****

**MDW Services Cookbook – Table of Contents**

**Prerequisites:**

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

Get Started with BPM Cloud Development.

[**Java Container**](#java)

Run Locally for Advanced Capabilities.

**Workflow Services:**

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

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

[**2: Create a Service Process**](#create_service_proc)

Create a basic service process definition.

[**3: Expose as a Web Service**](#expose_soap)

Create a WSDL and invoke your service process through SOAP, REST, and JMS.

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

Execute your process by invoking as a service.

[**5: MDW Camel Integration**](#MDW_Camel_Integration)

Incorporate Apache Camel capabilities into your services

**REST Services:**

[**1: Implement a JAX-RS Web Service**](#jax_rs_service)

Wired through annotations in your Dynamic Java code.

[**2: Add Swagger API Annotations**](#swagger_annotate)

Use Swagger annotations to expose your Service API.

[**3: View Generated REST APIs in MDWHub**](#rest_api_in_mdwhub)

Explore the Swagger-generated API specs as a service consumer would.

**MDW Services 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).

**Optional Plug-Ins:**

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

**http://dist.springsource.org/release/GRECLIPSE/e4.4**

* Quantum DB Plug-In (Database querying)

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

**MDW Services Cookbook – Container**

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

**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. At certain points in this tutorial, we'll link to container-specific steps in the Cookbooks for each supported container.

**Supported Java Containers:**

* Apache Tomcat 7:
  + [**http://tomcat.apache.org/download-70.cgi**](http://tomcat.apache.org/download-70.cgi)
  + [**http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwTomcatCookbook.html**](http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwTomcatCookbook.html)
* Apache ServiceMix 4.4.1 or 5.1.0:
  + [**http://servicemix.apache.org/downloads.html**](http://servicemix.apache.org/downloads.html)
  + [**http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwServiceMixCookbook.html**](http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwServiceMixCookbook.html)
* JBoss Fuse 6.1.0:
  + [**http://www.jboss.org/products/fuse/download**](http://www.jboss.org/products/fuse/download)
  + [**http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwFuseCookbook.html**](http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwFuseCookbook.html)
* Pivotal Cloud Foundry 2.x:
  + [**http://pivotal.io/platform**](http://pivotal.io/platform)
  + [**http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwCloudFoundryCookbook.html**](http://cshare.ad.qintra.com/sites/MDW/Developer%20Resources/Tutorials/MdwCloudFoundryCookbook.html)

**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, MariaDB 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)

**Workflow Services**

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

**1. Create a Local Cloud Project**

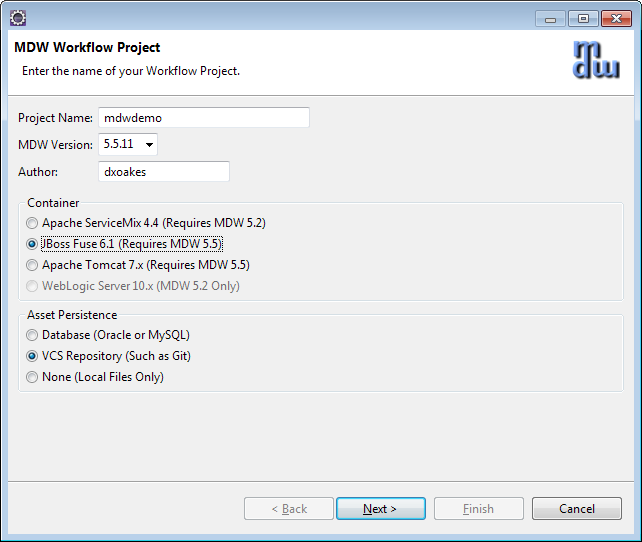
A local cloud project is useful if you want to debug your Java source code and Groovy scripts.

**Open the Designer Perspective in Eclipse:**

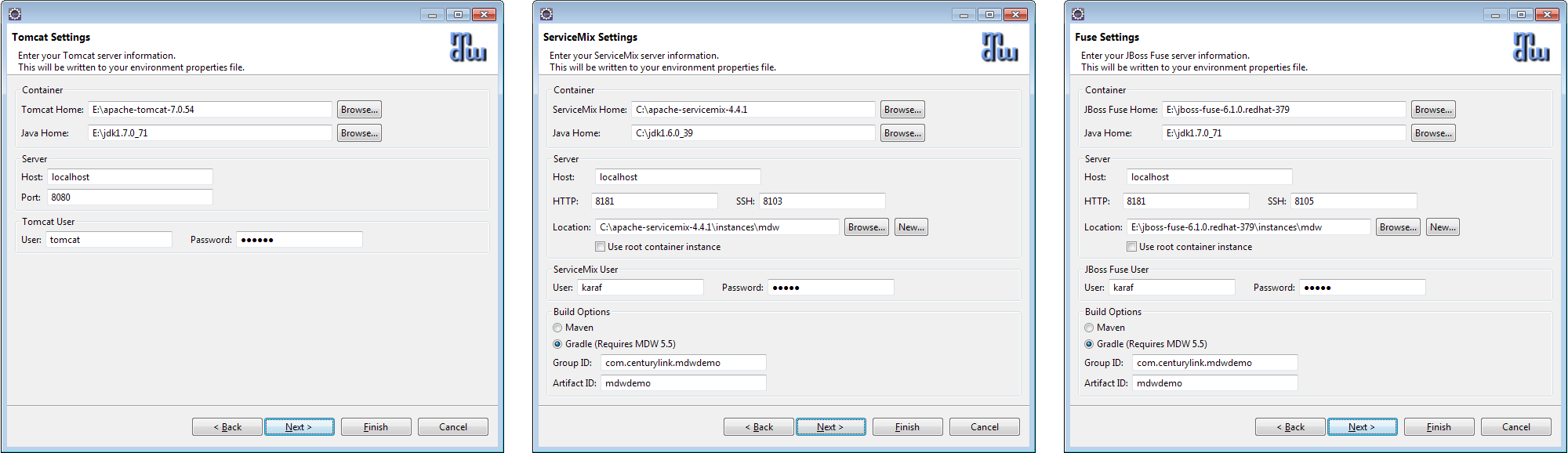
* Launch Eclipse (with the MDW Plug-In installed).
* From the menus select Window > Open Perspective > Other > MDW Designer.
* For detailed documentation covering Designer, refer to the User Guide:
  + [**http://qshare.ad.qintra.com/sites/MDW/User%20Documentation/MDW%20Designer%20User%20Guide.doc**](http://qshare.ad.qintra.com/sites/MDW/User%20Documentation/MDW%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

**Launch the Local Cloud Project wizard:**

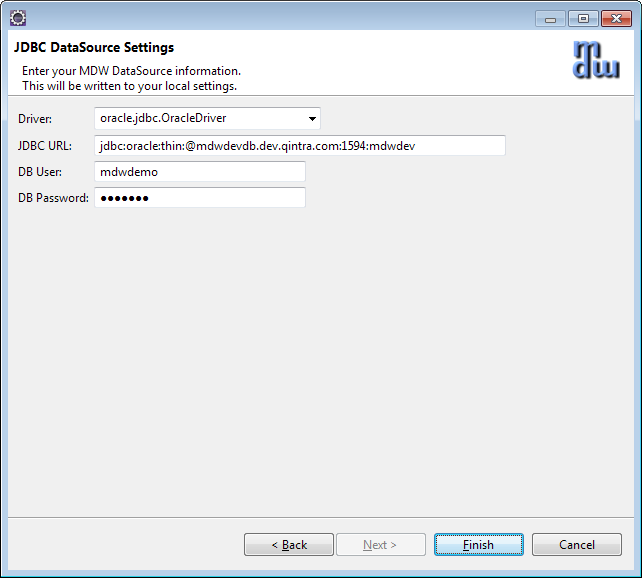
* Right-click inside the blank Process Explorer view and select New > Local Cloud Project. Select the [Supported Container](http://cshare.ad.qintra.com/sites/MDW/Releases/Compatibility) you'll be deploying in, and the type of [Asset Persistence](http://lxdenvmtc099.dev.qintra.com:10001/MDWHub/doc/assetPersistence.html) you'll use.



* When you click Next, you'll be presented with one of these pages specific to your container. Enter the settings for your environment. For details about these settings, refer to the server-specific cookbooks above under Supported Java Containers.



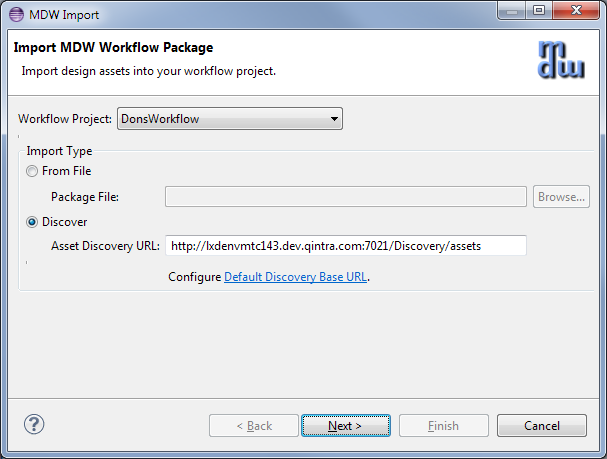
* Click Next again and enter your database connection info. The password for the default mdwdemo database is "mdwdemo".



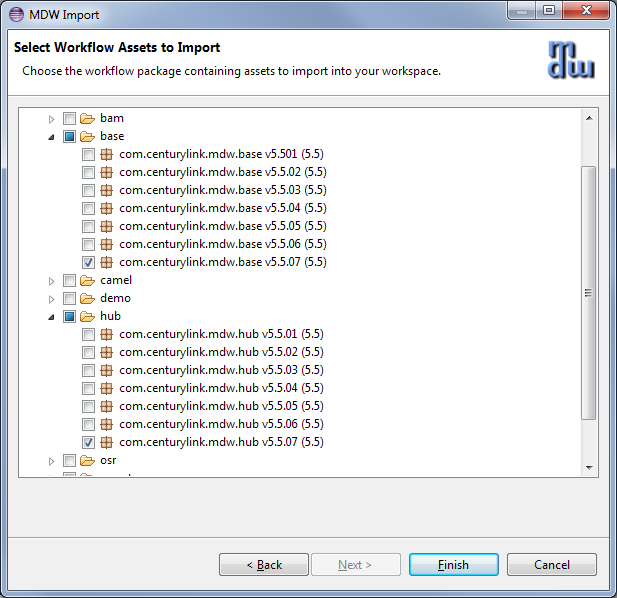
* Click Finish to generate your local cloud project.

**The MDW Base Package:**

* When you create design artifacts in MDW, 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 this way. The essential assets required by MDW are included in the packages "com.centurylink.mdw.base" and "com.centurylink.mdw.hub". If you chose database asset persistence using the mdwdemo database, these base packages will already exist, and you can skip down to [Section 2](#local_2). Otherwise, if you're using a new database or VCS asset persistence, you'll need to import these packages locally from the MDW repository as follows.
* Expand your newly-created workflow project in Process Explorer and you'll see that it currently contains no 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.

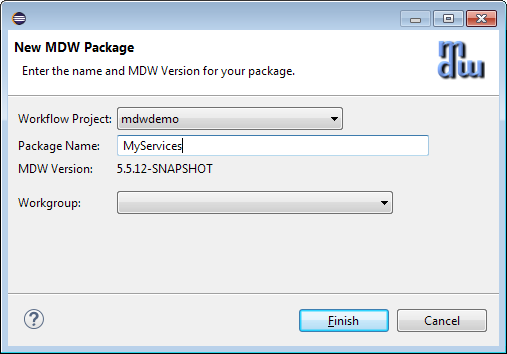
**Workflow Services**

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

**2. Create a Service Process**

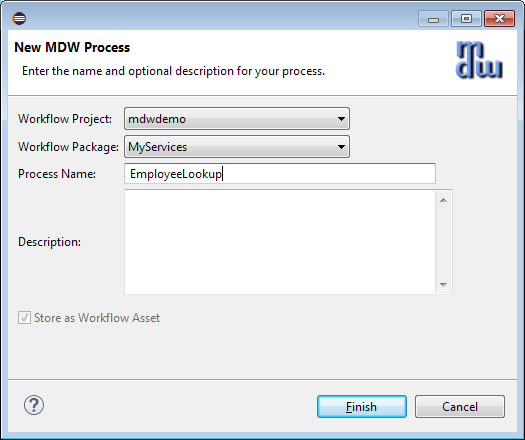
**Create a Workflow Package:**

* The top-level branches in the Process Explorer 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.

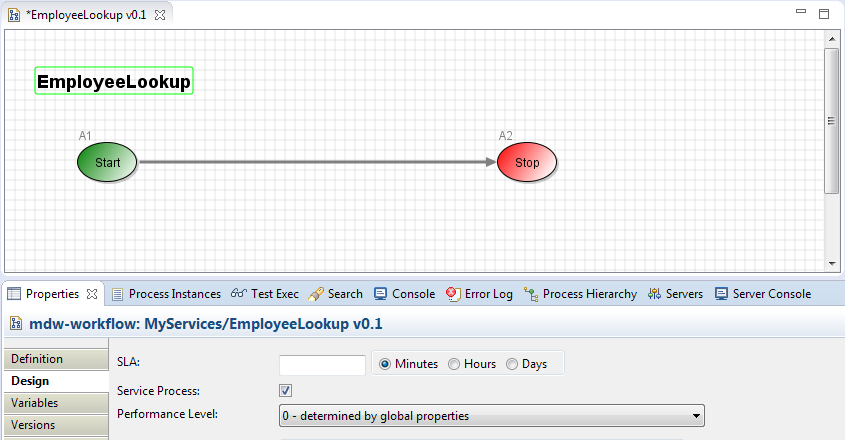


**Create the Service Process:**

* Right-click on your new package in Process Explorer and select New > MDW Process. Enter the process name and description (no workgroup), and click Finish.

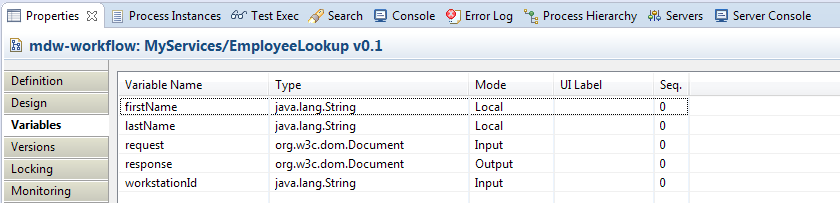


* After your process is created, double-click on the process title or on a blank area somewhere in the canvas to display the Properties View. Select the Design properties tab and check "Service Process" to identify EmployeeLookup as a synchronous process returning a response.

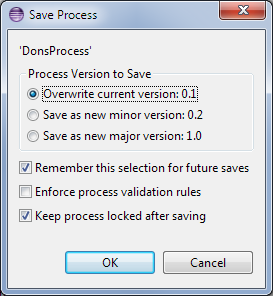


**Add some Process Variables:**

* The convention in MDW is that a service request variable is named "request" and a service response variable is named "response". There's the option to name these differently, but for simplicity let's go along with the convention here. On the Variables property tab, create these two variables in your process with type org.w3c.dom.Document. Set the mode for the request variable to be Input, and the mode for the response to be Output. Add an Input String variable called workstationId, and Local variables firstName and lastName.

****

* 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.

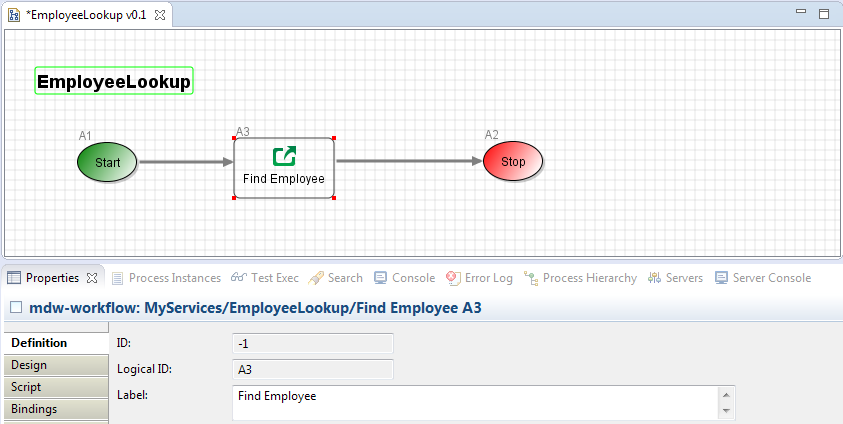


**Add an LDAP Lookup Step:**

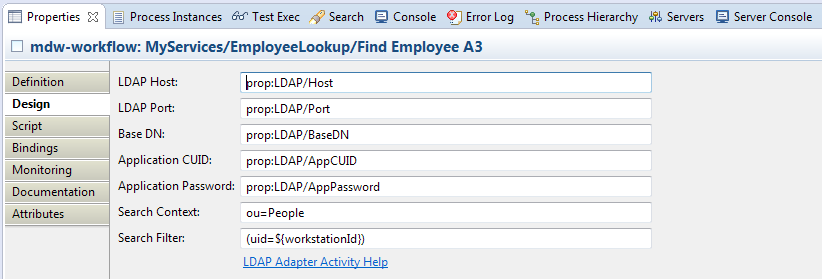
* In the toolbox view along the right-hand side of the design canvas, find the activity labeled LDAP Adapter and drag it onto the canvas and insert it into your flow between the Start and Stop activities.

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*")*.*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.

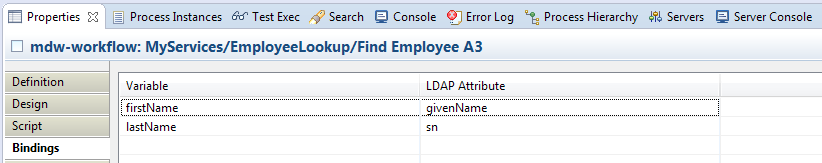
Change the label of the LDAP activity in your process to "Find Employee" by editing its value on the Definition properties tab.

****

* On the Design tab for the LDAP Adapter, enter (uid=${workstationId}).



* On the Bindings tab for the LDAP Adapter, specify how the firstName and lastName variables are to be populated from LDAP attributes.



* At this point you're ready to test your process before exposing it as a service. To execute EmployeeLookup, you'll need a running server instance.
* **Note**: In order to connect to the LDAP servers, your ServiceMix/Fuse or Tomcat server must be configured with the CenturyLinkQCA trust keystore. Ecom environments are usually configured with the appropriate certificates by default.
  + ServiceMix/Fuse: reference the jks file as follows in etc/system.properties:  
    javax.net.ssl.trustStore=etc/CenturyLinkQCA.jks  
    In your local development environment this should happen automatically when you run the buildConfig Ant target.
  + Tomcat: Edit the MDW Tomcat server and add the following Java Option (substitute your workspace location):   
    -Djavax.net.ssl.trustStore=c:\workspaces\mdw\mdw-framework\deploy\certs\CenturyLinkQCA.jks

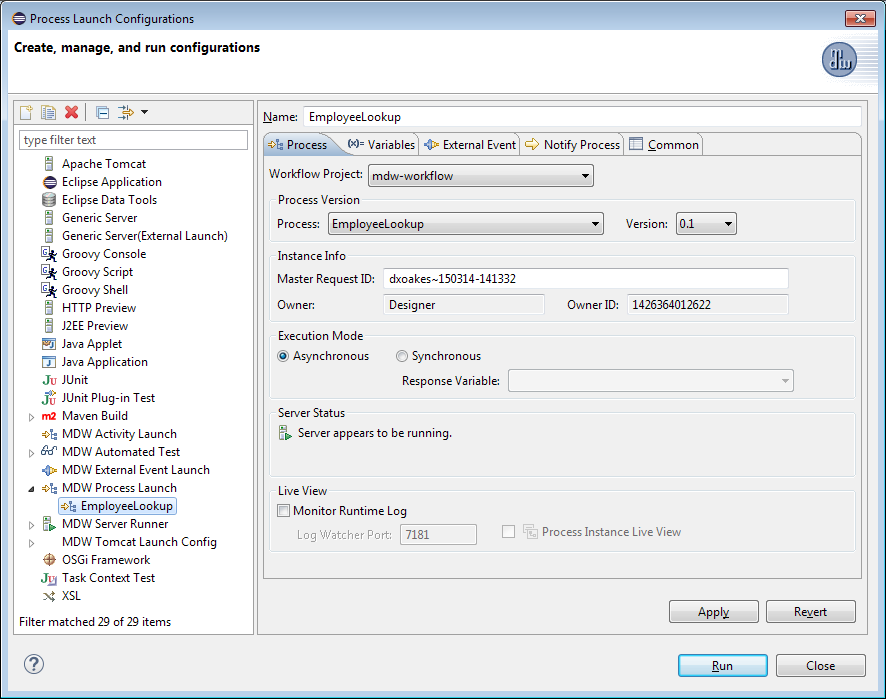
**Get Your Server Running:**

* Depending on which supported container you're using, you can follow one of the server setup exercises. You'll need to follow the steps from one of these guides to the point where MDW is deployed and you're able to start and stop your server from the Eclipse Servers view. For ServiceMix and Fuse this involves building and deploying your workflow bundle as well as MDW.
  + [Tomcat Server Setup](MdwTomcatCookbook.html#local_3)
  + [ServiceMix Server Setup](MdwServiceMixCookbook.html#local_project)
  + [Fuse Server Setup](MdwFuseCookbook.html#deploy_mdw)
* You can confirm that MDW was successfully deployed by accessing MDWHub in your browser:
  + Tomcat: **http://localhost:8080/mdw**
  + ServiceMix/Fuse: **http://localhost:8181/MDWHub**
* Troubleshooting: if you encounter an exception like below, it means you need to download the [JCE 6 Unlimited Strength Jurisdiction Policy Files](http://www.oracle.com/technetwork/java/javase/downloads/jce-6-download-429243.html) for JRE 1.6. Or [JCE 7 Unlimited Strength Jurisdiction Policy Files](http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html)  for JRE 1.7

java.security.InvalidKeyException: Illegal key size or default parameters

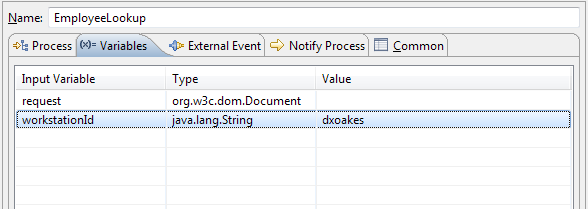
**Open the Process Launch Dialog:**

* Right-click on the EmployeeLookup process in Process Explorer view 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.



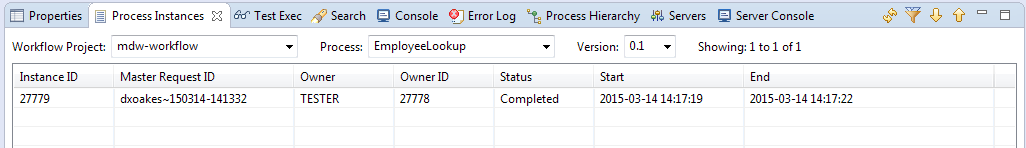
**Populate the Input Variable:**

* Select the Variables tab in the launch dialog, and populate the workstationId input with your CenturyLink workstation ID.

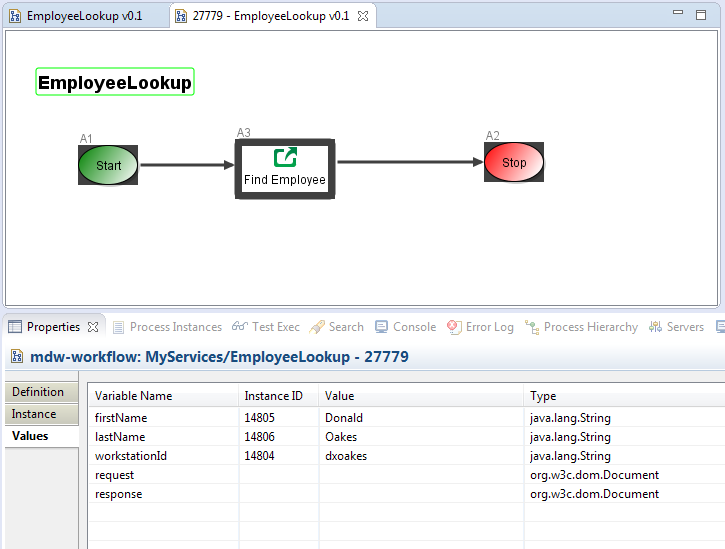


**Launch and View an Instance:**

* Click Run on the launch dialog to run an instance of your process. The Eclipse Console view should indicate that EmployeeLookup was successfully launched. To view the process instance you just ran, right-click somewhere inside the canvas and select View Instances. The Process Instance view should appear.



* Double-click on your instance row to display the runtime view. A Legend view appears on the right showing what the borders surrounding the activities mean. To inspect the runtime variable values for the instance, click the Values property tab. If the lookup ran correctly, you should see the firstName and lastName variables populated.



**Register an External Event Handler:**

* Do you have a service yet? Not quite. The lookup you just performed took a simple string input and returned nothing. A real service would likely take an XML or JSON document as input, and return something similar with the results. To expose this process as a service you'll need to register an External Event Handler. An External Event Handler is the MDW mechanism for performing content-based routing of incoming requests to your service process. So in our example it’s a way of specifying that an XML payload like the following should be routed to the EmployeeLookup service process:

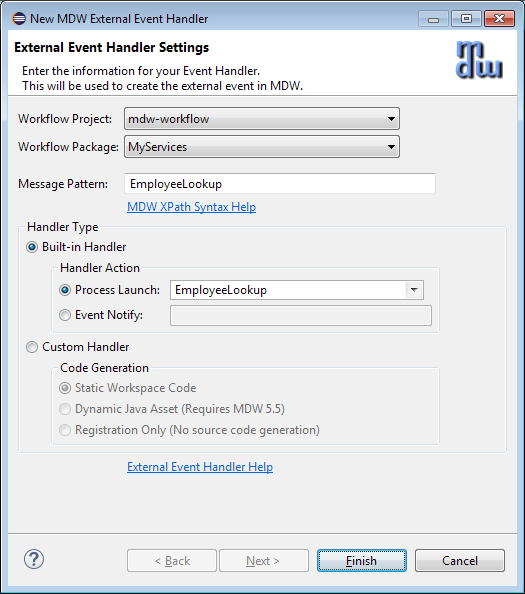
<EmployeeLookup>

<workstationId>dxoakes</workstationId>

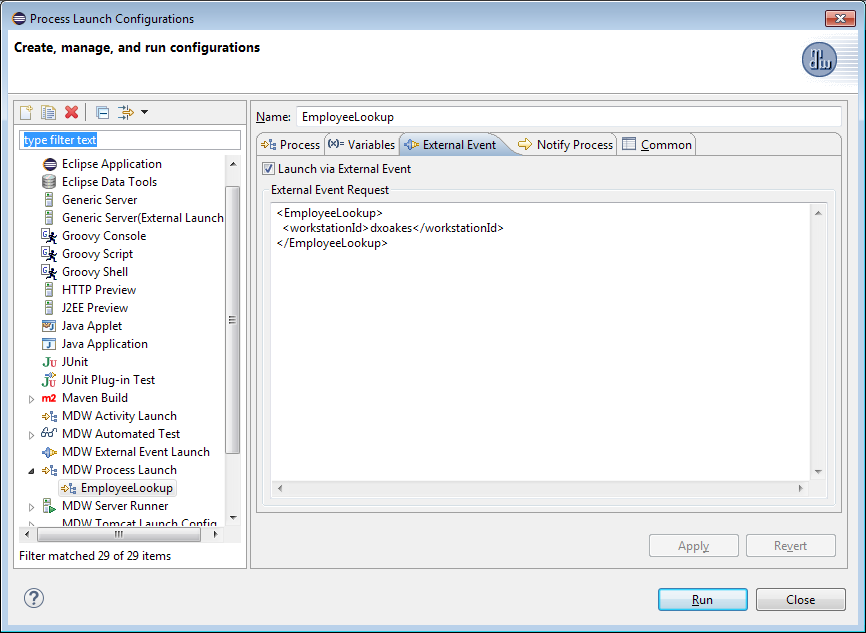
</EmployeeLookup>

* To create an Event Handler, right-click on your MyServices package in Process Explorer and select New > Event Handler > External Event Handler and populate the wizard page as illustrated. The Message Pattern in this case represents an XPath expression that tells MDW that matching requests should go through this handler. The built-in Process Launch handler simply runs an instance of the selected process, passing along the request document.

Note: If you're using a shared database you'll want to make your Message Pattern unique.



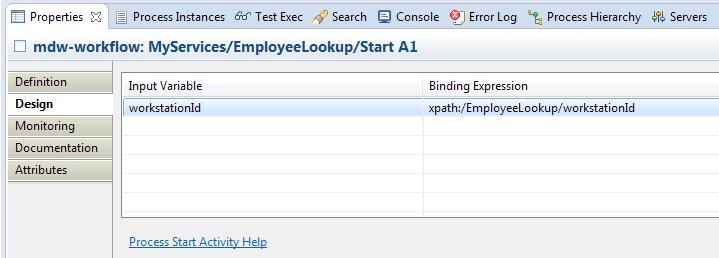
* Click Finish, and you should see the Event Handler appear under your package in Process Explorer. You can view and edit details on the Definition properties tab for the handler.
* Run your process again, but this time use the External Event tab to specify an XML value like the following. This tells Designer to send a REST request to MDW with the designated payload.



* After launching, if you view the values for your latest instance you'll see that the request variable was correctly populated with the incoming payload, but the LDAP lookup returned nothing because this time workstationId was not populated. There are many ways in MDW to pull values from a request document and use them in workflow activities like the LDAP adapter. For simplicity we'll use the XPath binding feature in the process start activity.

**Map an Input Variable Binding:**

* On the Design tab for the Start activity in your process, enter a Binding Expression value of xpath:/EmployeeLookup/workstationId mapped to the workstationId input variable.



**Create the Response:**

* The response output variable has so far remained unpopulated. There are innumerable ways to build a response, and in a real-world service this might be cumulative based on multiple workflow steps. To keep this exercise simple we'll use a PostScript on the LDAP Adapter activity we already have. On the Script tab for this activity, move the response variable over to the Writable column. Then make sure the selected PostScript language is Groovy, and click the Edit Script link. Write a script like the following to populate the response:

response = "<Employee>\n" +

" <workstationId>" + workstationId + "</workstationId>\n" +

" <firstName>" + firstName + "</firstName>\n" +

" <lastName>" + lastName + "</lastName>\n" +

"</Employee>"

* Notice that process variables are implicitly available and assignable from within your script. Notice also that we're building a string here, which MDW automatically translates to the appropriate type (org.w3c.dom.Document). Save the script content and also the process. If you run the process again through the event handler you should see the correct response in the Eclipse console, and the values should be correctly populated for the instance.

**Workflow Services**

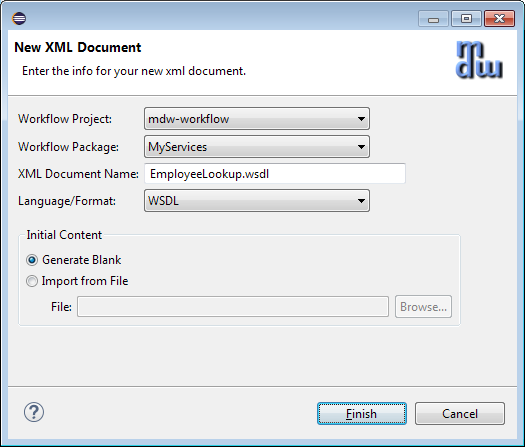
[Previous](#create_service_proc) [Contents](#contents) [Next](#consume_web_service)

**3. Expose as a SOAP Web Service**

At this point your process is already exposed through all the protocols that MDW supports (the default behavior). When you tested it above using Designer you were consuming it over the REST interface. It can also be consumed through a SOAP document-style request, and MDW will extract the payload from within the SOAP envelope and use this to perform the required routing. But what about the WSDL? In this section we'll create a WSDL asset that'll be hosted in MDW and available to the consumers of your service.

**Create a WSDL Asset:**

* You've already built and exposed the service, now you just need to create a document-style WSDL asset that describes it. In Process Explorer view, right-click on your 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"*

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

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

<wsdl:types>

<xsd:schema

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

elementFormDefault=*"qualified"*>

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

<xsd:complexType>

<xsd:sequence>

<xsd:element

name=*"workstationId"*

type=*"xsd:string"* />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

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

<xsd:complexType>

<xsd:sequence>

<xsd:element

name=*"workstationId"*

type=*"xsd:string"* />

<xsd:element

name=*"firstName"*

type=*"xsd:string"* />

<xsd:element

name=*"lastName"*

type=*"xsd:string"* />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

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

<xsd:complexType>

<xsd:sequence>

<xsd:element

name=*"workstationId"*

type=*"xsd:string"* />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:schema>

</wsdl:types>

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

<wsdl:part

name=*"payload"*

element=*"typens:EmployeeLookup"* />

</wsdl:message>

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

<wsdl:part

name=*"payload"*

element=*"typens:Employee"* />

</wsdl:message>

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

<wsdl:part

name=*"payload"*

element=*"typens:UnknownEmployeeFault"* />

</wsdl:message>

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

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

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

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

<wsdl:fault

name=*"UnknownEmployee"*

message=*"tns:UnknownEmployeeFault"* />

</wsdl:operation>

</wsdl:portType>

<wsdl:binding

name=*"EmployeeSOAPBinding"*

type=*"tns:EmployeeLookupPortType"*>

<soap:binding

style=*"document"*

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

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

<wsdl:input>

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

</wsdl:input>

<wsdl:output>

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

</wsdl:output>

<wsdl:fault name=*"UnknownEmployee"*>

<soap:fault

use=*"literal"*

name=*"UnknownEmployee"* />

</wsdl:fault>

</wsdl:operation>

</wsdl:binding>

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

<wsdl:port

binding=*"tns:EmployeeSOAPBinding"*

name=*"soap"*>

<soap:address location=*"${mdw.services.url}/SOAP/EmployeeLookup.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:
  + Tomcat: <http://localhost:8080/mdw/SOAP/EmployeeLookup.wsdl>
  + ServiceMix/Fuse: <http://localhost:8181/MDWHub/SOAP/EmployeeLookup.wsdl>

**Invoke Your Service through SOAP:**

* If you're familiar with a tool like SoapUI, you can use that to autogenerate a SOAP request for your service. The MDWHub System tab also includes the HTTP Poster utility that you can use to test your service. Access MDWHub in your browser through a URL like these:
  + Tomcat: <http://localhost:8080/mdw>
  + ServiceMix/Fuse: <http://localhost:8181/MDWHub>

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. Populate the Message Body with something like the following:

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

<soapenv:Header/>

<soapenv:Body>

<EmployeeLookup>

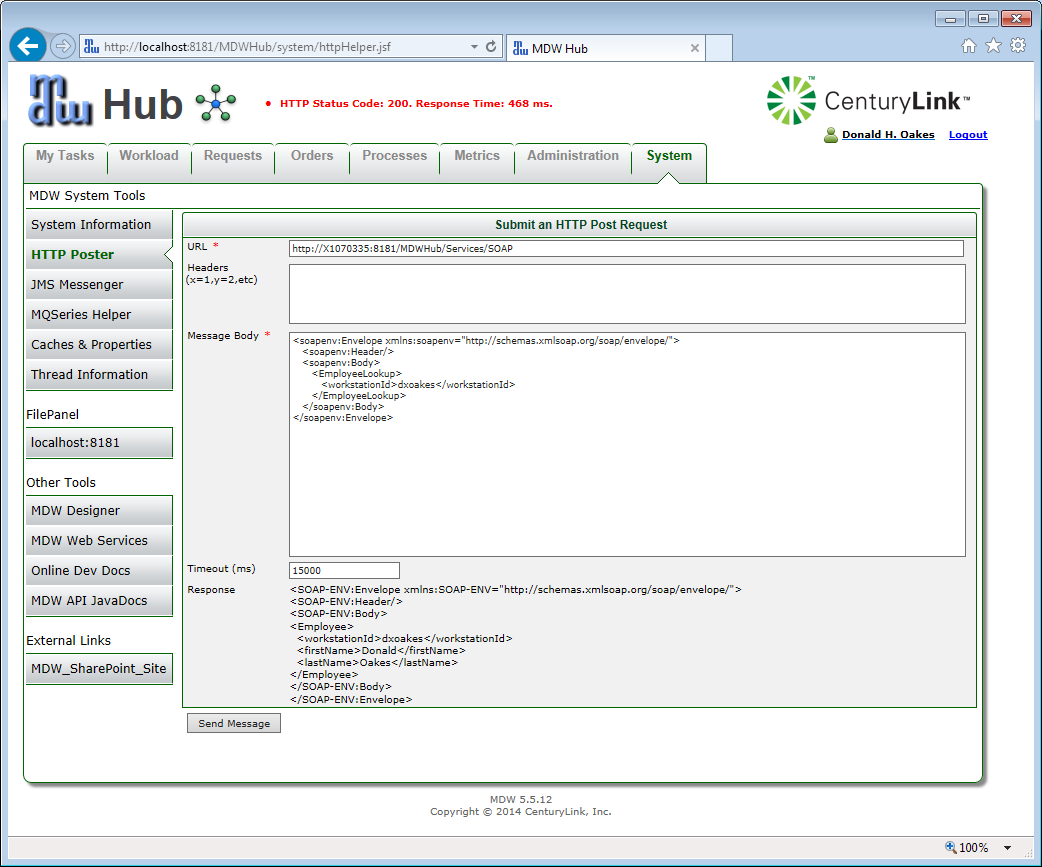
<workstationId>dxoakes</workstationId>

</EmployeeLookup>

</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:



**Invoke Your Service through REST and JMS:**

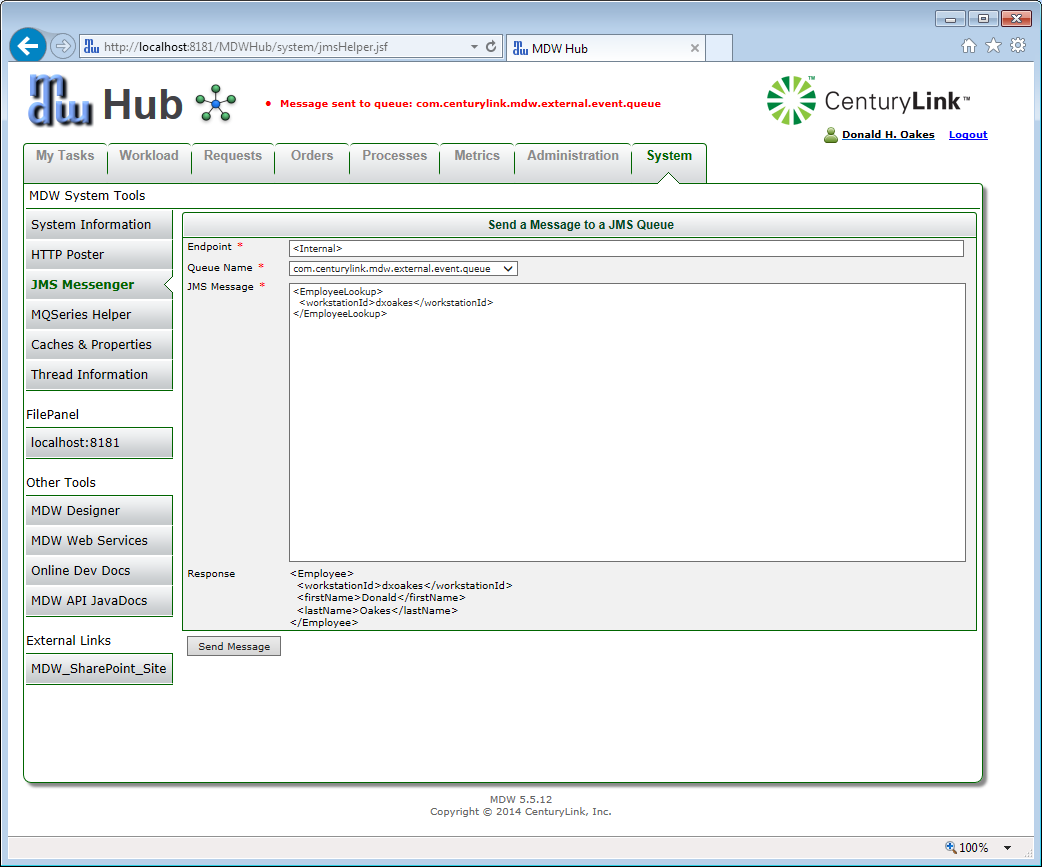
* To invoke through REST or JMS use the raw payload without the SOAP envelope:

<EmployeeLookup>

<workstationId>dxoakes</workstationId>

</EmployeeLookup>

On the MDWHub System tab you can use the HTTP Poster to submit to the REST endpoint, or you can use the JMS Messenger as illustrated below.



**Workflow Services**

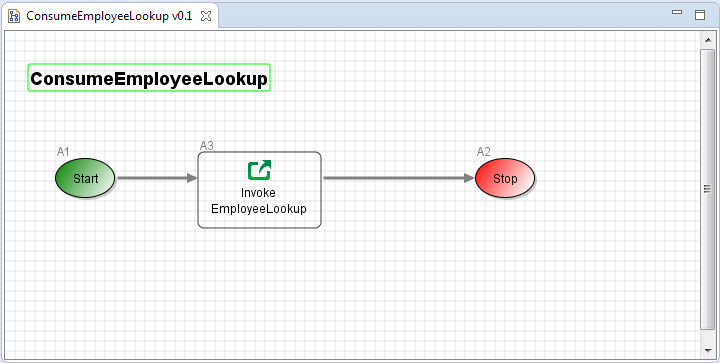
[Previous](#expose_soap) [Contents](#contents) [Next](#non_workflow_service)

**4. Consume a Web Service**

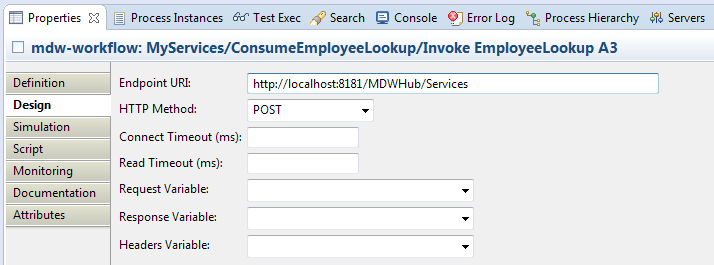
MDW comes with Adapter activities for consuming services over many protocols from within your workflow processes. In this exercise we'll use the REST Service Adapter activity to invoke the EmployeeLookup service you just created.

**Create a Process with a REST Service Activity:**

* Create a new process to consume your service. Add an input String variable called employeeId, and a Local String variable called employeeName. From the Toolbox view drag a RESTful Service Adapter onto the canvas and insert it into your process flow.



* On the Design tab for the web service activity, set the HTTP Method to POST and enter the same REST endpoint URL you used for testing your service in Section 3.
  + Tomcat: http://localhost:8080/mdw/Services
  + ServiceMix/Fuse: http://localhost:8181/MDWHub/Services



**Add Pre and Post Script:**

* With the REST activity in a real-world workflow, you might bind document variables to the service input and output through the Request Variable and Response Variable dropdowns pictured above. To simplify this tutorial, let's take advantage of the Pre and Post script to build the request and pull values out of the response. On the Script property tab for the Invoke EmployeeLookup activity, edit the prescript, adding the Groovy code below to return the request XML posted to the service (if you've installed the Groovy Eclipse plugin you'll get syntax highlighting and autocomplete):

**return** '''<EmployeeLookup>

<workstationId>''' + employeeId + '''</workstationId>

</EmployeeLookup>'''

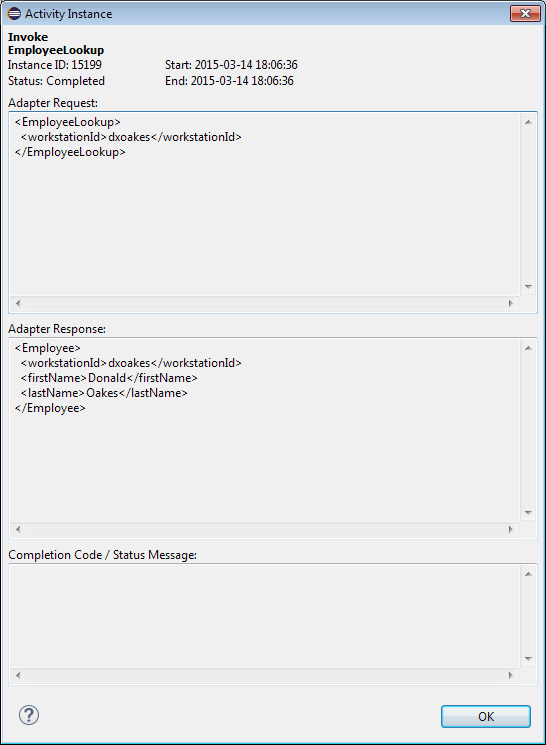
* Edit the postscript as follows:

**def** employeeNode = **new** groovy.util.XmlParser().parseText(response)

employeeName = employeeNode.firstName.text() + ' ' + employeeNode.lastName.text()

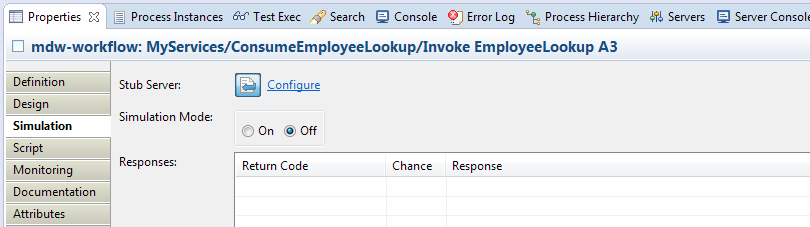
**Save and Run Your Process:**

* Launch your process, entering employeeId on the process launch Variables tab. View the instance to confirm that employeeName was populated as expected.
* In the process instance view, double-click the Invoke EmployeeLookup activity instance. Then on the Instance property tab, double-click on the activity instance row. The Activity Instance dialog shows you the raw request and response values that were sent over the wire. If we'd used a SOAP adapter activity instead of REST, this would show the full content including SOAP Envelopes.



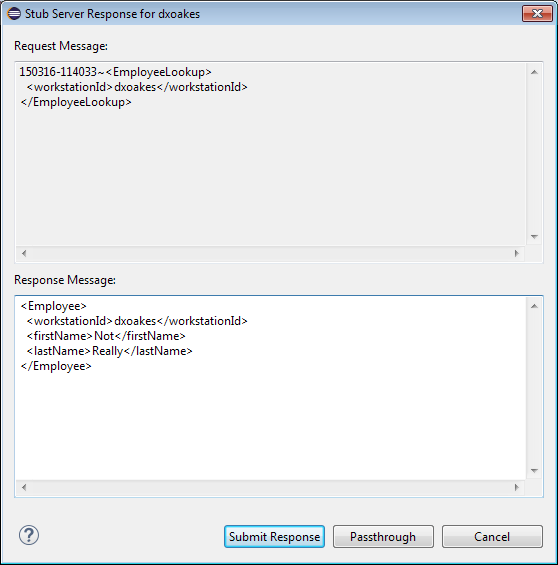
**Stub Mode and Response Simulation:**

* At times when performing services orchestration using MDW you may be designing a flow before one or more of your consumed services is not yet available. Or you may not be ready to make an actual call because you're still debugging your workflow. For situations like this MDW provides Stub Mode and Response Simulation. Stub Mode is for local development and [Automated Testing](http://lxdenvmtc099.dev.qintra.com:10001/MDWHub/doc/automatedTesting.html). Response Simulation is used to hardwire the responses for specific adapter activities within a given environment. Both of these features are accessed via the Simulation property tab. Click this tab for the Invoke EmployeeLookup REST adapter in the process you just build. To try out Stub Mode, depress the Stub Server button (no need to Configure since the defaults should be fine).



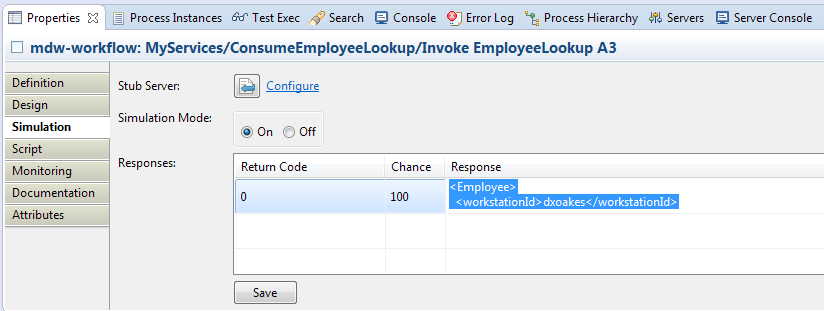
Note that this is a global setting; meaning once the stub server's running it intercepts all adapter activity requests. Note also that it can be difficult to determine whether the button is depressed (i.e. stubbing is on).

* Once you've got stub mode turned on, run the process again and you'll be presented with a dialog prompting you for the desired response for this case.



Whatever is typed in the Response Message textbox will be returned to your process as the adapter response, and you should be able to confirm this by checking the runtime values of the process instance.

* To simulate a response, disable the stub server and instead set Simulation Mode to On. Then provide a Return Code (not currently used), Chance (weighted probability when multiple responses), and Response value for each different hardwired response scenario.



* These simulated response settings are meant to be per-environment, so they don't get saved with the process definition but rather as so-called "override attributes". For this reason there's a Save button directly on the Simulation property tab.

**Workflow Services**

[Previous](#consume_web_service) [Contents](#contents) [Next](#jax_rs_service)

**5. MDW Camel Integration**

Check out the [MDW Camel Cookbook](MdwCamelCookbook.html) to see how you can integrate Apache Camel with your MDW application.

**REST Services**

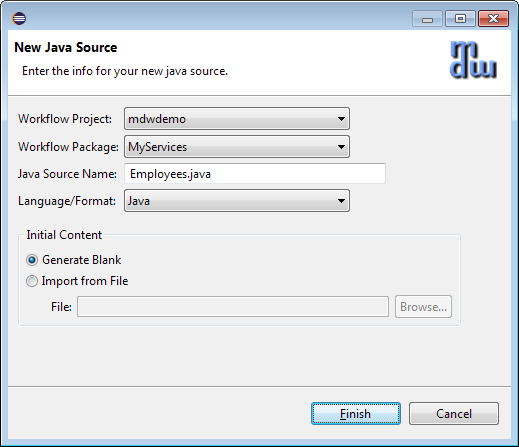
[Previous](#MDW_Camel_Integration) [Contents](#contents) [Next](#swagger_annotate)

**1.** **Implement a JAX-RS Web Service**

Besides implementing services by way of an MDW workflow process, you can easily expose your Dynamic Java class as a REST service using JAX-RS annotations.

**Create a Java Asset to Implement a Resource Service:**

* Right-click on your package in Process Explorer view and select new > Java Source. By convention the Java class name will also be the name of your service resource. Also by convention your workflow package name is the root of the REST endpoint URL path that consumers will use to access your service. For this simple example we're using the MyServices package. In a real-world app you'll probably use a qualified package name like com.centurylink.my.services, and in that case you can specify a simplified URL path through the JAX-RS **Path** annotation, which will be illustrated later.



* Implement a REST service, using the JAX-RS @Path annotation and extending the MDW JsonRestService class:

**package** MyServices;

**import** java.util.Map;

**import** javax.ws.rs.Path;

**import** org.json.JSONException;

**import** org.json.JSONObject;

**import** com.centurylink.mdw.common.service.ServiceException;

**import** com.centurylink.mdw.model.value.user.UserVO;

**import** com.centurylink.mdw.services.rest.JsonRestService;

@Path("/Employees")

**public** **class** Employees **extends** JsonRestService {

@Override

@Path("/{id}")

**public** JSONObject get(String path, Map<String,String> headers)

**throws** ServiceException, JSONException {

String id = getSegment(path, 2);

**if** ("dxoakes".equals(id)) {

UserVO emp = **new** UserVO(id);

emp.setName("Donald Oakes");

emp.setAttribute("Email", "donald.oakes@centurylink.com");

emp.setAttribute("Phone", "303 992 9747");

**return** emp.getJson();

}

**else** {

**return** **null**;

}

}

}

* Access your service using a GET request from your browser with a URL like the following:
  + Tomcat: <http://localhost:8080/mdw/Services/MyServices/Employee/dxoakes>
  + ServiceMix/Fuse: <http://localhost:8181/MDWHub/Services/MyServices/Employee/dxoakes>

**Add Create Capability to Your REST Service:**

* In the REST paradigm, creates are performed via HTTP POST. So to implement the ability to add a new Employee, override the post() method:

@Override

**protected** JSONObject post(String path, JSONObject content, Map<String, String> headers)

**throws** ServiceException, JSONException {

UserVO emp = **new** UserVO(content);

String id = emp.getCuid();

**if** (id == **null**)

**throw** **new** ServiceException(***HTTP\_400\_BAD\_REQUEST***, "Missing user id");

**if** (id.equals("dxoakes"))

**throw** **new** ServiceException(***HTTP\_409\_CONFLICT***, "Employee id exists: " + id);

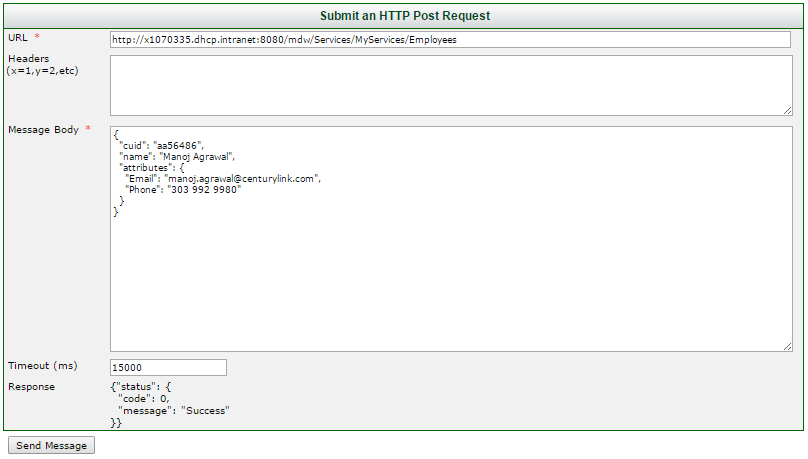
// **TODO**: actual work to create the employee

System.***out***.println("Creating user: " + emp.getJson().toString(2));

**return** **null**; // null indicates successful POST

}

* Save your Dynamic Java asset, and use the MDWHub HTTP Poster tool to submit a POST request to add a new employee from your browser:



* TODO:
* Illustrate PUT/DELETE
* Describe the JsonRestService and XmlRestService base classes, authentication, etc.

**REST Services**

[Previous](#jax_rs_service) [Contents](#contents) [Next](#rest_api_in_mdwhub)

**2. Add Swagger API Annotations**

With MDW REST services you can automatically generate Swagger documentation just by adding the appropriate annotations to your Dynamic Java. This is not only a convenient way to maintain this documentation, but it also means that it's always up-to-date with the implementation. [Swagger](http://swagger.io/) documentation is a powerful way to communicate the specifics of your REST interface to potential consumers. [Swagger annotations](https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X) represent a convenient mechanism for service developers to produce this documentation without having to hand-craft the JSON or YAML results.

**Add the @Api Annotation to Your Service:**

* The [Swagger Api](https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X#api) annotation goes on your class declaration along with the JAX-RS Path annotation. The tag value in your annotation provides a high-level description of the its purpose:

@Path("/Employees")

@Api("CenturyLink employees service")

**public** **class** Employees **extends** JsonRestService {

**…**

**Add @ApiOperation Annotations to Your Methods:**

* The [ApiOperation](https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X#apioperation) annotation documents the specifics of a service endpoint operation, including any input or output model types. The [ApiImplicitParams](https://github.com/swagger-api/swagger-core/wiki/Annotations-1.5.X#apiimplicitparam-apiimplicitparams) annotation is useful for indicating the body content of a POST or PUT requests. After adding these annotations to Employees.java, the code will look something like this:

**package** MyServices;

**import** java.util.Map;

**import** javax.ws.rs.Path;

**import** org.json.JSONException;

**import** org.json.JSONObject;

**import** com.centurylink.mdw.common.service.ServiceException;

**import** com.centurylink.mdw.common.service.types.StatusMessage;

**import** com.centurylink.mdw.model.value.user.UserVO;

**import** com.centurylink.mdw.services.rest.JsonRestService;

**import** io.swagger.annotations.Api;

**import** io.swagger.annotations.ApiImplicitParam;

**import** io.swagger.annotations.ApiImplicitParams;

**import** io.swagger.annotations.ApiOperation;

@Path("/Employees")

@Api("CenturyLink employees service")

**public** **class** Employees **extends** JsonRestService {

@Override

@Path("/{id}")

@ApiOperation(value="Retrieve an employee by their ID",

notes="Currently only retrieves a single employee, and only dxoakes.",

response=Employee.**class**)

**public** JSONObject get(String path, Map<String,String> headers)

**throws** ServiceException, JSONException {

String id = getSegment(path, 2);

**if** ("dxoakes".equals(id)) {

UserVO emp = **new** UserVO(id);

emp.setName("Donald Oakes");

emp.setAttribute("Email", "donald.oakes@centurylink.com");

emp.setAttribute("Phone", "303 992 9747");

**return** emp.getJson();

}

**else** {

**return** **null**;

}

}

@Override

@ApiOperation(value="Create an employee",

notes="Does not actually create anything as yet.",

response=StatusMessage.**class**)

@ApiImplicitParams({

@ApiImplicitParam(name="Employee", paramType="body", dataType="MyServices.Employee")})

**public** JSONObject post(String path, JSONObject content, Map<String, String> headers)

**throws** ServiceException, JSONException {

UserVO emp = **new** UserVO(content);

String id = emp.getCuid();

**if** (id == **null**)

**throw** **new** ServiceException(***HTTP\_400\_BAD\_REQUEST***, "Missing user id");

**if** (id.equals("dxoakes"))

**throw** **new** ServiceException(***HTTP\_409\_CONFLICT***, "Employee id exists: " + id);

// **TODO**: actual work to create the employee

System.***out***.println("Creating user: " + emp.getJson().toString(2));

**return** **null**; // null indicates successful POST

}

}

**Add Swagger Annotations to the Employee Model Class:**

* To enable consumers to easily create request content and interpret responses, you can annotate the related model objects so that they're discovered when documentation is generated. In the Employee dynamic Java class, add the following class-level annotation:

@ApiModel(value="Employee", description="Centurylink employee")

**public** **class** Employee **extends** UserVO **implements** Jsonable {

**…**

**REST Services**

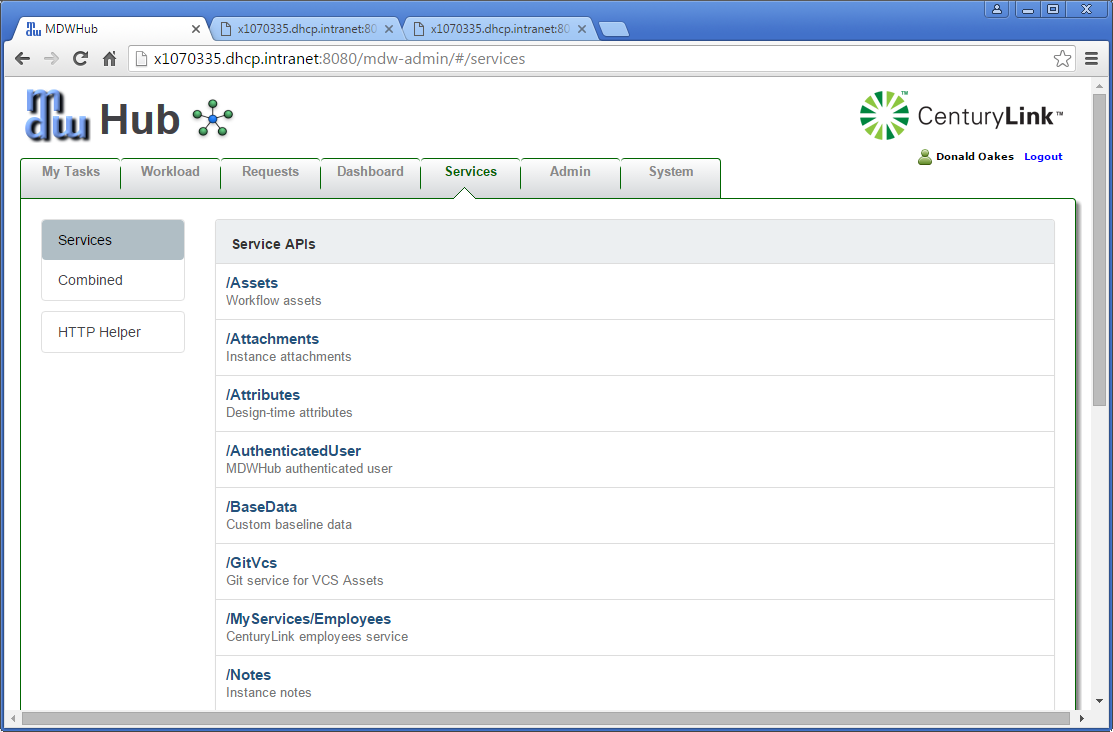
[Previous](#swagger_annotate) [Contents](#contents) [Next](#MDW_Camel_Integration)

**3. View Generated REST APIs in MDWHub**

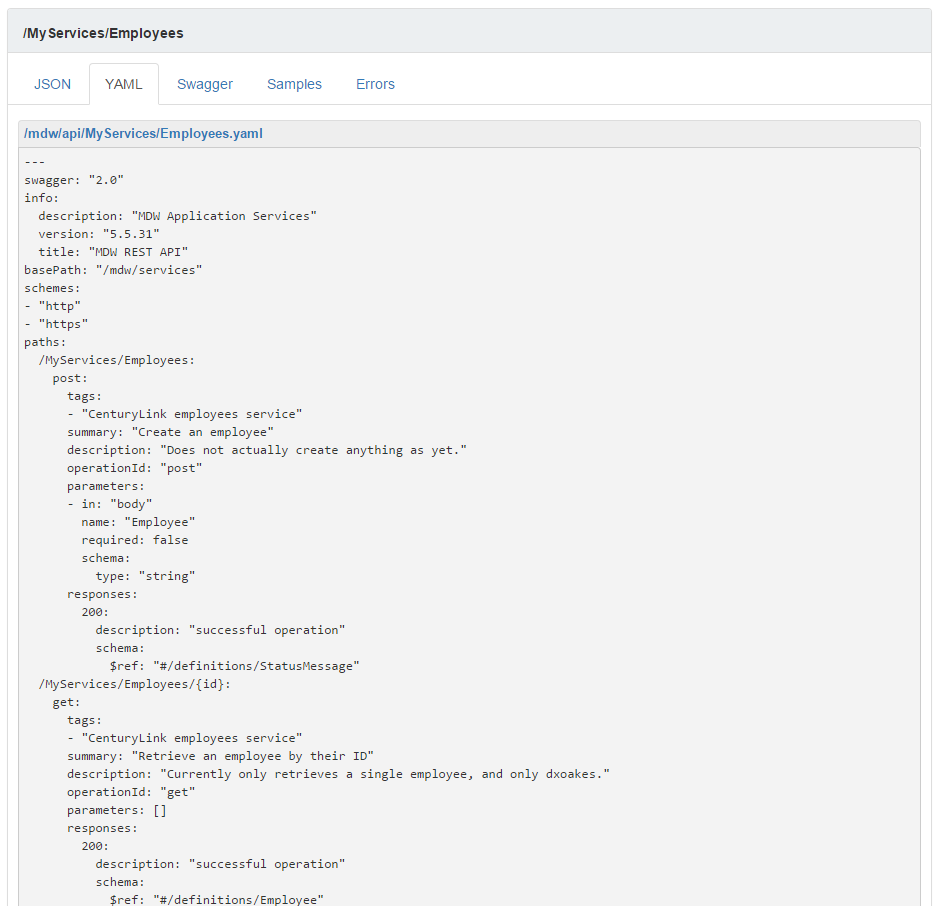
MDWHub comes with a UI for displaying your generated Swagger API documentation, along with the standard MDW REST APIs.

**Access the MDWHub Service API Page for Your Service:**

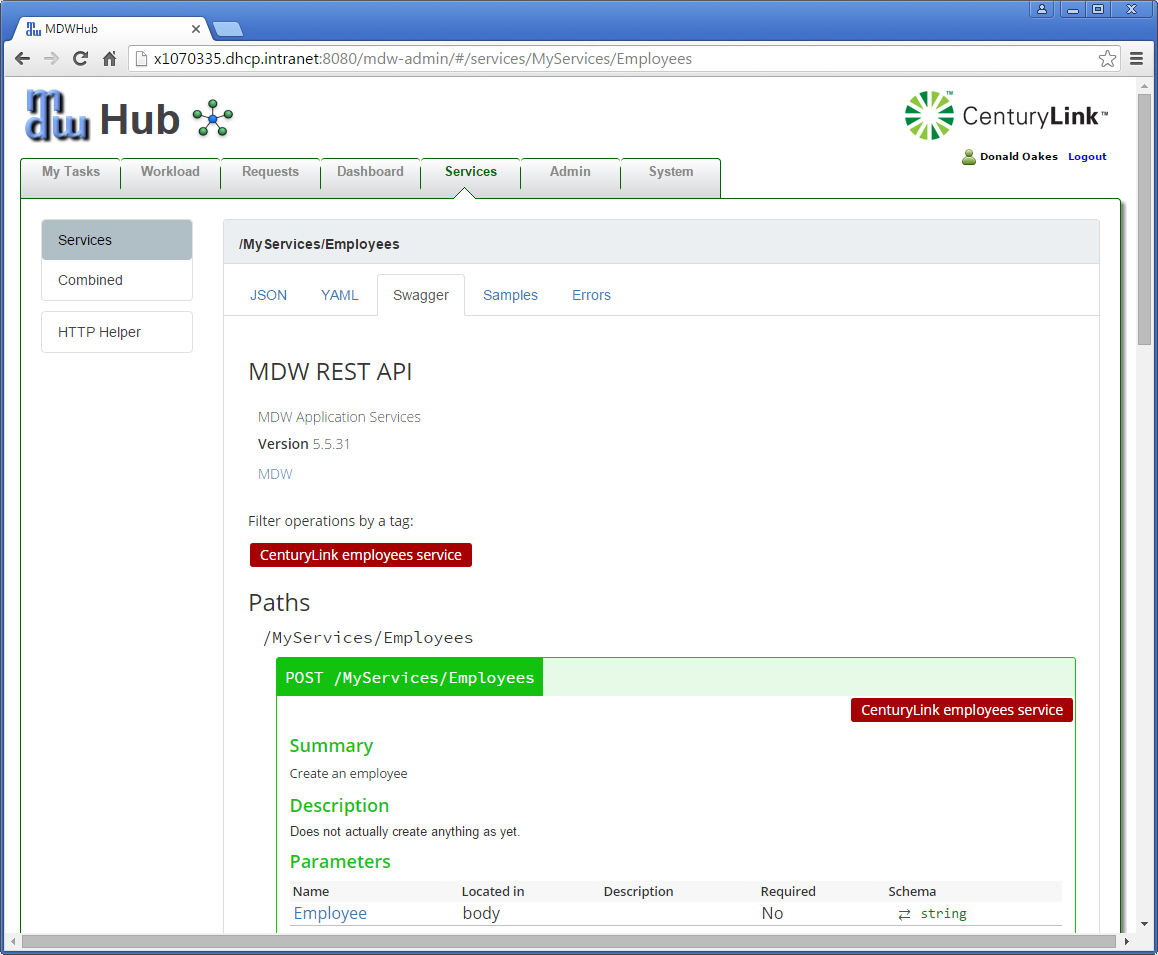
* Open MDW in your browser and click on the Services tab. Notice that API path for your service includes its package name to distinguish it from standard MDW services.



* Click on the /MyServices/Employees link. The JSON and YAML tabs include the [Swagger Spec](http://swagger.io/specification/) API definitions for the Employees endpoint. Click on the YAML tab to view a human-readable representation of your Employees API. Notice that much of the information is provided by annotations from the MDW base service class.



* Scroll down to the "definitions" section to see the Employee model object definition as well as other referenced types.
* Now click on the Swagger subtab to explore the friendly swagger-editor UI for your service.



**Add a Sample Request and Response:**

* Sample payloads in MDW are by convention kept in an asset package under the service package whose name ends with "api.samples". Each sample should be named to indicate its path and purpose, with an underscore separating these two parts. Create a new MDW package named "MyServices.api.samples" and add a JSON asset named Employees\_Get1.json with the following content:

// GET request to services/MyServices/Employees/dxoakes

{

"cuid": "dxoakes",

"name": "Donald Oakes",

"attributes": {

"Email": "donald.oakes@centurylink.com",

"Phone": "303 992 9747"

}

}

* Add another asset named Employees\_Create.json with content like this:

// POST request to services/MyServices/Employees

{

"cuid": "aa56486",

"name": "Manoj Agrawal",

"attributes": {

"Email": "manoj.agrawal@centurylink.com",

"Phone": "303 992 9980"

}

}

**View the Samples in MDWHub:**

* Now that your sample requests have been created in accordance with the MDW naming convention, they'll automatically be associated with the corresponding service path. And they'll also be displayed in the Samples tab for your service in MDWHub:

