.getFirstChild().getNodeValue();

**else** **if** ("lastName".equals(node.getLocalName()))

lastName = node.getFirstChild().getNodeValue();

}

**if** (firstName != **null** && lastName != **null**) {

runtimeContext.logInfo 'Found employee: ' + firstName + ' ' + lastName;

**return** **true**;

}

**else** {

runtimeContext.logInfo 'Employee not found: ' + employeeId;`

validationResult = 'Employee not found: ' + employeeId;

**return** **false**;

}

**Save and Run Your Process:**

* Save the modified process. When prompted, elect to "Save as new minor version". Whenever a process design that has runtime instances is changed structurally (new activities or transitions), it is highly recommended that you increment the version number so that Designer can correctly display runtime data in the process instance view.
* Right-click on a blank spot in the designer canvas and select Run to open the launch configuration dialog. On the Variables tab change the value for orderDoc to include a valid orderNumber (with a digit as the first character), and also your CenturyLink SAP ID for the employeeId. You can also use your workstationId (CUID) in place of the employeeId but make sure to change the code that references the employeeId to workstationId. Click Ok to close it and click Run on the configuration dialog to run it.

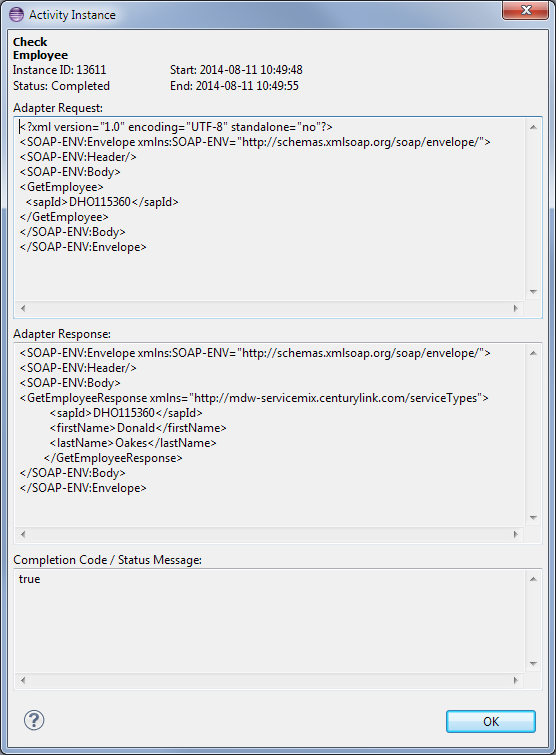
<order>

<orderId>0123456</orderId>

<employeeId>DHO115360</employeeId>

</order>

* Right-click again on a blank spot and select View Instances. Double-click the instance to open it. It should reflect that the service was invoked, your SAP ID was found, and the "true" outcome from Check Employee should be traversed. Double-click on the Check Employee activity in the process instance. On the Instance property tab, double-click on the instance row to display the raw SOAP request and response:



**Local Cloud Development**

[Previous](#local_4) [Contents](#contents) [Next](#advanced_1)

**5. Expose Your Process to External Systems**

**Designate Your Process as an MDW Service Process:**

* On the Design property tab for your process check the box labeled "Service Process". In MDW terminology this designates your process as one that runs synchronously and is able to generate a response in real time. For more background on Service Processes, click on the context help link "Process Configuration Help".
* Add a new process variable called "response" of type org.w3c.dom.Document. This is where you'll populate the output from your Order Validation service. Note: by default the "request" and "response" variable names are reserved and are automatically bound to the incoming and outgoing payload of your service.

**Create an Activity to Generate a Response:**

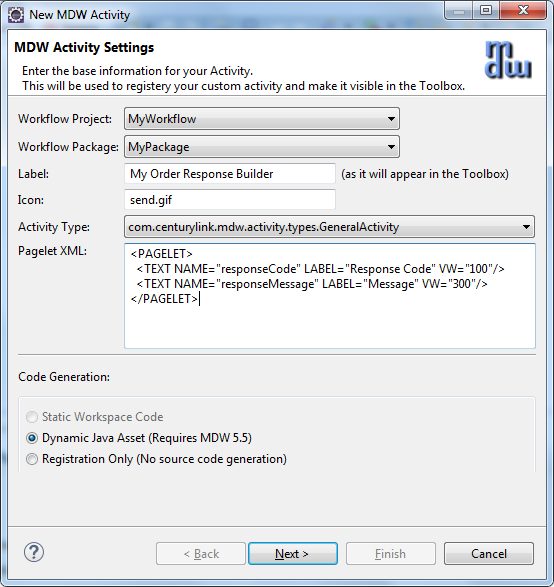
* Right-click on your package in Process Explorer and select New > Activity > General Activity. For this one use the icon send.gif, which is included in the MDW baseline workflow package, or you can use your own 24x24 pixel image (right-click on your package and select New > Web Resource > Binary > GIF > Browse for file). Also, this time since you're creating an activity to use in multiple places, add the following pagelet definition for configurable attributes.

<PAGELET>

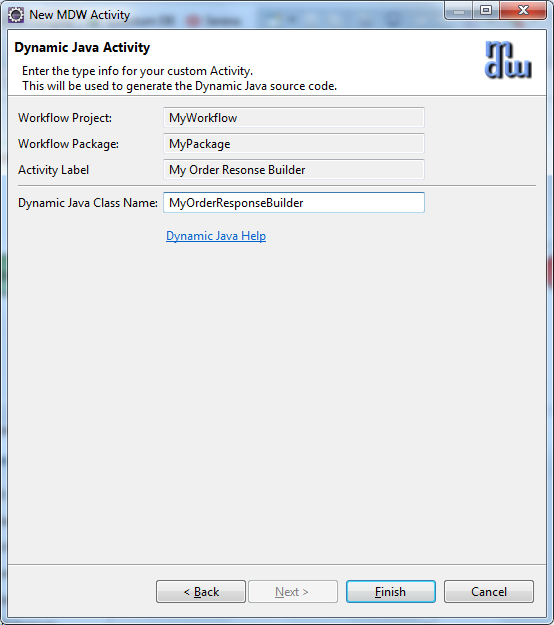
<TEXT NAME="responseCode" LABEL="Response Code" VW="100"/>

<TEXT NAME="responseMessage" LABEL="Message" VW="300"/>

</PAGELET>



* Click Next. Give a name for the class and click Finish.



* Make your activity implementor source code look something like this:

**package** MyPackage;

**import** com.centurylink.mdw.activity.ActivityException;

**import** com.centurylink.mdw.common.utilities.logger.StandardLogger.LogLevel;

**import** com.centurylink.mdw.common.utilities.timer.Tracked;

**import** com.centurylink.mdw.model.value.activity.ActivityRuntimeContext;

**import** com.centurylink.mdw.workflow.activity.DefaultActivityImpl;

**import** com.centurylink.mdw.xml.DomHelper;

@Tracked(LogLevel.*TRACE*)

**public** **class** MyOrderResponseBuilder **extends** DefaultActivityImpl {

@Override

**public** Object execute(ActivityRuntimeContext runtimeContext) **throws** ActivityException {

**try** {

String code = getAttributeValueSmart("responseCode");

**if** (code == **null**)

**throw** **new** ActivityException("Missing attribute: responseCode");

String resString = "<OrderValidationResponse orderId=\""

+ getVariableValue("orderId") + "\">\n"

+ " <Code>" + code + "</Code>\n";

**if** (!code.equals("0"))

resString += " <Message>" + getAttributeValueSmart("responseMessage")   
 + "</Message>\n";

resString += "</OrderValidationResponse>";

setVariableValue("response", DomHelper.*toDomDocument*(resString));

}

**catch** (Exception ex) {

**throw** **new** ActivityException(ex.getMessage(), ex);

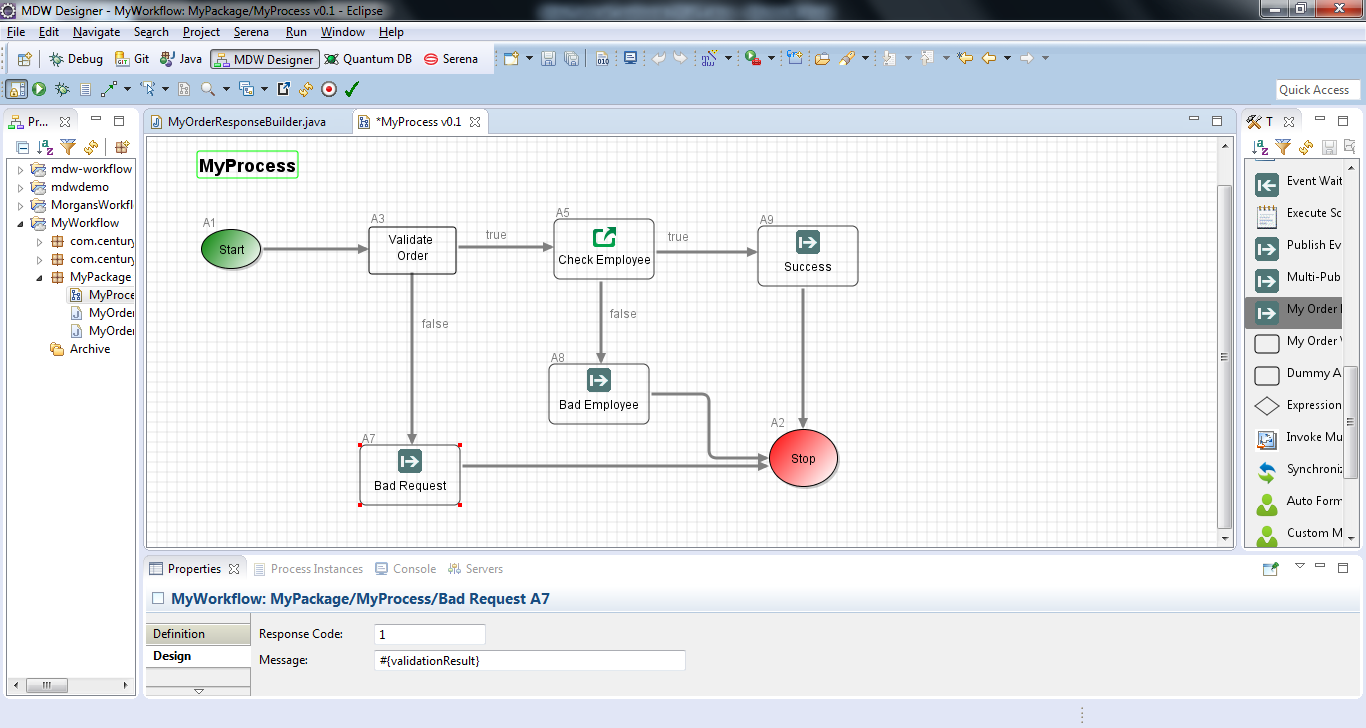
}

**return** **null**;

}

}

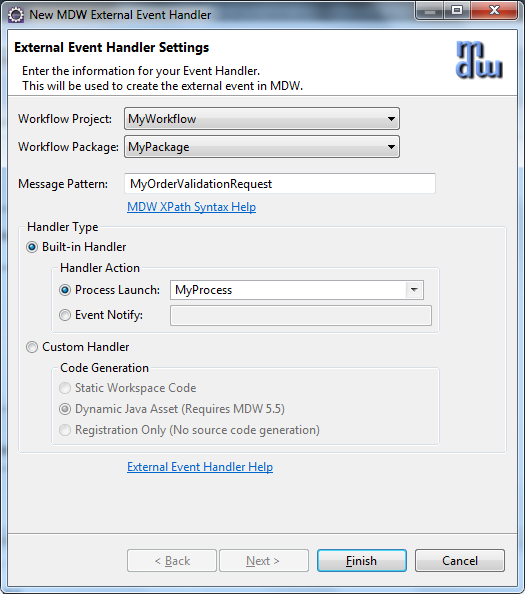
* Now rework your process so that the three possible outcomes all generate a response using this activity. For the two error paths set the Response Code to be non-zero in the Design tab. Since the validation result is kept in a process variable, you can use a [Java Expression](http://docs.oracle.com/javaee/6/tutorial/doc/gjddd.html) as illustrated below to parameterize the Message attribute value.



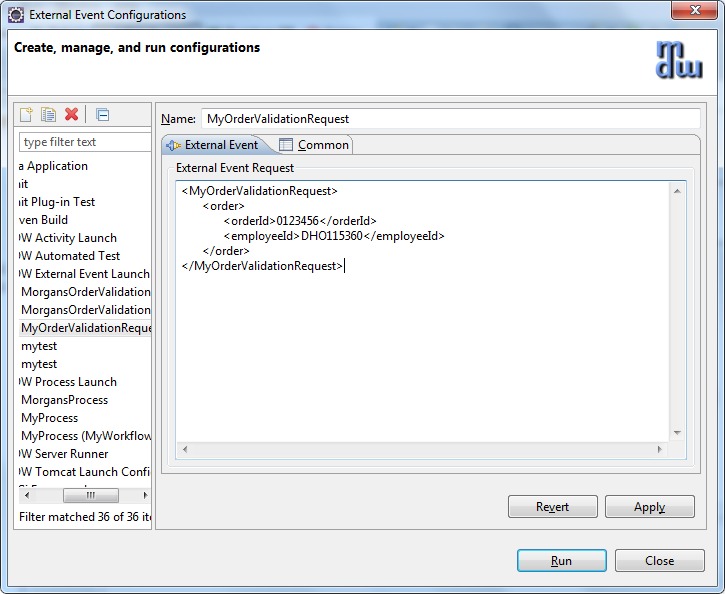
* You can still test your process execution by launching in Designer like we've done previously. However, for your process to be available as a service for consumption by external systems, you'll need to register an event handler.

**Create an External Event Handler:**

* MDW services are registered using a protocol-neutral mechanism based on the request document content. This registration is what we refer to as an External Event Handler. Once you register an event handler, then by default it's exposed over all the available transport channels that MDW supports (SOAP, REST, JMS, etc). For more information about this mechanism, refer to Help Topic: MDW Designer Help > Coding and Development > Listeners and External Event Handlers.
* Right-click on your workflow package in Process Explorer view and select New > Event Handler > External Event Handler. For the Handler Action select the service process you created in the previous step. For the Message Pattern enter an XPath expression that matches your request document. In effect, you're telling MDW to launch your service process whenever it receives a request whose document content matches the configured message pattern. Note that if a request is received that matches multiple registered event handlers, then it is undefined which of those handlers will be invoked. For this reason it's imperative that you make the XPath expression unique so that it does not match requests it is not intended to handle (and thereby hijack those requests from their intended handler).



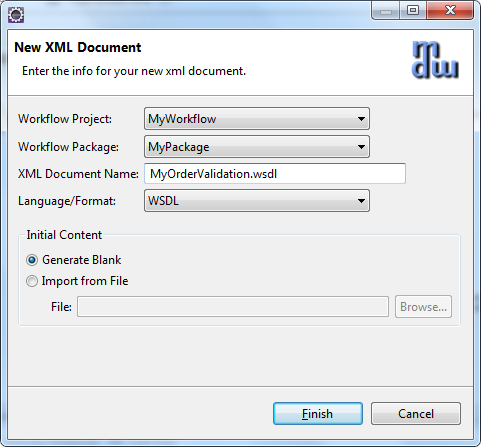
* In Process Explorer view your event handler appears labeled as its associated message pattern. You can test it by right-clicking on it and selecting Run, and then filling in the External Event Request in the launch dialog. When you run this way Designer submits the request document over HTTP to the MDW RESTful listener, so it not only tests your process flow but it also tests your Event Handler registration as well.



* After successfully invoking your External Event Handler, you should see the expected response in the console view, and you should also find that a new instance of your service process was created.

**Implement a SOAP Web Service:**

* The easiest way to expose your process as a SOAP service is to create a document-style WSDL workflow asset that describes it. In Process Explorer view, right-click on your workflow package and select New > XML Document. Name it something appropriate for your service, and select the language/format as WSDL.



* Edit the content of your WSDL to look something like the following (with appropriate substitutions based on your request and response).

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

<wsdl:definitions

name=*"wsdl-first"*

xmlns:wsdl=*"http://schemas.xmlsoap.org/wsdl/"*

xmlns:soap=*"http://schemas.xmlsoap.org/wsdl/soap/"*

xmlns:xsd=*"http://www.w3.org/2001/XMLSchema"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:tns=*"http://mdw-servicemix.centurylink.com"*

targetNamespace=*"http://mdw-servicemix.centurylink.com"*>

<wsdl:types>

<xsd:schema>

<xsd:element name=*"MyOrderValidationRequest"*>

<xsd:complexType>

<xsd:sequence>

<xsd:element

name=*"orderId"*

type=*"xsd:string"* />

<xsd:element

name=*"employeeId"*

type=*"xsd:string"* />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:element name=*"OrderValidationResponse"*>

<xsd:complexType>

<xsd:sequence>

<xsd:element

name=*"Code"*

type=*"xsd:string"* />

<xsd:element

name=*"Message"*

type=*"xsd:string"*

minOccurs=*"0"* />

</xsd:sequence>

<xsd:attribute name=*"orderId"* type=*"xsd:string"* use=*"required"* />

</xsd:complexType>

</xsd:element>

</xsd:schema>

</wsdl:types>

<wsdl:message name=*"OrderValidationRequestMessage"*>

<wsdl:part

name=*"payload"*

element=*"MyOrderValidationRequest"* />

</wsdl:message>

<wsdl:message name=*"OrderValidationResponseMessage"*>

<wsdl:part

name=*"payload"*

element=*"OrderValidationResponse"* />

</wsdl:message>

<wsdl:portType name=*"ValidateOrder"*>

<wsdl:operation name=*"ValidateOrder"*>

<wsdl:input message=*"tns:OrderValidationRequestMessage"* />

<wsdl:output message=*"tns:OrderValidationResponseMessage"* />

</wsdl:operation>

</wsdl:portType>

<wsdl:binding

name=*"OrderValidationSOAPBinding"*

type=*"tns:ValidateOrder"*>

<soap:binding

style=*"document"*

transport=*"http://schemas.xmlsoap.org/soap/http"* />

<wsdl:operation name=*"ValidateOrder"*>

<wsdl:input>

<soap:body use=*"literal"* />

</wsdl:input>

<wsdl:output>

<soap:body use=*"literal"* />

</wsdl:output>

</wsdl:operation>

</wsdl:binding>

<wsdl:service name=*"OrderValidationService"*>

<wsdl:port

binding=*"tns:OrderValidationSOAPBinding"*

name=*"soap"*>

<soap:address location=*"${mdw.services.url}/SOAP/DonsPackage/DonsOrderValidation.wsdl"* />

</wsdl:port>

</wsdl:service>

</wsdl:definitions>

<wsdl:output>

<soap:body use=*"literal"* />

</wsdl:output>

</wsdl:operation>

</wsdl:binding>

<wsdl:service name=*"OrderValidationService"*>

<wsdl:port

binding=*"tns:OrderValidationSOAPBinding"*

name=*"soap"*>

<soap:address location=*"${mdw.services.url}/SOAP/MyPackage/ MyOrderValidation.wsdl"* />

</wsdl:port>

</wsdl:service>

</wsdl:definitions>

* Note that the endpoint URL is parameterized in the WSDL <soap:address> so at runtime it will be substituted with the appropriate base URL for the specific environment where it's served from
* Once you save the WSDL you should be able to access it in your browser from a location similar to:

<http://localhost:8080/mdw/SOAP/MyPackage/MyOrderValidation.wsdl>

**Invoke Your Service:**

* The MDWHub System tab includes a utility that you can use to test your service. Access MDWHub in your browser through a URL like this:

<http://lxdenvmtc143.dev.qintra.com:8989/mdw/Services/SOAP>

Click on the System tab and the HTTP Poster navigation link (if you don't see the System tab you'll need to be granted Site Admin permissions for the environment where you're testing). The submittal URL for HTTP Poster defaults to the MDW REST endpoint, so change the context root from REST to SOAP as illustrated in the screenshot below. Populate the Message Body with something like the following:

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">

<soapenv:Header/>

<soapenv:Body>

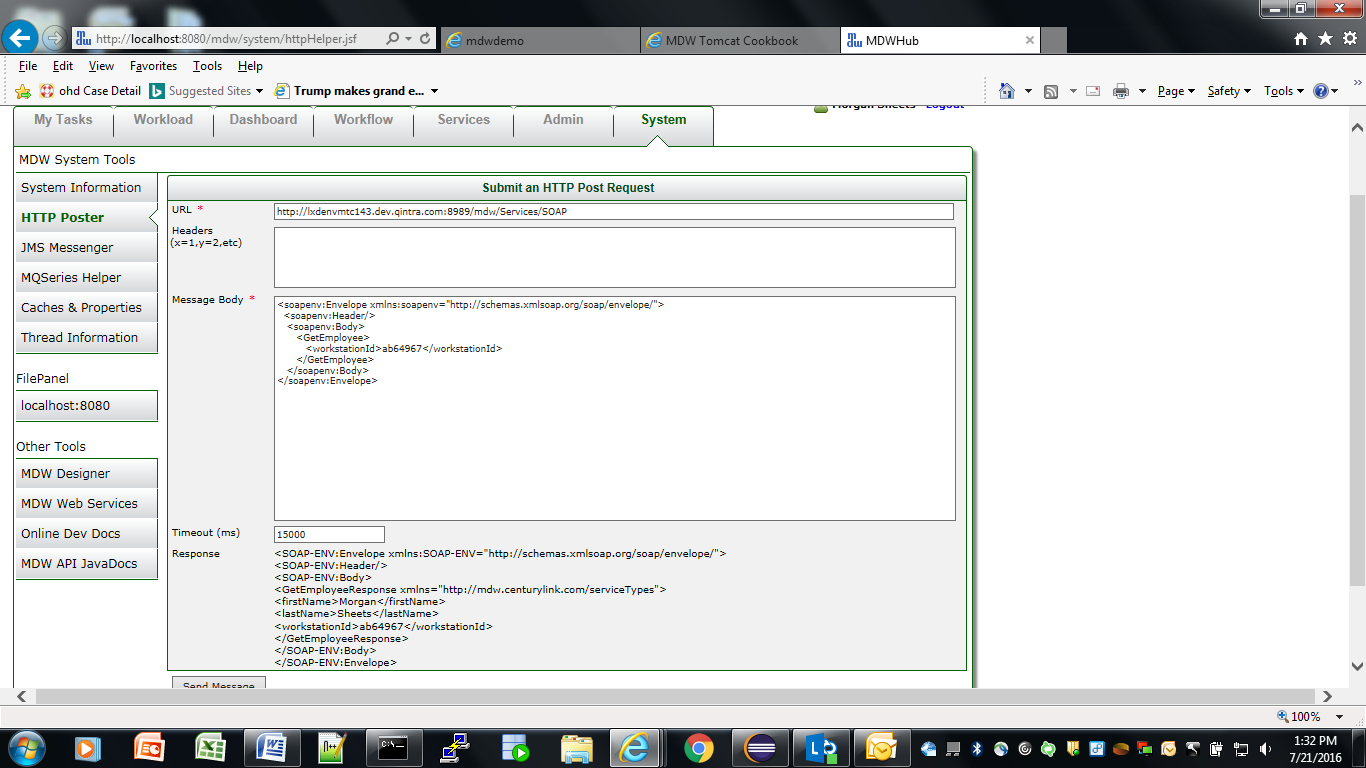
<GetEmployee>

<workstationId>ab64967</workstationId>

</GetEmployee>

</soapenv:Body>  
</soapenv:Envelope>

Click on the Send Message button, and your service process should be executed and you should see a SOAP response like in this screenshot:



If you have trouble getting Site Admin access, or you prefer to use another tool like SoapUI or SOAtest, you can accomplish the same thing by just making sure the endpoint URL is like that in the screenshot and that your request content inside the SOAP body matches your registered External Event Handler.

**Advanced Cloud Development**

[Previous](#local_5) [Contents](#contents) [Next](#advanced_2)

**1. Build a Web Page to Capture User Input**

**TODO:**

* More to come.

**Advanced Cloud Development**

[Previous](#advanced_1) [Contents](#contents) [Next](#advanced_3)

**2. Expose a Custom WebApp**

**TODO:**

* More to come.

**Advanced Cloud Development**

[Previous](#advanced_2) [contents](#contents)

**3. Design a Report and Access it through the Web**

**TODO:**

* More to come.