

Perceptive Connect Runtime

Developer's Guide

Version: 1.5.x

Written by: Product Knowledge, R&D

Date: Friday, December 23, 2016

© 2014-2016 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

Prerequisites	8
Terminology	8
Set up the development environment	9
Import the standard preferences (optional)	9
Install the plugins	9
<i>Maven Integration for Eclipse</i>	9
<i>Plug-in Development Environment</i>	10
<i>Tycho Configurator</i>	11
Add the archetypes	11
Build a connector	12
Create a new project	12
<i>Create a new root project based on an archetype</i>	12
<i>Create a new module under the root project</i>	13
<i>Update the target</i>	13
Modify a project	13
<i>Add a dependency</i>	13
<i>Add a referenced service</i>	14
Code Samples	15
<i>Add a provided service</i>	15
<i>Creating REST Endpoints / Handling CORS Requests</i>	16
<i>Building a Trust Validator</i>	16
Trust Validation Diagrams	16
Trust validation, step 1	17
Trust validation, step 2	18
Trust Validator Interfaces	19
<i>Using a Trust Validator</i>	21
Using Trust Validators with CORS and/or SSL	22
<i>Enable automatic connector upgrade handling</i>	22
Configure and run the project in Eclipse	23

<i>Run and debug in Eclipse</i>	23
<i>Modify the run configuration</i>	23
<i>Log and database file used in the default launch config</i>	23
Configure logging	24
<i>Configure logging in Perceptive Connect</i>	24
<i>About the Perceptive Connect log files</i>	24
<i>About the logging states</i>	24
Deploy the connector to Perceptive Connect Runtime	25
<i>Build the connector bundles for deployment</i>	25
<i>Install the connector</i>	25
<i>Verify the connector installation</i>	26
<i>Debugging the connector</i>	26
Core Concepts	27
Scheduler	27
<i>Scheduler Source Based Conflict Resolution</i>	27
Job	27
<i>JobDetail</i>	28
<i>DetailedJob</i>	28
JobTrigger	28
Schedule	28
<i>CronSchedule</i>	29
<i>SimpleSchedule</i>	29
Using the Connect Scheduler	29
Using the API	30
<i>Scheduler Service</i>	30
<i>Builder classes</i>	30
Using OSGi Services	30
<i>CronScheduleJobTrigger</i>	30
<i>AbstractDetailedJob</i>	31
Connector Development Tips	31
Connector Names and Bundle names	31

Connector Name	31
Bundle Name	31
Example	32
Readers and Writers	32
Adding Validation Filters to existing web services	32
For existing REST Services:	32
For existing SOAP Services:	33
About file descriptions	33
Launch files	33
Target files	33
POM files	33
Manifest files	33
Component descriptor files	34
About archetypes	34
General configuration	34
Archetype descriptions	34
pif-jaxrs-endpoint-archetype	34
Parameters	34
pif-jaxws-endpoint-archetype	34
Parameters	34
pie-trigger-archetype	35
Parameters	35
pie-simple-connector-archetype	35
Parameters	35
pie-configurable-connector-archetype	35
Parameters	35
pie-connector-unittest-archetype	35
Parameters	36
pie-connector-integration-test-archetype	36
Parameters	36
imagenow-connector-archetype	36

Parameters	36
<i>connect-assembly-archetype</i>	36
What is CORS	37
Terms	37
Detailed Explanation	37
CORS Requests with Secure Data	37
<i>What CORS Cannot Do</i>	38
CORS, PCR, and Your Application(s)	38
Passing secure data from Perceptive Experience to PCR	39
<i>Example 0</i>	40
System layout	40
Sequence diagram	40
<i>Example 1</i>	41
System layout	41
Sequence diagram	41
<i>Example 2</i>	42
System layout	42
Sequence diagram	42
<i>Example 3</i>	43
System layout	43
Sequence diagram	44
<i>Example 4 (Hypothetical)</i>	44
System layout	44
Sequence diagram	45
CORS with Secure Cookies	45
Making CORS Requests from Your Web Application	46
<i>Using Pure Javascript</i>	46
<i>Using jQuery</i>	46
<i>Passing Secure Data from Your Web Application to PCR</i>	46
Handling CORS Requests Made to Your Connector's Endpoint	47

<i>Using PCR Global CORS Settings</i>	47
<i>Using JAX-RS Annotations</i>	47
<i>Explicitly Setting CORS Headers</i>	48
<i>Configuring or Overriding CORS in Either Situation</i>	49
<i>Setting Browser Cookies from Your Endpoint</i>	49
Appendix: CORS Configuration for Web Servers	49
<i>Configuring CORS in Tomcat</i>	49
<i>Other Web Servers</i>	50
Frequently Asked Questions	50
<i>Q. 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?</i>	50
<i>Q. I am encountering Java.lang.NoClassDefFoundError or java.lang.ClassNotFoundException at runtime. How do I resolve these error messages?</i>	51

Prerequisites

- Latest version of [Perceptive Connect Runtime](#)
- [Java 8](#)
- [Eclipse](#)
Note This was most recently tested with [Eclipse Mars R](#), [Eclipse IDE for Java Developers](#).
- [Maven](#)

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 book [OSGi in Action](#) by Richard Hall. Below are a 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.
- **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.
- **Component.** An object whose lifecycle is managed by the OSGi runtime. A bundle may contain multiple components which may provide and consume multiple services. Each component is described by a [component descriptor XML](#) file 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.
- **Bundle fragment** A bundle fragment is a bundle that shares it's 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 it's host. A consequence of this is that bundle activators, component descriptors, and other lifecycle "metadata" cannot exist within a fragment.

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 (optional)

To import the 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>

```

Now 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 path for the maven settings file

For more information, refer to the following websites.

- <http://maven.apache.org/eclipse-plugin.html>
- <http://www.eclipse.org/m2e/>
- <http://www.eclipse.org/m2e/download/>

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 update 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 <http://www.eclipse.org/pde/>.

Tycho Configurator

To install the Tycho Configurator via `Discover m2e Connectors`, 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 **Tycho Configurator** and click **Finish**.

Note If tycho isn't listed use the manual step below

To install the Tycho Configurator manually, complete the following steps

1. Open **Eclipse** and click **Help > Install New Software...**
2. Set **Work with** to `http://repo1.maven.org/maven2/.m2e/connectors/m2eclipse-tycho/0.8.0/N/LATEST/`
3. Under **m2e extensions** select **Tycho Project Configurators** 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. Set the **Catalog File** field to `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

In this section, you learn how to create, modify, and deploy 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 implementations 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 will still appear in the list. These can be safely 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 that you would like to begin with. More information on the archetypes can be found in [Archetypes](#).
8. Fill out any required fields and click **Finish**. See the [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

Update the target

The eclipse `Target Platform` determines which plug-ins your connector will be built and ran 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 dependency, a referenced service, or a provided service.

Add a dependency

To add 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 that package with the following steps:

1. Download the required bundle.
2. Right click on 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 its box is checked.
8. Save and close the target file.

Add a referenced service

To add a referenced service, complete the following steps.

1. In the **Package Explorer**, right-click the XML file located in the **OSGI-INF** folder from the Package Explorer.
2. Click **Open With > Component Definition Editor**.
3. Click the **Services** tab located in the bottom left corner of the window.
4. Under **Referenced 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 will serve as the method names that receive service references. See the **Code Samples** below.
 10. Click **OK**.
 11. Click on the **Overview** tab located in the bottom left corner of the window.
 12. Click on 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 that you chose in step 9.
 - See below for code samples.
 14. Save the file.

Code Samples

- A singular service reference.
 - Assume that **Bind** was set to `bind`, and **Unbind** was set to `unbind` in step 9.


```
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 step 9.


```
private final List 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 **Provided 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.

Creating REST Endpoints / Handling CORS Requests

If your connector includes any REST Endpoints, we strongly recommend that your REST component(s) extend either `com.perceptivesoftware.pif.util.AbstractRESTComponent` or `com.perceptivesoftware.pif.util.AbstractSecureRESTComponent`. These classes, among other things, automatically apply CORS filters to your endpoints as configured by the PCR administrator. If you do not extend one of these classes, such as if your component extends `com.perceptivesoftware.pif.util.AbstractLifecycleComponent`, you will need to manually set up a CORS filter for your endpoint.

Note: If you created your endpoint using the Connect Archetype from 1.3.x or later, you should be using one of the abstract classes already.

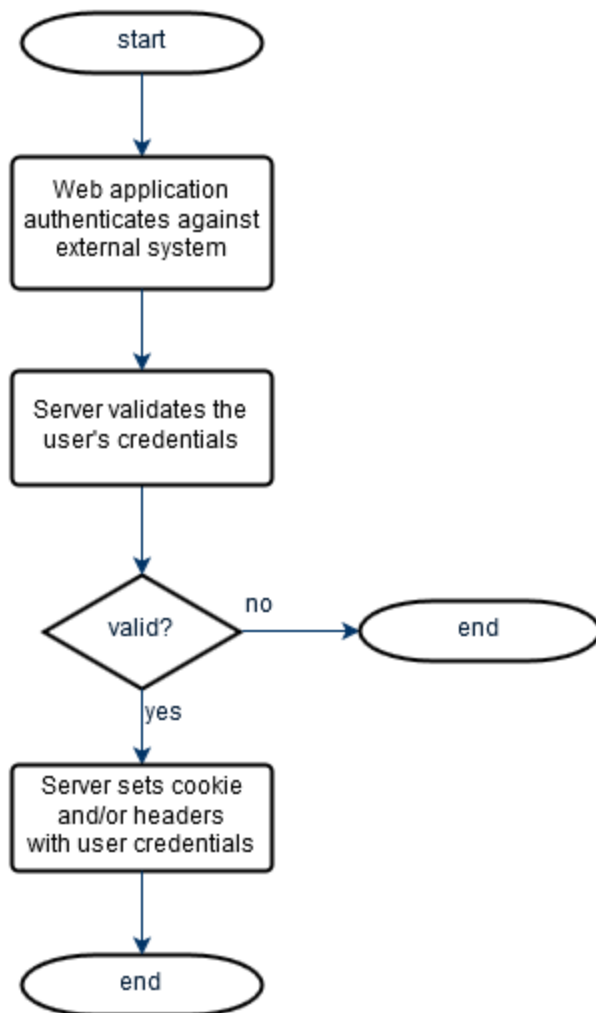
Building a Trust Validator

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.

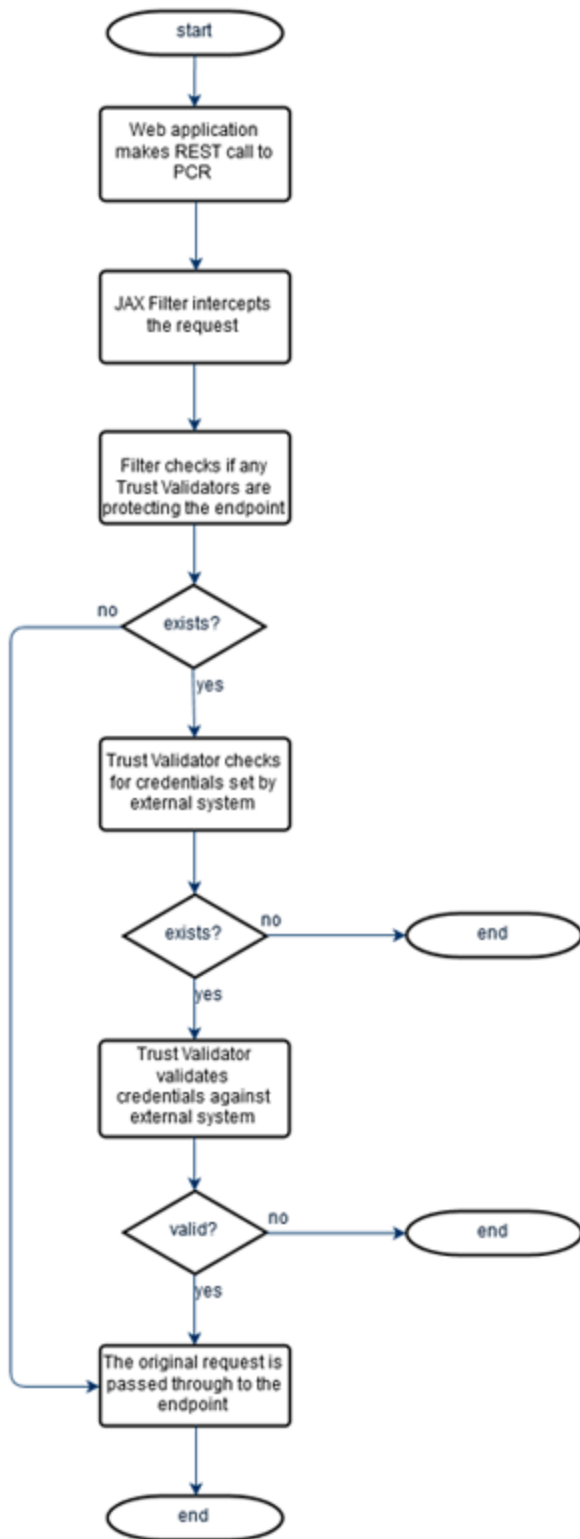
Trust Validation Diagrams

The following diagrams give a basic idea of how trust works within PCR. The first diagram does not directly involve Connect, but simply shows how a web application might interact with some external system prior to consuming a trusted endpoint inside the runtime.

Trust validation, step 1



Trust validation, step 2



Trust Validator Interfaces

The `RESTTrustValidator` interface contains 3 methods:

- `Object getToken(ContainerRequestContext context)` throws `TrustRequiredException` This method is responsible for pulling out/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`.
- `void postProcess(ContainerRequestContext context, Object token)` throws `Exception` This method is optional and does nothing by default. It is called on every request after a token is successfully validated (including instances where the token is retrieved from the cache and `validateToken()` is not called). It may be used to modify the context once the request has been designated as "trusted".

The `SOAPTrustValidator` interface contains 3 methods:

- `Object getToken(SOAPMessageContext context)` throws `TrustRequiredException` This method is responsible for pulling out/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`.
- `void postProcess(SOAPMessageContext context, Object token)` throws `Exception` This method is optional and does nothing by default. It is called on every request after a token is successfully validated (including instances where the token is retrieved from the cache and `validateToken()` is not called). It may be used to modify the context once the request has been designated as "trusted".

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

1. Depending on whether your validator will purpose SOAP, REST or both types of web services, your validator class should implement `SOAPTrustValidator`, `RESTTrustValidator` or both. Here is an example:

```
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> headers = (Map>) context.get(MessageContext.HTTP_REQUEST_
HEADERS);
        List 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.SampleTrustValidato
r"/>
    <service>
        <provide
interface="com.perceptivesoftware.pif.common.trust.RESTTrustValidato
r"/>
            <provide
interface="com.perceptivesoftware.pif.common.trust.SOAPTrustValidato
r"/>
        </service>
    </scr:component>

```

Using a Trust Validator

To use a validator on new SOAP and/or REST endpoints, choose one of the following options:

1. Preferred

End users can configure a validator at runtime by configuring a trust validator on the **Web Services** page. See the **Configuring Trust Validation** section of the [Perceptive Connect Runtime Installation and Setup Guide](#). This method does not require any additional effort by the developer and is the recommended choice for most connectors.

2. Discouraged

You can add a validation filter service dependency to your component. This should only be done if your connector explicitly depends on a specific trust validator because components with trust validators assigned via this method will not be modifiable by the end user. If you need a specific validator, complete the following steps:

- Build your web service through the use of one or both of the **pif-jaxrs-endpoint-archetype** and **pif-jaxws-endpoint-archetype**.
- 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 as `bind` and `unbind`, respectively. These services become available when the validator services component you just created is running.
- To depend on a specific validation service, add the following 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] (04_connector_development_tips.md).

Using Trust Validators with CORS and/or SSL

If an endpoint is configured to use trust, and CORS is not enabled in PCR, only *same-origin* requests will be able to be trusted, since *cross-origin* requests will not include authentication headers or cookies. If the cookie containing the user credentials required by the trust validator is flagged as HTTPS-only, PCR must also be using SSL.

If PCR is behind a reverse proxy, CORS may not be required in order for trust to function properly. A proxy does not, however, preclude the necessity of SSL if secure cookies will be used.

For further information see [CORS documentation](#).

Enable automatic connector upgrade handling

Perceptive Connect Runtime provides the `InstallService` interface which represents an "upgrade hook" that may be implemented to receive messages about bundle updates. The implemented service will receive 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, etc.

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 wish 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 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 will start 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** allow 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 checkbox, Eclipse preserves channel mappings and any settings that are changed in the admin console.

*** WARNING *** You should take care in changing the `Run Configuration`. It is very easy to include libraries that will not be available in the PCR Runtime Distribution, breaking your connector when it is ran in the stand-alone PCR engine.

Log and database file used in the default launch config


By default, your pif log and database files can be found in the `pcr-debug` directory under your Eclipse project's Workspace. To change the location of these files you can modify your launch file's `Arguments` (Eclipse tab in Debug Configurations) and change `-Dpcr.db.server.type="h2" -`
`Dpcr.db.databasesname=${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:
 - **Log Level**. The level of detail output when exceptions occur. (pif.log.level) For more information, refer to Logging states below.
 - **Log Directory**. The path location of the PIF log file. (pif.log.directory)
 - **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

Typically, you want to set minimal logging (Level 0) to only capture critical exceptions, unless you are debugging an issue. A verbose state (Level 4) can generate large log files affecting 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. This typically includes even more verbose details for debugging.

Deploy the connector to Perceptive Connect Runtime

Eclipse, by default, will continuously compile your project and allow 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 via Eclipse's m2e maven plugin, complete the following steps

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

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

1. Ensure that your `M2_HOME` and `JAVA_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`

To verify your build look for the `Reactor Summary:` section in either your eclipse console output or shell output. All projects should report SUCCESS.

* Your connector bundles will be available in the ``\target`` directories in your sub-projects.

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 in the **Java Options** input:
 - `-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=6006`
 - **NOTE** You may modify the *suspend* and *address* arguments to your needs. A *suspend* argument of **y** will cause 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 on **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** text field.
4. In the **Project** box, click on **Browse** and then select the connector project to debug.
5. In **Connection Properties** field, fill in the **Host** and **Port** fields.
6. 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.

7. 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.

Core Concepts

The Connect Scheduler design relies on a few key concepts: the **Scheduler**, **Jobs**, **JobTriggers**, and **Schedules**.

Scheduler

The Connect Scheduler provides a single `Scheduler` service that allows PCR users to create jobs that trigger on a set schedule. The `Scheduler` registers a `Job` using an identifying `JobDetail` and executes a `Job` using a registered `JobTrigger` that matches the identifying `JobDetail`. The `Scheduler` may contain 0..n `JobTriggers` associated with a single `Job`. However, a `Job` never executes without an associated `JobTrigger`, and a `JobTrigger` only has 0..1 associated `Jobs`. Each `JobDetail` and `JobTrigger` must be unique within the `Scheduler`. The `Scheduler` handles conflict resolution between `JobDetails` and `JobTriggers` based on the source of these Objects.

Scheduler Source Based Conflict Resolution

The `Scheduler` provides both API calls and OSGi bindings to register `Jobs` and `JobTriggers`. The bindings take precedent over the API calls. When an added object conflicts with a newly bound object, the `Scheduler` replaces the added object with the bound one, and when a bound object conflicts with a newly added object, the `Scheduler` rejects the added object with an exception. When an added object conflicts with another stored, added object, then the new object replaces the original object.

The scheduler only replaces bound conflicting objects when the objects have the same Service PID. If a newly bound object conflicts with another stored, bound object that has a different PID, then the `Scheduler` logs an error and stores the new binding without registering it and without affecting the original bound object. The stored bound object remains in this state until it is unbound, rebound, or updated. If the object is rebound or updated, then the scheduler checks for conflicts and acts accordingly. Updating or rebinding an existing bound object with a conflicting object removes the original, valid object.

Job

A `Job` is a unit of work executed by the `Scheduler` based on the `Schedule` of a `JobTrigger` associated with the `Job's JobDetail`. All `Jobs` must implement the `Job` interface. This interface contains a single method `execute`, which consumes a `JobExecutionContext` that contains information about the previous, current, and future executions of the `Job`. The `Scheduler` registers `Jobs` either through its API with methods that require the `Job's` associated `JobDetail` or as a `DetailedJob` through its OSGi bindings.

JobDetail

The `JobDetail` interface provides the identifying information for a `Job`, the `Job`'s durability, and whether or not the `Scheduler` can execute the `Job` concurrently. The `Scheduler` requires a `JobDetail` to add a `Job` through the API. When the `Scheduler` unregisters a `JobTrigger` associated with a `Job`, if the `Job` is not durable and does not have any other associated `JobTriggers`, then the `Scheduler` also unregisters the `Job`. To properly enforce OSGi binding, bound `Jobs` must be durable.

DetailedJob

The `DetailedJob` interface represents a `Job` that is also its own `JobDetail`. The `Scheduler` automatically binds any OSGi `Services` that implement `DetailedJob`. If necessary, you can programmatically add `DetailedJobs` through the API by casting the `DetailedJob` to `Job` and `JobDetail` for the respective arguments. However, the `Scheduler` replaces any added `Job` with a matching bound job, so providing your `DetailedJob` as a service helps protect it against potential key conflicts.

JobTrigger

The `JobTrigger` interface defines a schedule, along with other properties, that describes how and when the `Scheduler` executes a `Job`. Multiple `JobTriggers` can reference the same job, but each `JobTrigger` only has one associated `Job`. Once the `Scheduler` contains a `Job` and a `JobTrigger` associated with the `Job`'s `JobDetail`, then the `Scheduler` begins executing the `Job` based on the `Schedule` provided by the `JobTrigger`.

Each `JobTrigger` contains an activation date, a deactivation date, the key of the `Job` associated with the `JobTrigger`, a priority, and a `Schedule`. Regardless of the `JobTrigger`'s `Schedule`, the `Scheduler` will not execute the associated `Job` before the activation date or after the deactivation date. However, the `Scheduler` cannot stop a `Job` that continues an execution past its deactivation date. If a `JobTrigger` does not have an activation date, then the `Scheduler` assigns it the current date once the `JobTrigger`'s associated `Job` is also in the `Scheduler`. If a `JobTrigger` does not have a deactivation date, then the `Scheduler` executes the associated `Job` indefinitely according to the `JobTrigger`'s `Schedule`.

A `JobTrigger`'s priority determines execution order when the `Scheduler` triggers multiple `Jobs` at once. `JobTriggers` with higher priorities execute before those with lower priorities. `JobTriggers` with the same priority execute in an indeterminate order.

Schedule

A `Schedule` defines when a `JobTrigger` fires within its activation and deactivation date. There are two provided `Schedule` types: `CronSchedule` and `SimpleSchedule`.

CronSchedule

A `CronSchedule` uses a [Cron Expression](#) to define when the `JobTrigger` fires. For example String Cron Expressions and format information, see the Oracle documentation for [A Cron Expression](#). The `CronSchedule` follows the Cron Expression using its Time Zone. The PCR Scheduler provides a `JobTrigger` with a `CronSchedule` through the PCR Config Admin. This `CronScheduleJobTrigger` is the primary source for bound `JobTriggers`.

SimpleSchedule

A `SimpleSchedule` defines an optionally repeating Schedule that fires on a set interval. Provided the Schedule exists between its `JobTrigger`'s Activation Date and Deactivation Date, a `SimpleSchedule` always fires at least once.

All `SimpleSchedules` have a repeat count and a repeat interval. The repeat count may be -1, 0, a positive integer. A -1 repeat count indicates that the `SimpleSchedule` repeats indefinitely, and a 0 repeat count indicates that the `SimpleSchedule` executes once. For definite repeat counts, the maximum number of Scheduled executions is the `repeat-count+1`. If `activation-date + (repeat-count * (job-execution-time + repeat-interval)) > deactivation-date`, then `SimpleSchedule` may execute less than this max.

The repeat interval is a long representing the number of milliseconds between Schedule executions. This interval must be 0 or a positive long. If the job allows concurrent execution, then a Schedule with a 0 repeat interval fires all of its executions concurrently. If the job does not allow concurrent execution, then a Scheduler with a 0 repeat interval fires each execution as soon as the previous execution finishes.

The PCR Scheduler only provides `SimpleSchedules` through its API. The `SimpleSchedule` API has some unintuitive behavior that should not be exposed non-programmatically. The behavior occurs because the `SimpleSchedule` starts calculating fire times from the moment the `Scheduler` registers it. Registering a `SimpleScheduler` with an associated `Job` that appears after the `SimpleSchedule`'s `JobTrigger` Activation date exposes this problem. Once the `Scheduler` registers the `SimpleSchedule` `JobTrigger` with its `Job`, the `SimpleSchedule` calculates the number of missed fire times between its Activation Date and the time the `Job` appeared and fires accordingly. So if your `SimpleSchedule` activates now, fires once every minute and it takes 10 minutes for it to register with its `Job`, then the `SimpleSchedule` immediately fires 10 times.

Using the Connect Scheduler

The Connect Scheduler provides two access routes: through the API, and through OSGi Services. The Connect Scheduler exposes the API through its `Scheduler` service and some builder classes. The `Scheduler` service dynamically binds all `JobTrigger` and `DetailedJob` services present in its OSGi environment.

Using the API

The Connect Scheduler API allows you to programatically register your own `Jobs`, `JobDetails`, and `JobTriggers` through your own OSGi classes. The API also provides `CronSchedule`, `SimpleSchedule`, `JobDetail`, and `JobTrigger` builder classes to assist you in creating your own objects.

Scheduler Service

To use the API, your OSGi service must bind `com.perceptivesoftware.pif.scheduler.Scheduler`. Once bound, the `Scheduler` service provides methods to add and remove `Jobs` and `JobTriggers`. Bound Connect Scheduler objects take precedent over added Connect Scheduler objects, so be aware that the `Scheduler` may replace your added `Jobs` and `JobTriggers` with bound ones if the bound objects have the same key as an added object. See the [Scheduler Javadoc](#) for more detailed API behavior.

Builder classes

The API provides classes that help build Connect Scheduler objects. These builders cover each of the provided implementations for their related objects. If necessary, you may provide your own implementations for the `JobDetail`, `JobTrigger`, `CronSchedule` and `SimpleSchedule` interfaces. See the related Javadocs for more information about each interface's expected behavior.

Using OSGi Services

The Connect Scheduler dynamically binds any service that provides the `DetailedJob` or `JobTrigger` interface. The bindings replace any added objects that conflict with the bound object. The Connect Scheduler provides two classes to simplify bindings: a `CronScheduleJobTrigger` that creates configurable `JobTrigger` services, and an `AbstractDetailedJob` that provides the base implementation for `DetailedJob` services.

CronScheduleJobTrigger

The `CronScheduleJobTrigger` is a `ManagedService` configured through Config Admin that provides a `JobTrigger` service with a `CronSchedule`. The `CronScheduleJobTrigger` provides every field required to register a `JobTrigger` with a few default values that cover most use cases. Without explicitly setting the related configuration properties, the `JobTrigger` activates once its saved, never deactivates, and uses the systems default Time Zone for the `CronSchedule`. The configuration in OSGi displays its Trigger Key as its label.

AbstractDetailedJob

The Connect Scheduler provides `AbstractDetailedJob` as a starting point for your `DetailedJob` service. An `AbstractDetailedJob` is an `AbstractLifecycleComponent` that implements `DetailedJob`. This component expects the `DetailedJob`'s `JobKey` and description to come from either its registration or configuration properties as defined by `getSchedulerDescriptionProperty()`, `getSchedulerJobGroupProperty()`, and `getSchedulerJobNameProperty()`. If these values need to be hardcoded, you must override `getKey()` and `getDescription()` and simply return null from the abstract property methods. However, we suggest allowing users to configure the `JobKey`. Each `JobKey` must be unique within the `Scheduler`. If your hardcoded `JobKey` conflict with another bound `DetailedJob`, you will need to recompile your project with a unique key to guarantee your `DetailedJob` registers properly.

`AbstractDetailedJob` are all durable Jobs. You cannot override `isDurable()`. By default, `AbstractDetailedJob` allow concurrent execution. If your `AbstractDetailedJob` implementation cannot execute concurrently or need configurable concurrency, you must override `isConcurrentExecutionDisallowed()`.

Connector Development Tips

This section will contain various tips, tricks, and best practices that have been established connector developers have encountered during development.

Connector Names and Bundle names

Connector Name

PCR 1.5 introduces a new, optional field for bundle manifests called `PCR-Connector-Name`. It is recommended that connector developers add this property to the `manifest.mf` for each of the bundles (not including third-party dependencies which are packaged with the connector) in their connector. In PCR 1.5, this field is only used to identify the source of JAX applications on the Web Services page. However, future releases will build upon the larger concept of a concrete "connector" within the runtime.

Though encouraged, the `PCR-Connector-Name` header is not required in PCR 1.5. Therefore, connector developers are not required to release a new version simply to add this field. Features in PCR core which expect the `PCR-Connector-Name` header will use a new set of utilities to make an educated guess based on the available information at execution time.

Bundle Name

Manifests also contain a `Bundle-Name` field, part of the OSGi specification. The `Bundle-Name` should be a name which encompasses all of the components/services and code inside the bundle.

Example

```
project.logging/META-INF/MANIFEST.MF
```

```
PCR-Connector-Name: My Connector  
Bundle-Name: Logging Utilities
```

```
project.common/META-INF/MANIFEST.MF
```

```
PCR-Connector-Name: My Connector  
Bundle-Name: Core Utilities and Interfaces
```

```
project.rest/META-INF/MANIFEST.MF
```

```
PCR-Connector-Name: My Connector  
Bundle-Name: REST Endpoints for Application
```

Readers and Writers

- `ParameterSource` and `ParameterTarget` types must be visible to JAXB to enable XML serialization and de-serialization. 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 this new functionality:

For existing REST Services:

1. Your `RESTComponent` should extend `AbstractRESTComponent` rather than `AbstractLifecycleComponent`. If your REST Service already extends `AbstractRESTComponent`, skip to step 2. If not, complete the following steps:
2. 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.
3. In the same class, modify the startup method by changing `service.registerApplication` to `registerApplication`
4. In the same class, remove the methods `registerServlet` and `unregisterServlet`. Additionally, you can remove the `shutdown` method.
5. In the component definition to your `RESTComponent`, update the service reference to the `JAXRSService` to call `bind` and `unbind` rather than `registerServlet` and `unregisterServlet`
6. In the component definition to your `RESTComponent`, you must add the reference to the REST filter service `RESTValidationFilter`.
7. 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 an 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.

For 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 that your component can reference the validation filter if it exists:

```
``` public void bind(SOAPValidationFilter newValidationFilter) { validationFilter = newValidationFilter; }  

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

2. Modify the call to `registerEndpoint` in the startup by adding the `validationFilter` as an additional argument like so:

```
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

### 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 Eclipse documentation.

### POM files

Refer to the "[POM](#)" topic in Maven documentation.

### 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`. 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 that the component is bound to.

## 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 is included below.

### 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, these are configured after selecting an archetype. In general, the **Group Id** and **Version** should be consistent for all projects in a connector, and 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

#### 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 it's 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 it's 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 via 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 it's 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 (mostly) 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 it's 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.

# What is CORS

## Terms

- **CORS**: Cross-Origin Resource Sharing
- **origin**: The protocol, fully qualified domain, and port (such as `protocol://fulldomain:port/`).
- **protocol**: for the purpose of this documentation, *protocol* refers to either `http` or `https` in a web addresss.
- **domain**: The fully qualified name for a web address includes all sub-domains as well as the domain root and TLD extension (such as `sub1.sub2.sub3.domain.com`); alternatively, a domain may be a machine name (such as `APPSERVER-1`)
- **cross-origin**: An `XMLHttpRequest` made from one origin to another is considered *cross-origin* if the source and destination have different *origins*.
- **same-origin**: An `XMLHttpRequest` made from one origin to another is considered *same-origin* if the source and destination have identical *origins*.
- **secure cookie**: A cookie which is only sent by the browser when the destination of the request (regardless of whether it is *cross-origin* or *same-origin*) is using SSL.
- **HTTP-only cookie**: A cookie which is not readable from client-side code; only the browser and server are able to read HTTP-only cookies.
- **CSRF**: Cross-site request forgery, an attack vector similar to XSS but not requiring JavaScript to exploit. Sometimes protected against by the use of **CSRFTokens**.

## Detailed Explanation

CORS allows JavaScript code that has been served from origin A to access resources served by origin B inside the client's browser. If a web application served at `app1.domain.com` (origin `http://app1.domain.com:80/`) initiates a request to `app2.domain.com` (origin `http://app2.domain.com:80/`), that request is considered **cross-origin**. The web server for `app2.domain.com` (e.g. Apache, Tomcat, Jetty, Nginx, PCR, etc) must be configured to allow cross-origin requests.

However, if a web application running on a web server makes a request to another application running on the same web server, that request is not considered to be cross-origin. For example, a web application running at `apps.domain.com/app1/` making a request to an application running at `apps.domain.com/app2/` is not cross-origin, because the origin for both applications is `http://apps.domain.com:80/`.

## CORS Requests with Secure Data

**Note** Secure data in the examples below refers only to data that is domain-specific, such as cookies or authentication headers. It does not explicitly mean that the data is served over `https`.

A **cross-origin** request with credentials occurs when the request from `app1.domain.com` to `app2.domain.com` includes cookies and other secure data stored in the browser related to `app2.domain.com`. In other words, the browser must have previously visited `app2.domain.com` or some other application(s) which set data (such as cookies and authentication headers) for `*.domain.com` in order for secure data to be included in a request from `app1.domain.com` to `app2.domain.com`.

If the `app2.domain.com` application sets any cookies (such as a `lastAccess` cookie), they will be ignored by the browser during a CORS request unless the `withCredentials` flag is set to `true`. In that case, the cookie will be set normally and will be included in subsequent CORS requests to `app2.domain.com` which use the `withCredentials` flag.

To make a CORS request with credentials, you must set the `withCredentials` flag on the `XMLHttpRequest`.

**Note:** Cookies for a given host are shared across all the ports on that host. This differs from the usual "same-origin policy" used by web browsers that isolate content retrieved via different ports.

## What CORS Cannot Do

A **cross-origin** request from `app1.domain.com` to `app2.domain.com` will not include secure data related to `app1.domain.com`, regardless of whether the `withCredentials` flag is set. However, secure data which is tied to `.domain.com` or `*.domain.com` will be sent with the requests when `withCredentials` flag is set.

If you wish to send secure data from `app1.domain.com` to `app2.domain.com`, you will either need to use a reverse proxy, or you will need to attach the desired data directly to the CORS request in the form of custom headers. In the latter case, you will also need to ensure that the server running `app2.domain.com` is configured to allow these headers in cross-origin requests. An example of this scenario is using Trust in PCR. For more information about Trust, refer to the sections related to Trust Validators and Validation Filters in ["Build a Connector"](#) and ["Connector Development Tips"](#). **Note** Specific instructions for configuring either allowed CORS headers or a reverse proxy are outside the scope of this document.

## CORS, PCR, and Your Application(s)

There are a wide variety of ways in which PCR, Integration Server, and other applications may be deployed. Each variation will require slightly different configuration, based on the your needs. The following examples show how PCR might interact with a Perceptive Experience application which uses Integration Server session tokens for user authentication.

## Passing secure data from Perceptive Experience to PCR

Although the explanations below demonstrate how an Experience application can pass its credentials to PCR, it is very important to reiterate that CORS does not allow one origin to send its own secure data (such as cookies or headers) to an application with a different origin. However, in some of the examples below, the desired credentials are stored in a browser cookie with the server's domain (such as `localhost`). Because of this, the cookie from Experience would normally be sent by the browser if the user interacted directly with PCR, as would cookies sent by any application which runs on `localhost`. Because these credentials are applicable to PCR's origin from the browser's perspective, they are included in CORS requests made using the `withCredentials` flag.

PCR and Experience must use a reverse proxy if they are on different domains. This is because cookies are tied to a specific domain, and a CORS request from Experience to PCR with credentials will only include cookies tied to the domain on which PCR is running.

The following list provides a description of the colored arrows used to connect services in the diagrams below.

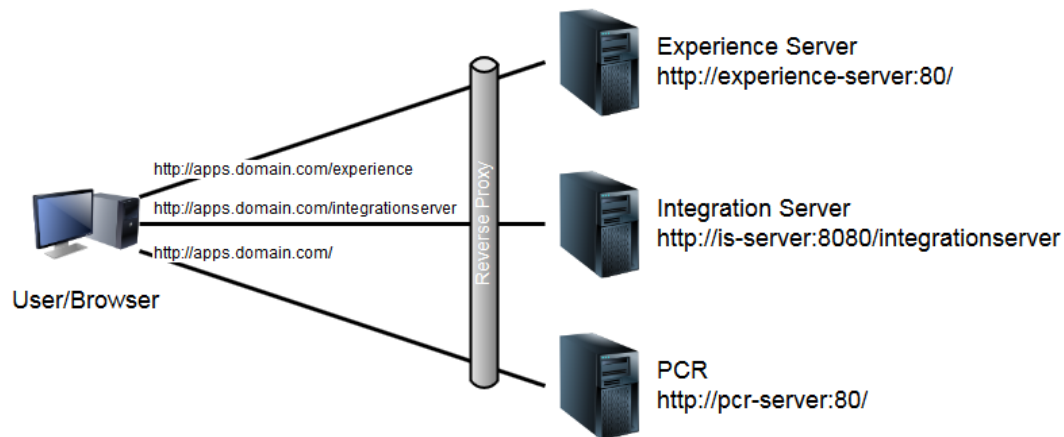
- **Black** A black arrow indicates that the communication between the systems is a plain **same-origin** request.
- **Orange** An orange arrow represents a request which is **cross-origin** but does not use the `withCredentials` flag and therefore does not include cookies or other secure data.
- **Green** A green arrow represents a request which is **cross-origin** and which also sets the `withCredentials` flag to `true`. These requests may include domain cookies or other secure data as described under each diagram.

These examples show a few different ways that PCR can interact with Perceptive Experience Applications, but any web application could be used in place of Experience. This is not meant to be a complete list of all possible scenarios, but rather a representative set of examples.

Although the examples refer to a reverse proxy, in practice it could be a load balancer, firewall, or other intermediate system. These systems have a wide range of possible configurations, some including the termination of SSL at the barrier and using plaintext for internal communication. Generally speaking, these variations in network topology are transparent to the browser and do not have an effect on the examples below.

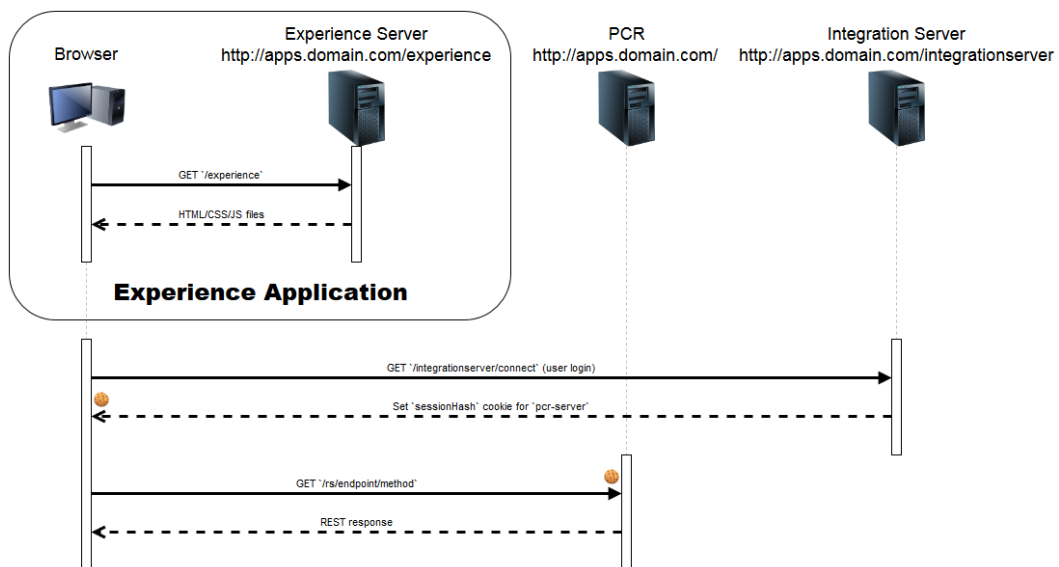
## Example 0

### System layout



### CORS system connections - example 0

### Sequence diagram



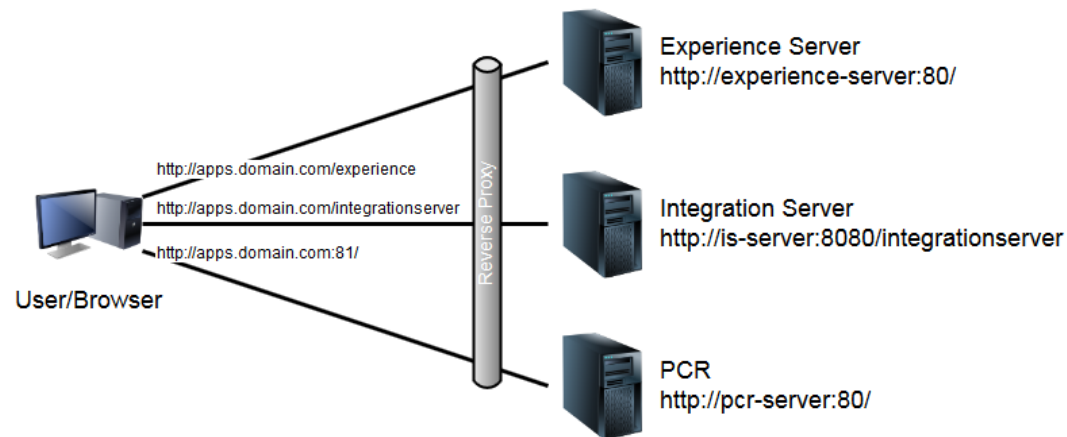
### CORS sequence diagram - example 0

In this example, all of the systems are behind a reverse proxy and aliased to the same domain and port. As a result, AJAX requests between them are considered *same-origin*, and CORS does not come into play.



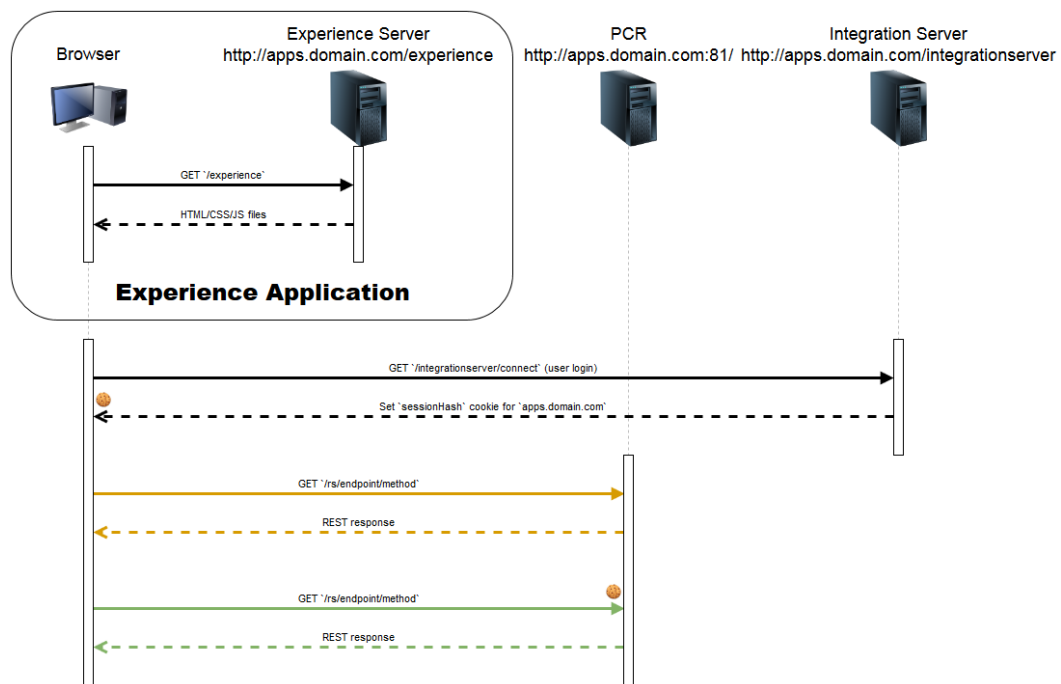
## Example 1

### System layout



### CORS system connections - example 1

### Sequence diagram

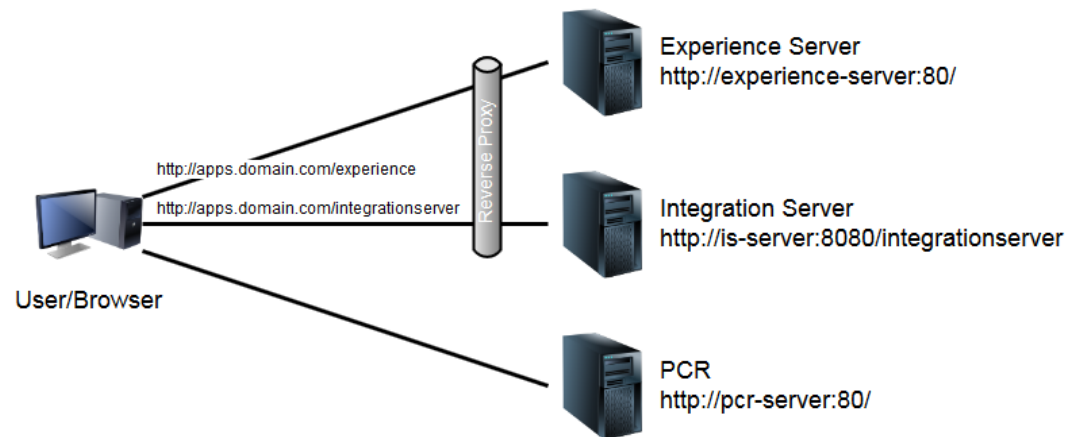


### CORS sequence diagram - example 1

In this example, all of the browser's requests to the different systems go through a reverse proxy. In the *cross-origin* request using the `withCredentials` flag, the `sessionHash` cookie from Integration Server will be included by the browser. In the request which does not set the flag, the cookie is not sent to PCR.

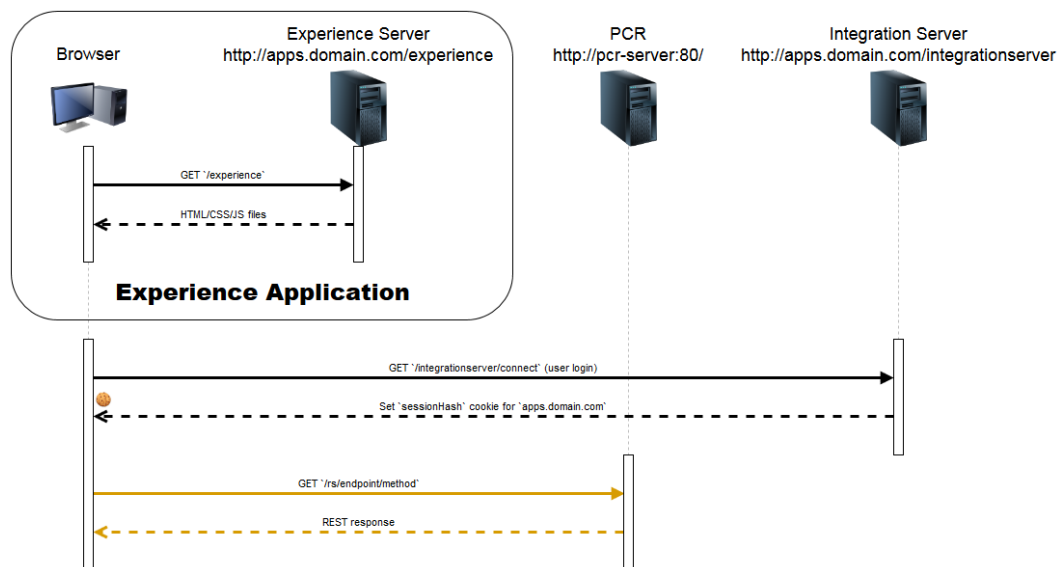
## Example 2

### System layout



### CORS system connections - example 2

### Sequence diagram

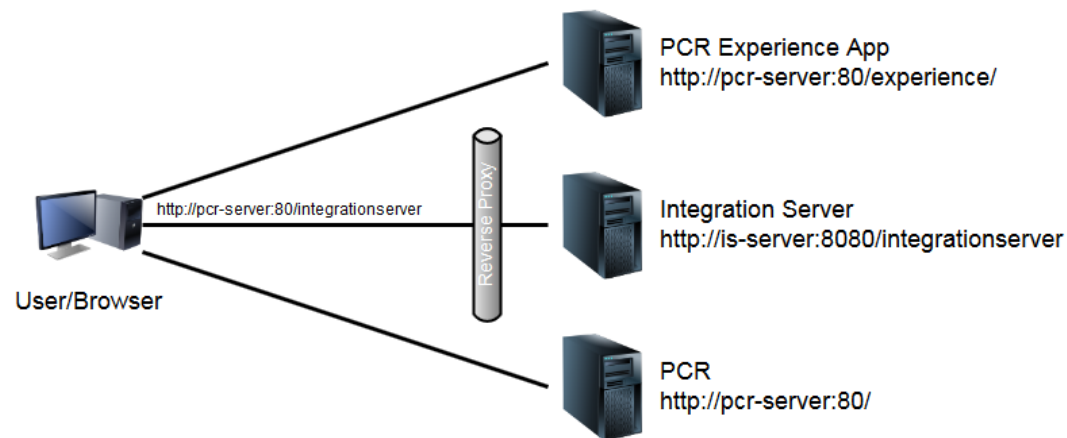


### CORS sequence diagram - example 2

In this example, the browser's requests to Experience and Integration Server go through a reverse proxy, while the requests to PCR do not. As a result, all interaction between the Experience application and PCR is *cross-origin*, but the `sessionHash` cookie set by Integration Server will never be sent by the browser, regardless of the `withCredentials` flag. This is because PCR is on a different domain from Experience/IS, not just another origin on the same domain. Since the cookie is tied to `domain.com`, it cannot be sent to `pcr-server`.

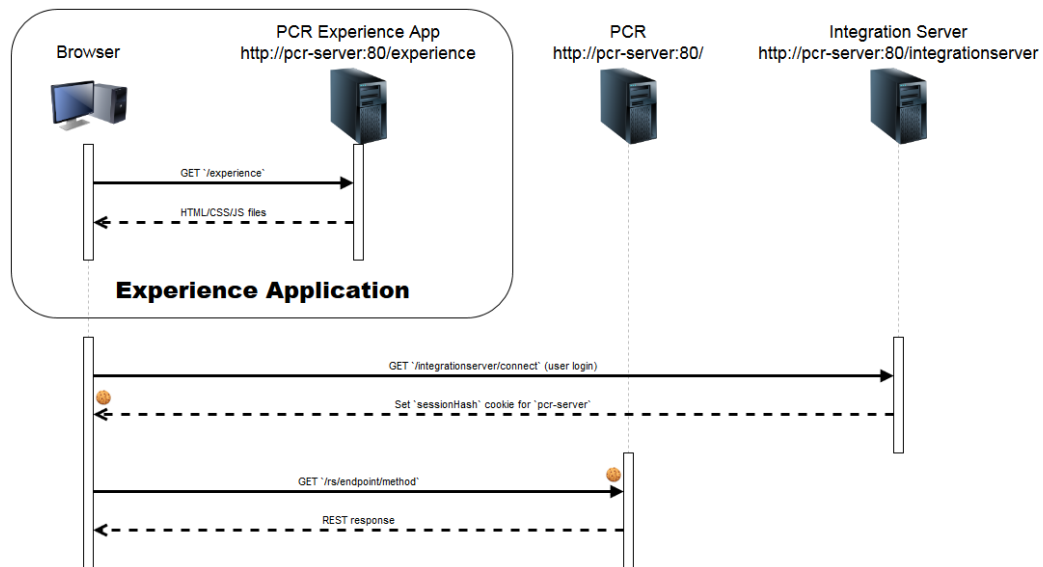
## Example 3

### System layout



### CORS system connections - example 3

## Sequence diagram



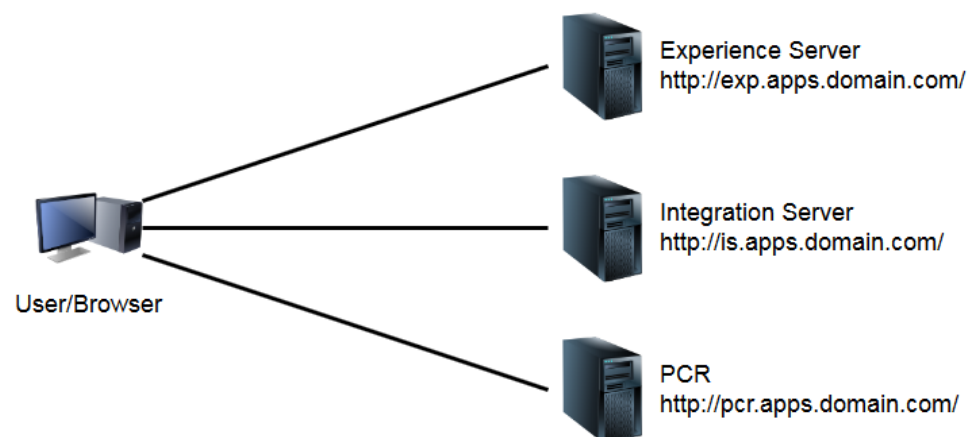
## CORS sequence diagram - example 3

In this example, the Experience application is running within PCR itself. As a result, all interactions between Experience and PCR are *same-origin*, and no special configuration is required to pass the `sessionHash` cookie set by Integration Server.

*Note:* The proxy for Integration Server in this example could be a software proxy within PCR or another proxy server.

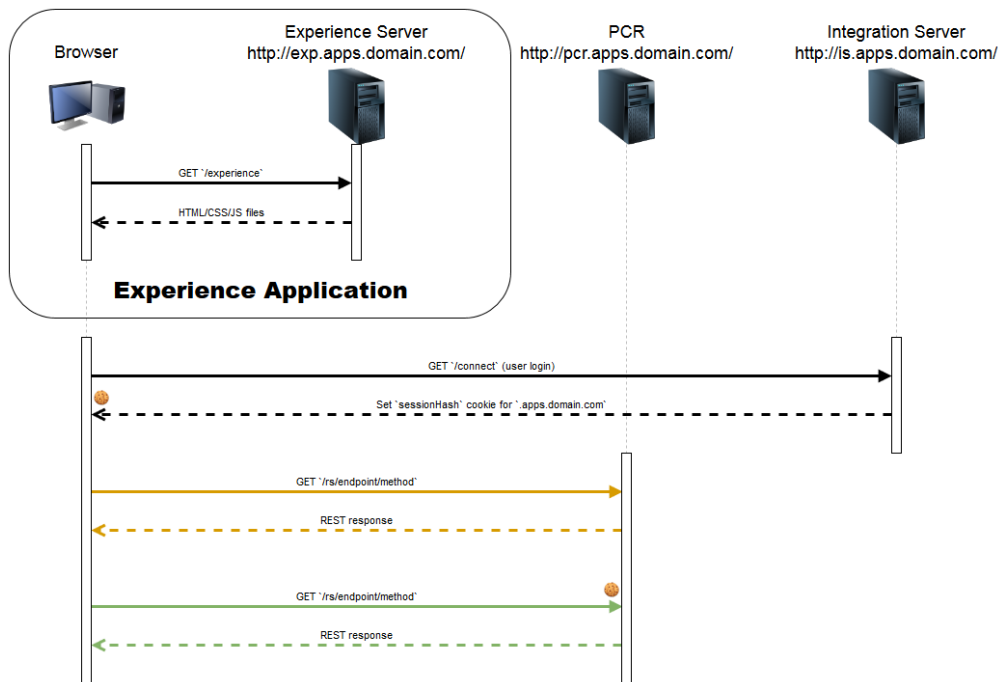
## Example 4 (Hypothetical)

### System layout



## CORS system connections - example 4

## Sequence diagram



### CORS sequence diagram - example 4

This is a hypothetical variation of Example 1 above. In this example, each application lives on a subdomain of `apps.domain.com`. Integration Server would be configured to set the `sessionHash` cookie for `.apps.domain.com` instead of the more specific default of `is.apps.domain.com`\*. As a result, in the *cross-origin* request using the `withCredentials` flag, the `sessionHash` cookie from Integration Server will be included by the browser. In the request which does not set the flag, the cookie is not sent to PCR.

\*Note: Integration Server does not currently support this functionality, but we have suggested it to the Content teams.

## CORS with Secure Cookies

If you anticipate that some cookies included with CORS requests to PCR might be flagged as secure, you must also ensure that PCR is running under SSL. If it is not, the secure cookies will never be received by PCR, regardless of whether or not the incoming CORS request sets the `withCredentials` flag.

See [Appendix E: Configuring SSL](#) in the Connect Install Guide for more information about configuring PCR to use SSL.

## Making CORS Requests from Your Web Application

If you are developing a web application (such as an Experience module), and your application will make REST calls to PCR, follow the instructions in [Appendix B of the Connect Install Guide](#) to configure PCR to allow CORS requests. Once PCR is properly configured, you can make CORS requests from your web application.

### Using Pure Javascript

If you are not using jQuery (or a front-end framework which uses jQuery for AJAX requests), this is a very basic example for how you might make a CORS request to PCR from your application.

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://pcr-server:80/some/endpoint', true);
xhr.send(null);
```

If you want to use cookies or secure data set by PCR (such as the user's login session), you need to set the `withCredentials` flag on your `xhr` object.

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://pcr-server:80/some/endpoint', true);
xhr.withCredentials = true;
xhr.send(null);
```

### Using jQuery

The jQuery equivalents for the two previous examples are as follows:

```
$.ajax({'url': 'http://pcr-server:80/some/endpoint'});

$.ajax({
 'url': 'http://pcr-server:80/some/endpoint',
 'xhrFields': {
 'withCredentials': true
 }
});
```

## Passing Secure Data from Your Web Application to PCR

If you want to make AJAX requests from a web application to PCR which include cookies and/or other secure data for your web application's domain, either both applications must be running on the same domain, or you must use a reverse proxy. As mentioned above, CORS requests do not allow one origin to send its own secure data to another origin. In such a case, requests made from your application to PCR are likely to be considered **same-origin**.

## Handling CORS Requests Made to Your Connector's Endpoint

### Using PCR Global CORS Settings

If you are developing a connector for PCR which will include a REST endpoint that doesn't require any advanced CORS directives but will be used from *cross-origin*, there are no extra steps you need to take at development time. However, the PCR administrator will need to follow the instructions in the Connect Install Guide ([LINK](#)) to configure PCR to allow CORS requests. Once PCR is properly configured, your endpoint will be able to receive CORS requests from external web applications.

### Using JAX-RS Annotations

If your connector might be deployed in a runtime which has CORS disabled, or if you need to override some settings for a specific endpoint or method, you may use the JAX-RS annotation

`CrossOriginResourceSharing` to set specific CORS settings.

The following settings in Connect runtime can be overridden by the `CrossOriginResourceSharing` annotation.

- **`connect.rs.cors.allow.credentials`**: `allowCredentials`
- **`connect.rs.cors.allow.all.origins`**: `allowAllOrigins`
- **`connect.rs.cors.allowed.origins`**: `allowOrigins`

An example of the `CrossOriginResourceSharing` annotation is below. The annotation can be applied to an entire class, to individual methods, or both. The most specific annotation or filter is the one which will be applied to CORS requests for a given path.

```
@CrossOriginResourceSharing(
 allowOrigins = {
 "http://localhost:8080"
 },
 allowCredentials = true,
 allowHeaders = {
 "x-custom-1", "x-custom-2"
 },
 exposeHeaders = {
 "x-custom-3", "x-custom-4"
 },
 maxAge = 1,
 allowAllOrigins = false,
)
```

Choose the properties that are right for your endpoint/method.

See documentation for [org.apache.cxf.rs.security.cors](#) for more details on the available properties.

## Explicitly Setting CORS Headers

The most manual option for handling CORS requests for your endpoint is to explicitly define `@OPTIONS` methods for any endpoints which will be accessible from other origins. You will also need to manually set the appropriate CORS headers in the response sent by both the `OPTIONS` method as well as in the actual (e.g. `GET`, `POST`, etc) method.

For example, this method sets headers for the preflight request.

```
@OPTIONS
@LocalPreflight
@Path("/")
public Response options() {
 return Response.ok()
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_METHODS, "GET
DELETE PUT")
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_CREDENTIALS,
"true")
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_HEADERS, "x-
custom-1, x-custom-2")
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_ORIGIN,
"http://localhost:8080")
 .build();
}
```

Use `@LocalPreflight` when you want to overwrite the `OPTIONS` preflight when Connect runtime is configured for CORS.

Once a browser passes the preflight check, it makes the actual request, which is handled by this method.

```
@GET
@Path("/")
public Response DoWork(@QueryParam("param1") String param1) throws
Exception {
 String message = "Welcome to DoWork with " + param1;
 // do something with param1
 Gson gson = new Gson();
 return Response.status(200)
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_METHODS, "GET
DELETE PUT")
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_CREDENTIALS,
"true")
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_HEADERS, "x-
custom-1, x-custom-2")
 .header(CorsHeaderConstants.HEADER_AC_ALLOW_ORIGIN,
"http://localhost:8080")
 .entity(gson.toJson(message))
 .build();
}
```



## Configuring or Overriding CORS in Either Situation

The recommended approach to using CORS in PCR is to enable it globally and, if necessary, annotate endpoints/methods which require specific CORS configuration different from the anticipated global settings. Using both annotations and manually handling the OPTIONS request is technically possible, but is prone to subtle errors if not done correctly.

## Setting Browser Cookies from Your Endpoint

If your REST endpoint sets one or more browser cookies as part of handling web requests, cross-origin requests made to the endpoint must set the `withCredentials` flag to `true` in order for the cookies to be persisted. Same-origin requests do not require any special configuration to set browser cookies.

## Appendix: CORS Configuration for Web Servers

Typical deployments of Connect Runtime which include Experience and Integration Server will likely have Experience and Integration Server running under an Apache Tomcat server. While the sections above describe how to configure PCR for incoming CORS requests, it is also important to note that Integration Server itself can potentially be on the receiving end of a *cross-origin* request. This requires Tomcat to have a CORS filter configured either globally or for the IS application specifically. One key benefit of this approach is that it removes the requirement which mandates that the Experience application must run from the same Tomcat server as Integration Server. This does not necessarily eliminate the need for a reverse proxy in all scenarios. It simply allows for more flexibility in deployments.

## Configuring CORS in Tomcat

As with many Tomcat configurations, CORS can be configured globally for all Catalina applications on the server or just one. Here is a simple example of a CORS filter configuration:

```
<filter>
 <filter-name>CorsFilter</filter-name>
 <filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
 <init-param>
 <param-name>cors.allowed.origins</param-name>
 <param-value>http://domain1.com</param-value>
 </init-param>
 <init-param>
 <param-name>cors.allowed.methods</param-name>
 <param-value>GET, POST, HEAD, OPTIONS, PUT</param-value>
 </init-param>
 <init-param>
 <param-name>cors.allowed.headers</param-name>
 <param-value>Content-Type, X-Requested-With, accept, Origin, Access-
Control-Request-Method, Access-Control-Request-Headers</param-value>
 </init-param>
</init-param>
```

```

 <param-name>cors.exposed.headers</param-name>
 <param-value>Access-Control-Allow-Origin,Access-Control-Allow-
Credentials</param-value>
 </init-param>
 <init-param>
 <param-name>cors.support.credentials</param-name>
 <param-value>true</param-value>
 </init-param>
 <init-param>
 <param-name>cors.preflight.maxage</param-name>
 <param-value>10</param-value>
 </init-param>
</filter>
<filter-mapping>
 <filter-name>CorsFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>

```

Example adapted from [Tomcat's documentation](#)

## Other Web Servers

Most web servers provide a way to configure CORS settings for individual sites, globally, or some combination thereof. For specific instructions for your web server, refer to the server's documentation. An additional resource is [enable-cors.org](#), which provides configuration examples for a wide range of servers and clients.

## 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.

Q. 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.cxf" bundle was not being imported into the project. The OSGI dependency resolver sometimes returns unexpected error messages, and this was one of those instances.

## Q. 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-dependant classes that are provided by the Java runtime (ie. `sun.*` or `javax.*` packages). By default, these packages are blocked by the Felix classloader. The reasoning for this is that these packages are platform specific, and thus, not guaranteed to be available on every JVM. As a developer, this is an annoyance as the error is only exposed at runtime via `NoClassDefFoundError/ClassNotFoundException`.

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.
2. In Eclipse, open the Manifest file, and click on the 'Dependencies' tab.
3. Click 'Add' under 'Imported Packages' and add the required package. In the example above, this would be `"javax.imageio.metadata"`
4. Select the newly added package, and click 'Properties'.
5. Check the 'Optional' checkbox, and click 'OK'
6. Save the Manifest file, and you should now see something like  
`javax.imageio.metadata;resolution:=optional`, if you view the file in a text editor.
7. 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 will need to perform this step after connector installation.
1. Open the `PCR Install Directory/conf/config.properties` for the 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.