Perceptive Connect Runtime

Developer's Guide

Version: 1.3.x

Date: August 2016

Lexmark™ Enterprise Software

© 2016 Lexmark. All rights reserved.

Lexmark is a trademark of Lexmark International, Inc., registered in the U.S. and/or other countries. All other trademarks are the property of their respective owners. No part of this publication may be reproduced, stored, or transmitted in any form without the prior written permission of Lexmark.

Table of Contents

About the Perceptive Connect Developer's Guide

This guide gives you the steps necessary to create a connector for the Perceptive Connect Runtime service. For more information about the runtime service, refer to the *Perceptive Connect Runtime Installation and Setup Guide*.

Prerequisites

Refer to the following list of prerequisites.

- Latest version of Perceptive Connect Runtime
- Java 8
- Eclipse

  Note This was most recently tested with Eclipse Luna SR1, Eclipse IDE for Java Developers.
- Maven, version 3.0.5

Terminology

Perceptive Connect Runtime runs on the OSGi framework. OSGi is a specification for creating modular Java applications. For an in-depth introduction to OSGi, we recommend the OSGi in Action: Creating Modular Applications in Java by Richard Hall. Below are some OSGi terms used throughout this document.

Bundle. An OSGi module. A bundle is a standard jar file whose manifest file contains additional metadata used by the OSGi runtime.

Bundle fragment. A bundle fragment is a bundle that shares its classloader with a host bundle. Thus, a fragment has access to any packages in the host, and vice-versa. A fragment also has the same lifecycle as its host. A consequence of this is that bundle activators, component descriptors, and other lifecycle "metadata" cannot exist within a fragment.

Component. An object whose lifecycle is managed by the OSGi runtime. A bundle may contain multiple components that may provide and consume multiple services. Each component is described by a component descriptor XML files that is included in a bundle.

Plug-in. Within the Eclipse Plug-in Development Environment (PDE), bundles are referred as plugins. This is because Eclipse itself runs on OSGi, and thus Eclipse plug-ins are also OSGi bundles.

Service. A service is any object that implements an interface that is registered with the Service Registry. Objects can obtain references to a service through the Service Registry. Perceptive Connect uses the Declarative Services component framework to manage service registration.

Set up the development environment

To set up your development environment, complete the following tasks.

- Optional. Import the standard preferences
- Install the plugins
- Add the archetypes

Import the standard preferences

To import the optional Perceptive Connect standard preferences, complete the following steps.

1. Get the preference file by cloning the following git repo: http://pswgithub.rds.lexmark.com/integrated-products/ipa.git.
2. Click **File > Import > General > Preferences**, and then click **Next**.
3. In the **From preference file** field, browse to the preference file located in the directory cloned from the git repo.
4. Select **Import All** and click **Finish**.

Install the plugins

To install the required plugins, complete the following steps.

Maven Integration for Eclipse

To install Maven Integration for Eclipse (m2e), complete the following steps.

Note m2e may already be installed, depending on your version of Eclipse.

1. Open **Eclipse** and click **Help > Install New Software**.
2. Set **Work with** to `http://download.eclipse.org/technology/m2e/releases/`and press Enter.
3. Select **Maven Integration for Eclipse** and click **Next**.
4. Finish the installation.

Setting Proxy for Maven in Eclipse

This is required for development on Lexmark's local network.

Add the following proxy configuration to the **settings.xml** file located in **%USERPROFILE%/.m2**.

```
<proxies>
    <proxy>
        <active>true</active>
        <protocol>http</protocol>
        <host>ps-auto.proxy.lexmark.com</host>
        <port>80</port>
        <nonProxyHosts>*.pvi.com</nonProxyHosts>
    </proxy>
</proxies>
```

If the file does not exist, create it with the following content.

```
<?xml version="1.0" encoding="utf-8"?>
<settings
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd"
        xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <proxies>
        <proxy>
            <active>true</active>
            <protocol>http</protocol>
```

```
            <host>ps-auto.proxy.lexmark.com</host>
            <port>80</port>
            <nonProxyHosts>*.pvi.com</nonProxyHosts>
        </proxy>
    </proxies>
</settings>
```

Configure Eclipse with the location of the **settings.xml** file.

1. Open **Eclipse** and click **Window > Preferences > User Settings**.
2. Expand **Maven** and under **User Settings**, provide a path for the maven settings file.

For more information, refer to the following websites.

- Maven Eclipse Plugin
- M2Eclipse
- M2Eclipse Downloads

Plug-in Development Environment

To install the Plug-in Development Environment (PDE), complete the following steps.

Note PDE may already be installed, depending on your version of Eclipse.

1. Open **Eclipse** and click **Help > Install New Software**.
2. Set **Work with** to the updated URL that corresponds to your version of Eclipse, such as http://download.eclipse.org/releases/luna or http://download.eclipse.org/releases/kepler.
3. In the **filter** field, type `plug-in development environment` and press Enter.
4. Under **General Purpose Tools**, select **Eclipse Plug-in Development Environment** and click **Next**.
5. Finish the installation.

For more information, refer to the Eclipse PDE website.

Tycho Configurator

To install the Tycho Configurator, complete the following steps.

1. Click **Window > Preferences**.
2. From the list on the left, select **Maven**.
3. Select **Discovery** and click **Open Catalog**.
4. In the **Find** field, type `tycho`
5. Select **TychoConfigurator** and click **Finish**.

   Note If tycho is not listed, use the manual procedure below.

To install the Tycho Configurator manually, complete the following steps.

1. Open **Eclipse** and click **Help > Install New Software**.
2. To set **Work with**, see https://repo1.maven.org/maven2/.m2e/connectors/m2eclipse-tycho/0.9.0/N/LATEST/
3. Under **m2e extensions**, select **Tycho Project Configurations** and click **Next**.
4. Finish the installation.

Add the archetypes

To add the archetypes, complete the following steps.

1. Click **Window > Preferences**.
2. From the list on the left, select **Maven**.
3. Select **Archetypes**.
4. Click **Add Remote Catalog**.
5. To set the **Catalog File** field, see http://repo.pcr.psft.co/nexus/content/repositories/releases/
6. Set the **Description** to a relevant name, such as **Connect Archetypes**.
7. Click **OK**.

Build a connector

This section contains information about creating, modifying, and deploying a new connector project with the following tasks.

- Create a new project
- Modify a project
- Configure and run the project in Eclipse
- Configure logging
- Deploy a connector to Perceptive Connect Runtime

Create a new project

Archetypes provide an easy way to get started with development in Perceptive Connect and provide basic implementation of several common use cases. The project layout convention is a main project that contains the launch file and target files. Under the main project, you have the supporting projects.

```
/some-project-main
         some-project-main.launch
         connect.target
         pom.xml
         /some-project-api
              ....
         /some-project-impl
             ....
         /some-project-impl-test
```

Create a new root project based on an archetype

To create a new root project based on an archetype, complete the following steps.

1. In **Eclipse**, click **File > New > Other**.
2. Select **Maven > Maven Project** and click **Next**.
3. Keep the default settings. Verify that the **Create a simple project** check box is cleared and click **Next**.
4. Set the **Catalog** to the name of the archetype catalog you created earlier, such as **Connect Archetypes**.
5. Verify that the **Show the last version of Archetype only** check box is selected.

   Note Archetype Artifact IDs were changed before the 1.0 release of Perceptive Connect, so *0.11.0* artifacts still appear in the list but can be ignored.
6. Select the archetype with a **Group Id** of **com.perceptivesoftware.connect.archetype** and **Artifact ID** of **pom-root**.
7. Click **Next**.
8. Enter a **Group Id**. We recommend using reverse domain notation, for example `com.mycompany.myproduct`.
9. Enter an **Artifact Id**. This will become the name of the project. We recommend using a format of `<connector-name>-main`.
10. Click **Finish**.

    Note Your new project may show Eclipse errors until you update the eclipse target.

Create a new module under the root project

To create a new module under the root project, complete the following steps.

1. Click **File > New > Other**.
2. Select **Maven > Maven Module** and click **Next**.
3. Enter a project name in the **Module Name** field.
4. For the **Parent Project** field, click **Browse** and select the root project that you previously created.
5. Click **Next**.
6. Set the **Catalog** to the name of the archetype catalog you created earlier, such as, **Connect Archetypes**.
7. Select the base archetype with which you would like to begin. More information on the archetypes can be found in the About archetypes section.
8. Fill out any required fields and click **Finish**. See the About archetypes section for information on required fields.

   Note The **Finish** button may not activate until you click off any required fields in the **Eclipse** dialog box.

Update the target

The eclipse `Target Platform` determines which plug-ins your connector will be built and run against. The archetype's `Target Platform` matches the components available in the Perceptive Connect Runtime.

To update the project target, complete the following steps.

1. In the root project, right-click the **TARGET** file and select **Open With > Target Editor**.
2. In the top right corner of the screen, click **Set as Target Platform**. The operation may take a few minutes to complete.
3. Setup is complete. Close the target file.

Modify a project

This section describes how to modify a project by adding a dependence, a referenced service, or a provided service.

Add a dependency

To modify a project by adding a dependency, complete the following steps.

1. In the **Package Explorer**, open the **MANIFEST.MF** file.
2. At the bottom of the screen, click the **Dependencies** tab.
3. In the **Imported Packages** section, click **Add**.
4. Select the required package and click **OK**.

Note The **Required Plug-ins** option should be avoided except in integration test projects. See the Use Import-Package instead of Require-Bundle topic in the OSGi wiki for more information.

If the package you need is not available in the list, you can add a bundle that exports the package using the following steps.

1. Download the required bundle.
2. Right-click your **TARGET** file and click **Open With > Target Editor**.
3. In the **Locations** section, on the **Definitions** tab, click **Add**.
4. Select **Directory** and click **Next**.
5. Browse to the location of the bundle and click **Finish**.
6. Open the **Content** tab.
7. In the **Content** section, search the list for the required bundle, such as **org.apache.commons.codec** and verify that the check box is selected.
8. Save and close the target file.

Add a referenced service

To modify a project by adding a referenced service, complete the following steps.

1. In the **Package Explorer**, right-click the XML file located in the **OSGI-INF** folder.
2. Click **Open With > Component Definition Editor**.
3. Click the **Services** tab located in the bottom left corner of the window.
4. Under **References Services**, click **Add**.
5. Find the service type you want to add and click **OK**.
6. Select the new service and click **Edit**.
7. Set the **Cardinality**.
   - **0..1** – An optional singular service.
   - **1..1** – A required singular service.
   - **0..n** – An optional multiple service
   - **1..n** – A required multiple service
8. Set the **Policy**.
   - **static**. The component is deactivated and a new instance is created whenever the service is switched.
   - **dynamic**. The component instance remains the same whenever the service is switched.

     Note The **dynamic** policy provides better performance and should be preferred. The **static** policy is the default policy.
9. Add values to the **Bind** and **Unbind** fields.

   Note These values serve as the method names that receive service references. See the Code Samples below.
10. Click **OK**.
11. Click the **Overview** tab located in the bottom left corner of the window.
12. Click the **Class*:** hyperlink under **Component**.
13. Add a reference and bind and unbind methods for the new service, where the method names are equal to the values you previously provided for **Bind** and **Unbind**. See the Code Samples below.
14. Save the file.

Code Samples

- A singular service reference.
  - Assume that **Bind** was set to `bind`, and **Unbind** was set to `unbind` in the previous procedure.

```
private ActionManager manager;

public void bind(ActionManager newManager) {
    manager = newManager;
}

public void unbind(ActionManager oldManager) {
    manager = null;
}
```

- A multiple service reference.
  - Assume that **Bind** was set to `addAction`, and **Unbind** was set to `removeAction` in the previous procedure.

```
private final List<Action> actions = new ArrayList<>();

public void addAction(Action action) {
    actions.add(action);
}

public void removeAction(Action action) {
    actions.remove(action);
}
```

Add a provided service

To add a provided service, complete the following steps.

1. In the **Package Explorer**, in the **OSGI-INF** folder, right-click the XML file.
2. Click **Open With > Component Definition Editor**.
3. Click the **Services** tab located in the bottom left corner of the window.
4. Under Prov**ided Services**, click **Add**.
5. Find the service type you want to add and click **OK**.

   Note The implementation class for this component must provide the interface, or implement the class that you select in the list. It is a best practice to provide interfaces for services.
6. Save the file.

Build a Trust Validator

A Trust Validator can be used to ensure that only authenticated users can access sensitive information through REST/SOAP endpoints. In this way, consumers of these endpoints are forced to provide specific headers for authentication when they make their request. The **RESTTrustValidator** and **SoapTrustValidator** interfaces are used to enforce authentication on REST and SOAP endpoints respectively. An implementer of either interface must then publish these interfaces as provided services through their component definitions.

The RESTTrustValidator interface contains the following methods.

- `Ojbect getToken(ContainerRequestContext context)` throws `TrustRequiredException`. This method is responsible for pulling out and returning the appropriate token from the `ContainerRequestContext`.
- `void validateToken(Object token)`throws `InvalidTrustTokenException`. This method is responsible for actually validating the token. When the token is valid, there is "trust." When the token is invalid, an exception should be thrown.
- `Duration getTimeout()` This method is responsible for setting the interval in which a token is considered valid after a successful call to `validateToken`.

The SOAPTrustValidator interface contains the following methods.

- `Object getToken(SOAPMessageContext context)` throws `TrustRequiredException`. This method is responsible for pulling out and returning the appropriate token from the `SOAPMessageContext`.
- `void validateToken(Object token)` throws `InvalidTrustTokenException` This method is responsible for actually validating the token. When the token is valid, there is "trust." When the token is invalid, an exception should be thrown.
- `Duration getTimeout()` This method is responsible for setting the interval in which a token is considered valid after a successful call to `validateToken`.

To create a Trust Validator service, complete the following steps.

1. Depending on whether your validator will provide SOAP, REST, or both types of web services, your validator class should implement **SOAPTrustValidator**, **RESTTrustValidator**, or both.

```
public class SampleTrustValidator implements RESTTrustValidator, SOAPTrustValidator
{
    public static final String TRUST_HEADER_NAME = "X-Fake-Trust-Header";
    public static final String TRUST_HEADER_VALID = "gooduser";

    @Override
    public Object getToken(SOAPMessageContext context) throws TrustRequiredException
{
        Map<String, List<String>> headers = (Map<String, List<String>>)
context.get(MessageContext.HTTP_REQUEST_HEADERS);
        List<String> trustHeaders = headers.get(TRUST_HEADER_NAME);

        if (trustHeaders == null || trustHeaders.size() != 1) {
            throw new TrustRequiredException(String.format("Failed to find '%s'
header in message.", TRUST_HEADER_NAME));
        }
```

```
        return trustHeaders.get(0);
    }

    @Override
    public Object getToken(ContainerRequestContext context) throws
TrustRequiredException {
        String headerValue = context.getHeaderString(TRUST_HEADER_NAME);
        if (headerValue == null) {
            throw new TrustRequiredException(String.format("Failed to find the '%s'
header in message.", TRUST_HEADER_NAME));
        }

        return headerValue;
    }

    @Override
    public void validateToken(Object token) throws InvalidTrustTokenException {
        if (!token.toString()
                .equals(TRUST_HEADER_VALID)) {
            throw new InvalidTrustTokenException(String.format("The '%s' token was
not valid.", TRUST_HEADER_NAME));
        }
    }

    @Override
    public Duration getTimeout() {
        return new Duration(1, TimeUnit.MINUTES);
  }

}
```

2. Create a Service Component File for the validator you implemented. Set **SOAPTrustValidator**, **RESTTrustValidator**, or both as provided services.

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="Sample Trust
Validator">
   <implementation
class="com.perceptivesoftware.pif.common.trust.SampleTrustValidator"/>
   <service>
      <provide
interface="com.perceptivesoftware.pif.common.trust.RESTTrustValidator"/>
      <provide
interface="com.perceptivesoftware.pif.common.trust.SOAPTrustValidator"/>
   </service>
</scr:component>
```

To use a validator on new SOAP and REST endpoints, complete the following steps.

1. Build your web service using the **pif-jaxrs-endpoint-archetype**, **pif-jaxws-endpoint-archetype**.
2. In the component definition for the RESTComponent/SOAPComponent class, list **RESTValidationFilter** or **SOAPValidationFilter** as referenced services and specify the bind and unbind fields to these services a `bind` and `unbind`, respectively. These services become available when the validator services component you just created is running.
3. To depend on a specific validation service, add the following information to the `target` field:

   (validatorName=<*fully qualified name*>) where <*fully qualified name*> is the fully qualified name of the validator service.

   Example

   `com.perceptivesoftware.pif.common.trust.SampleTrustValidator.`

To update REST/SOAP endpoints on an existing connector, refer to Adding Validation Filters to existing web services.

Enable automatic connector for upgrade handling

Perceptive Connect Runtime provides the `nstallService` interface that represents an "upgrade hook" that may be implemented to receive messages about bundle updates. The implemented service receives an instance of `UpdateInfo` when a new version of the containing bundle is installed in Perceptive Connect Runtime. `UpdateInfo` contains the original bundle version, the new bundle version, and the time the bundle was upgraded. A connector may find it useful to provide an `InstallService` implementation so that post-upgrade processes may be performed, such as upgrading channel mappings, updating database records, and so on.

To provide an implementation of InstallService, complete the following steps.

1. Create a class that extends `InstallService`.
2. Override the `updated (updateInfo info)` method.
3. Implement any upgrade logic that you want to perform.
4. Create a Component Definition that lists your class as providing the interface for the `InstallService`.

```
public class InstallServiceImpl extends InstallService {
    @Override
    public void updated(UpdateInfo info) {
        //implement upgrade logic
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="PIF Sample
Connector Update Service">
   <implementation class="com.acme.connector.InstallServiceImpl"/>
   <service>
      <provide interface="com.perceptivesoftware.pif.common.bundle.InstallService"/>
   </service>
</scr:component>
```

Configure and run the project in Eclipse

Run and debug in Eclipse

To run or debug the project in Eclipse, complete one of the following steps.

- To run your connector from Eclipse, right-click the **LAUNCH** file and click **Run As > [*project name*].** This starts an OSGi runtime in which all of the required Perceptive Connect bundles are installed, as well as your connector bundles.
- To debug your connector from Eclipse, right-click the **LAUNCH** file and click **Debug As > [*project name*]**.

Modify the run configuration

To modify the run configuration, complete the following steps.

1. In the **Package Explorer**, right-click the **LAUNCH** file.
2. Click **Run As > Run Configurations**.
3. Modify the run configuration with the options on the following tabs.
    - **Bundles** allows you to include and exclude bundles from the OSGi container deployed by eclipse.
    - **Arguments** allows you to modify VM arguments. The working directory for Eclipse and the listening port can be set in here.

      Note You can change the default tcp port that the web console listens on by modifying `-Dorg.osgi.service.http.port=.`
    - **Settings** is where the JRE is set. Under the **Configuration Area**, there is the option to **Clear** the configuration area before launching. By clearing this check box, Eclipse preserves channel mappings and any settings that are changed in the admin console.

Important You should use caution when changing the Run Configuration. It is easy to include libraries that will not be available in the PCR Runtime Distribution, which will break your connector when it is run in the stand-alone PCR engine.

Log and database file used in the default launch config

By default, your PIF log and database files are located in the pcr-debug directory under your Eclipse project's Workspace. To change the location of these files, complete the following step.

- Modify your launch file's `Arguments` (refer to the **Eclipse** tab in **Debug Configurations**) and change `–Dpcr.db.server.type="h2" –Dpcr.db.databasename=$-{workspace_loc}/pcr-debug/PCRDebugDatabase` as desired.

Configure logging

You can use log files to interpret issues encountered when running Perceptive Connect Runtime. A system administrator or developer enables logging, sets the logging state, and specifies the log file location. The Perceptive Connect Runtime Service must be running to enable logging.

Configure logging in Perceptive Connect

To configure logging, complete the following steps.

1. In the **Perceptive Connect Web Console**, on the **Configuration** tab, select **View Configuration**. In the **Name** column, locate the PIF Logger row.
2. Click the **Edit** button to open the **PIF Logger Configuration** menu.
3. Input the values for the following logging parameters.
    1. **Log Level**. The level of detail output when exceptions occur (**pif.log.level**). For more information, refer to Logging states.
    2. **Log Directory**. The path location of the PIF log file (**pif.log**.**directory**).
    3. **Logger Name**. The name of the logger in the system being configured (**pif.log.name**).
4. Click **Save**.

About the Perceptive Connect log files

Log files are located in [*drive:*]/{*install path*}/PIF/logs/ folder.

- **pifservice-stderr.[date].log** logs errors.
- **pifservice-stdout.[date].log** logs the standard output.
- **pif.all.log** combines entries from the **pifservice-stderr** and **pifservice-stdout** log files.
- **commons-daemon.[date].log** logs the Perceptive Connect Runtime launch process.

About the logging states

You want to set minimal logging, Level 0, to capture only critical exceptions, unless you are debugging an issue. A verbose state, such as Level 4, can generate large log files that affect system performance and hard disk space.

- **Error (Level 0)**. This state records runtime errors or unexpected conditions about the current operation, such as
    - Assertion failures
    - Network connection problems
    - Issues with retrieving valid authentication tokens
- **Warning (Level 1).** This state records events forewarning potential problems, such as
    - A non-secure data access connection
    - A data access implementation failure
    - Performance issues

- **Info (Level 2).** This state writes data to the log file as part of the normal operational flow of the service, such as
  - Startup and shutdown
  - Normal timers
  - Workflow events
- **Debug (Level 3).** This state reveals diagnostic details often useful for debugging.
- **Trace/All (Level 4).** This state writes all log messages to the log file. Typically, this includes even more verbose details for debugging.

Deploy a connector to Perceptive Connect Runtime

By default, Eclipse continuously compiles your project and allows you to run your connector within an eclipse debug session. However, to fully compile your connector for deployment to a stand-alone Perceptive Connect Runtime, you must execute a maven build.

To build your connector through Eclipse's m2e maven plugin, complete the following steps.

1. Right-click your root maven project and select **Maven > Update Project**.
2. Right-click your root maven project and select **Run As > Maven Install**.

To build your connector through Stand-alone Maven, complete the following steps.

1. Ensure that your `M2_HOME` and `JAVE_HOME` environmental variables are correct.
2. In a cygwin, dos, or unix shell, navigate to your root project's directory.
3. Execute `mvn clean install`.

Build the connector bundles for deployment

Install the Connector

To install a connector, complete the following steps.

1. In the **Perceptive Connect Runtime Dashboard**, click **Install a Connector**.
2. On the **Bundle Management** page, there is a column on the left side of the page that says **DRAG FILES HERE**. Drag your project JAR, ZIP and PCR files.
3. Perceptive Connect displays the installation results on the right side of the page.

Verify the connector installation

To verify connector installed correctly, complete the following steps.

1. In **Perceptive Connect Runtime Web Console**, click **Perceptive Connect > View Bundles**.
2. On the **Perceptive Connect Runtime Web Console Bundles**, there is a list of installed bundles. Verify that the connector bundles you installed are in the **Active** state. If they are not, click the bundle's **Start** button in the **Action** column.

Debugging the connector

To debug a connector that is deployed to a running instance of Perceptive Connect Runtime, perform the following steps.

1. Run the **PerceptiveConnectRuntimew.exe** file.
2. Click the **Java** tab.
3. Insert the following information in the **Java Options** input.
   - `-agentlib:jdwp=transport=dt_socket,sever=y,suspend=n, address=6006`.

     Note You can modify the *suspend* and *address* argument to your needs. A *suspend argument* of **y** causes the service to wait for the debugger to attach before completing the startup process. The *address* argument defines the port on which debugging is allowed.
4. Click the **Apply** button.
5. If the service is already running, restart the service.

To connect the Eclipse debugger to the running instance of Perceptive Connect Runtime, perform the following steps.

1. On the **Eclipse** toolbar, click **Run > Debug Configurations**.
2. Right-click **Remote Java Application** and click **New** on the resulting context menu.
3. On the right side of the screen, name the debug configuration in the **Name** field.
4. In the **Project** box, click **Browse** and then select the connector project to debug.
5. In the **Connection Properties** field, fill in the **Host** and **Port** fields.
   - In the **Host** field, input the IP Address of the machine on which Perceptive Connect Runtime is running, or *localhost* if you are debugging on the same machine.
   - In the **Port** field, provide the port number that you configured to allow debugging. This value is the argument for the *address* parameter entered into the **Java Options** field of the service executable.

Connector Development Tips

This section contains tips, tricks, and best practices that connector developers have defined to support your efforts.

Readers and Writers

`ParameterSource` and `ParameterTarget` types must be visible to JAXB to enable XML serialization and deserialization. To be visible, the class definitions need to be public or static.

Adding Validation Filters to existing web services

As of PCR 1.2, connector developers have the ability to add validation filters to their web services. For new connectors, archetypes have been updated accordingly. However, existing connectors can be modified to add the following new functionality.

Existing REST Services

1. Your RESTComponent should extend `AbstractRESTComponent` rather than `AbstractLifecylceComponent`. If your REST Service already extends `AbstractRESTComponent`, go to the next step. If not, complete the following substeps.
    1. If the REST service was created using a pre-1.3 archetype, remove the declaration of the JAXRSService in your RESTComponent as it is already declared in the AbstractRESTComponent class.
    2. In the same class, modify the startup method by changing service.registerApplication to registerApplication.
    3. In the same class, remove the methods registerServlet and unregisterServlet. Additionally, you can remove the shutdown method.
    4. In the component definition to your RESTComponent, update the service reference to the JAXRSService to call bind and unbind rather than registerServlet and unregisterServlet.
2. In the component definition to your RESTComponent, you must add the reference to the REST filter service `RestValidationFilter`.
3. To require a specific validation service, you may add the following to the `target` attribute of the service reference: `(validatorName=<`*`fully qualified service name`*`>)`, where `<`*`fully qualified service name`*`>` is the fully qualified name of the service. For example, `com.perceptivesoftware.imagenow.service.validator.ImageNowValidator`.

    Note Version 1.1 of Content Connector includes and ImageNow Validator service that extracts a `sessionHash` cookie from a REST request and forwards the session hash on to the ImageNow service for validation. The ImageNow Validator service can be used to require validation with ImageNow on your REST endpoints. For more information on the ImageNow Validator service, consult the *Content Connector Install Guide*.

Existing SOAP Services

1. Create a local variable `validationFilter` of type `SOAPValidationFilter` so that you can store the filter reference. Add in additional bind() and unbind() methods so your component can reference the validation filter if it exists.

```
    validationFilter = newValidationFilter;
}

public void unbind(SOAPValidationFilter oldValidationFilter) {
    validationFilter = null;
}
```

2. The call to **registerEndpoint** in the startup by adding the **validationFilter** as an additional argument as shown in the following example.

```
public void startup() throws Exception {
       ...
       service.registerEndpoint(ALIAS, endpoint, validationFilter);
       ...
   }
```

3. In the component definition to your SOAPComponent, you must add the reference to the SOAP filter service `SOAPValidationFilter`.

About file descriptions

The following section defines file descriptions used in the Runtime Connector process.

Launch files. The launch file contains the information needed for your Eclipse run and debug configuration, including VM arguments, JRE settings, and bundles to install in the runtime service.

Target files. Refer to the Target Platform topic in the Eclipse documentation.

POM files. Refer to this Maven document for more information.

Manifest files. In OSGi, the `metadata` about how to run a bundle (which is a JAR with OSGi metadata) is contained in the manifest, **MANIFEST.MF**.file This file is located in **./META-INF/** within a JAR, as well as in any Eclipse PDE project.

Component descriptor files. Component descriptors are XML files located in the **OSGI-INF/** directory of an OSGI bundle. These files are used by the Declarative Services component framework. A component descriptor contains the name of the class providing the component implementation, as well as a list of services that the component provides and services to which that component is bound.

About archetypes

Maven Archetype is a templating tool for creating Java projects. Several Maven archetypes are provided to ease connector development. A brief description of each archetype and its configuration parameters are included in the following sections.

General configuration

Every Maven artifact is uniquely identified by a **Group Id**, **Artifact Id**, and **Version**. When creating a new Maven project or Maven module in Eclipse, there are configured after selecting an archetype. In general, the **Group Id** and **Version** should be consistent for all projects in a connector, should be defined in the root project, and will be inherited by all child projects. We recommend a Group Id that follows standard Java package naming conventions (for example, `com.mydomain.myconnector`).

Additionally, there is a default Package associated with the project. This can be ignored for the root project. For modules, this is the package in which generated source code is placed. Package names should be consistent with the `Group ID` (for example, `com.mydomain.myconnector.endpoints`). A module may contain any number of packages. This setting is only used when generating archetype source files.

Archetype descriptions

Several Maven archetypes are provided to ease connector development.

pif-jaxrs-endpoint-archetype - Creates a simple JAX-RS web service with a single "Hello World" REST endpoint.

Parameters

- **endpoint** The name of the class providing the endpoint, the URL alias of the endpoint, and the name of the component providing the web service.

pif-jaxws-endpoint-archetype - Creates a simple JAX-WS web service with a single "Hello World" SOAP method.

Parameters

- **endpoint** The name of the class providing the method, the URL alias of the endpoint, and the name of the component providing the web service.

pie-trigger-archetype - Creates a simple PIE Trigger service that has a single identifier, and a few output parameters.

Parameters

- **connector** The "component.group" property in the trigger's component descriptor.
- **trigger** The name of the trigger including its class name, and the name of the component providing the `Trigger` service.

pie-simple-connector-archetype - Creates a simple PIE Action service that has a single input and output parameter.

Parameters

- **action** The name of the action including its class name, and the name of the component providing the `Action` service.
- **connector** The "component.group" property in the trigger's component descriptor and the bundle-name property in the manifest file.

pie-configurable-connector-archetype - The OSGi Metatype spec provides a way for services to be configured at runtime. Currently, the Connect Runtime uses the Felix web console to provide a UI for Metatype configuration. At the service level, the metatype is described through an XML file in `OSGI-INF/metatype/.` This archetype creates a project containing a metatype definition for a configurable action. For a brief introduction to metatypes, we suggest this article.

Parameters

- **action** The name of the action including its class name, and the name of the component providing the `Action` service.
- **connector** The "component.group" property in the trigger's component descriptor and the bundle-name property in the manifest file.

pie-connector-unittest-archetype - Creates a **bundle fragment** project containing unit tests for an Action within the host project. The test class in this file is designed to test an Action, so this should be paired with either the **pie-simple-connector-archetype** or **pie-configurable-connector-archetype** for a working example.

Parameters

- **connector** The bundle-name property in the manifest file.
- **action** The name of the action under test.
- **host_artifactId** The artifact id of the bundle under test.

pie-connector-integration-test-archetype - Creates a bundle that provides basic integration test functionality. Test fixtures and a launch file are included so the developer can focus on quickly writing tests.

Parameters

- **connector-name** The name of the connector that the integration test bundle will be testing. This value will be included in the test bundle's manifest.

imagenow-connector-archetype - Creates a simple PIE Action service that does not have any input or output parameters defined. The action also has a service reference to the `ImageNowEndpoint` in Content Connector. This archetype should be used when creating a connector that requires direct communication with ImageNow.

Parameters

- **action** The name of the action including its class name, and the name of the component providing the `Action` service.
- **connector** The "component.group" property in the trigger's component descriptor and the bundle-name property in the manifest file.

connect-assembly-archetype - Creates a maven project that inserts project module artifacts into a zip file. By default, the only module artifacts included in the zip file are jar files. Add each module of your project as a dependency in the pom of the assembly archetype project to include the modules' artifacts in the zip archive. The zip archive is especially useful for connectors that are composed of multiple bundles. The single zip archive can be distributed to customers and installed through the Connector Installer UI of Perceptive Connect Runtime.

Frequently Asked Questions

This section contains the answers to questions that either have an unexpected answer or that we have encountered commonly when working with connector developers.

I am attempting to use packages in the "pif.test_utils" bundle, and am seeing error messages such as "Package uses conflict: Import-Package: org.apache.http.impl.auth; version="4.3.3"" at runtime. What is going on?

Typically, the uses conflict means that there is a version incompatibility between different bundles importing the same package. See this article for an in-depth explanation. However, we encountered this specific error message when the "pif.cfx" bundle was not being imported into the project. The OSGI dependency resolver sometimes returns unexpected error messages, and this was one of those instances.

I am encountering `Java.lang.NoClassDefFoundError` or `java.lang.ClassNotFoundException` at runtime. How do I resolve these error messages?

Unfortunately, this is a fairly common issue with OSGI applications. As with any ordinary Java application, these exceptions mean that the class could not be found or could not be instantiated by the classloader. A goal of bundle package exports is to avoid this type of run-time error, and replace it with an unresolved bundle, which is easier to troubleshoot. Typically, this problem is encountered with platform-dependent classes that are provided by the Java runtime (such as sun.* or javax.* packages). By default, these packages are blocked by the Felix classloader. The reasoning developer, this is an annoyance as the error is only exposed at runtime through `NoClassDefFoundError/ClassNotFoudnException`.

We are investigating a better way for resolving these issues, but the current solution is as follows.

1. Add the problem package as an optional import to your manifest.
    1. In Eclipse, open the **Manifest** file, and click the **Dependencies** tab.
    2. Under **Imported Packages**, click **Add*** and add the required package. In the example above, this would be **javax.imageio.metadata**.
    3. Select the newly added package, and click **Properties**.
    4. Select the **Optional** check box and click **OK**.
    5. Save the **Manifest** file, and you should now see something like **javax.imageio.metadata;resolution:=optional**, if you view the final file in a test editor.
2. Add the package to the runtime configuration.

    Note This step must be done in the installed PCR instance. In other words, anyone who installs your connector needs to perform this step after connector installation.

    1. Open the **PCR Install Directory/conf/config.properties** for currently installed runtime.
    2. Add the new package (**example.javax.imageio.metadata)** to the **org.osgi.framework.system.packages.extra** setting.
    3. Save the file. You may also need to restart the runtime.

Troubleshooting this type of issue with system packages typically involves multiple iterations of the above steps. The reason is that resolving one exception might expose another later in the code path.