### Subversion

### Java

The developer's machine and server must have Java 1.6+ installed.

### J2EE Servlet Container

The CPF is deployed as two web applications (WAR) to a J2EE application server or servlet container. The following servlet container is currently supported.

* Tomcat 7.x

For Tomcat 7.x you will need to add a user account in the manager-script role to deploy the web applications to the tomcat contained. Edit the tomcat-users.xml file in the tomcat conf directory.

  <role rolename="**manager-script**"/>

  <user username="**admin**" password="\*\*\*\*\*\*\*\*" roles="**manager-script**"/>

### Spring Framework

The CPF and plug-ins depend heavily on the spring framework[[1]](#footnote-1) and the XML configuration mechanism. Developers should have a least a basic understanding of defining spring beans and dependency injection.

http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/

# Plug-in Development

## Plug-in Overview

A CPF Business Application plug-in is a service that takes one or more request parameters, performs an action using the request parameters and generates one or more result attributes. The CPF allows a user to create a job that contains one or more requests to be executed against a single business application. For example a job may contain 1 million addresses to geo-code.

For an example a power service would have the request base and exponent and the result attribute result, which is the calculation of base to the power of exponent.

For input parameters a plug-in can accept either one or more structured request attributes or a per request input data with an InputStream containing an opaque blob of data.

* Structured request attributes can be thought of as a key value pair map of values. The users of the plug-in can specify a single file (e.g. CSV, JSON) with a record containing the values for each request to be executed by the plug-in. For example each address to geocode would have one record in the input file and the new instance plug-in will be executed for each record. In addition to request level parameters there are also job parameters that apply to all requests in the job. The CPF will convert the input data from the file format specified by the user to Java objects for use by the plug-in.
* Per request input data is used where the data is more complex than key value pairs, or is a binary blob of data. Each request is specified using a separate file stream or URL to the binary blob of data. For example a face detection plug-in would take a JPEG file as an input parameter for each face to recognize. Per request input data plug-ins cannot also have request attributes, although they may have job attributes. If structured request attributes and binary blobs are required for the same plug-in the plug-in can accept a URL to the input data and process that URL within the plug-in.

For result fields the plug-in can either return a single record of structured result data, a list of structured result data or a single binary blob of data,

* Like structured request attributes structured result attributes are a key value pairs. The CPF will create a single file in the requested output file for the user to download with one record for each result returned from the plug-in.
* The list of structured results also creates a single output file for the user to download. There will be one record for each entry in the list of results, with a relative result number showing the order of the result in the list of results.
* Like per request input data, per request result data returns one binary blob of data for each request. There will be one file for the user to download for each request. For example a WMS service would return a JPEG, or PNG for each map to render.

More detail on implementing these request parameters and result data attributes are described in the remainder of this section.

## Create a plug-in project

The first step is to create a plug-in project. Plug-ins must be developed using Apache Maven 3.0.x.

A new plug-in project can be created using the Maven archetype mechanism. The archetype defines a template project that can be created by passing in parameters to the following command.

cd /projects[[2]](#footnote-2)

mvn \

archetype:generate \

-DinteractiveMode=false \

-DarchetypeGroupId=ca.bc.gov.open.cpf \

-DarchetypeArtifactId=cpf-archetype-plug-in \

-DarchetypeVersion=**4.0.0-SNAPSHOT** \

-DgroupId=**ca.bc.gov** \

-DartifactId=**demo** \

-Dversion=**1.0.0-SNAPSHOT** \

-Dpackage=**ca.bc.gov.demo \**

-Dplug-inName=**Demo \**

-Dplug-inAcronym=**demo**

|  |  |
| --- | --- |
| Parameter | Description |
| archetypeVersion | The most recent version of the CPF framework. |
| groupId | The maven group identifier also used as the java package name for the plug-in class. For BC Government plug-ins this should be ca.bc.gov. |
| artifactId | The base maven artifact identifier used for the maven modules created in the project. This should be the project acronym for BC Government projects. |
| version | The version identifier you’d like to give to your plug-in. Must include -SNAPSHOT the -SNAPSHOT is removed on migration to test. |
| package | The Java package name for the generated code. Must be ca.bc.gov.<acronym> for BC Government projects. |
| plug-inName | The name of the plug-in. This will be used to generate the plug-in class name and for descriptions in the Maven build files. |
| plug-inAcronym | The acronym of the plug-in. This will be used to generate the plug-in jar name. |

The following directory structure would be created if the command were run using the parameters above.

| File/Directory | Description |
| --- | --- |
| demo/ | The root directory of the plug-in project. |
| scripts/ | Any SQL scripts required to create the database or data required by the plug-in. Samples are included. |
| README.txt | A sample readme file for deployment to the Ministry. Edit for your application. |
| pom.xml | The maven build file for the plug-in. |
| src/ |  |
| main/java/ca.bc.gov/demo/ | The java package for the plug-in, add any support classes in here. |
| DemoPlug-in.java | The java class for the plug-in. |
| main/resources/ | The non-java resource files to include in the plug-in jar. |
| META-INF/ | The META-INF directory for the plug-in jar. |
| ca.bc.gov.open.cpf.plugin.sf.xml | The main spring configuration file for the plug-ins in this module. |
| ca/bc/gov/demo/Demo.sf.xml | The spring configuration file containing any beans used by the plug-ins. |

## Add Dependencies

A plug-in can include dependencies to additional libraries that are not deployed as part of the core CPF application. These must be approved prior to delivering to the Ministry.

All plug-ins must either be available in the Maven central repository <http://search.maven.org/> or in the CITZ BC Government Maven Repository. Contact your BA to get any non-standard libraries added to the CITZ Maven repository.

If any of the dependencies include dependencies to logging frameworks such as LOG4J, SLF4J and especially commons-logging these must be excluded from those dependencies[[3]](#footnote-3). This ensures that carefully controlled versions of these files are used. The CPF internally uses SLF4J as the logging API with a LOG4J binding to perform the actual logging and a bridge so that commons logging is logged via SLF4J. It is recommended for plug-ins to use SLF4J for logging in their module. Plug-ins must not include a log4j.xml file in their Jar file. A log4j.xml file can however be included in src/test/resources for testing the plug-in.

Any dependencies that are included in the CPF application itself must be marked with a scope of provided. If they are dependencies of other modules then they should be excluded from that dependency. CPF ignores any dependencies to the CPF libraries or the libraries it uses. This ensures that the correct versions of these libraries are used at runtime.

Any dependencies to test frameworks such as junit must be included using the test scope so that they are not included in the jar. Also any test code must be included below src/test as opposed to src/main.

## Write Plug-in Class

A business application plug-in is implemented as a Java class. The class is annotated[[4]](#footnote-4) with CPF specific annotations on the plug-in’s classes and methods. A single plug-in project can have more than one plug-in class, one for each plug-in provided by that module.

An instance of the plug-in class is created for each request processed by the plug-in. Therefore it must not include the initialization of any resources that have significant overhead to create. These should be defined as spring beans and made available to the plug-in using spring dependency injection. If there are data structures that vary based on the parameters to a request then these can be created within the plug-in.

### Plug-in Spring Configuration File

The CPF business application plug-in classes can be registered with the CPF by including the **plug-in definition spring** file META-INF/ca.bc.gov.open.cpf.plugin.sf.xml in the jar file for the plug-in.

The following file shows an example of this file.

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

<beans

xmlns="http://www.springframework.org/schema/beans"

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

xmlns:util="http://www.springframework.org/schema/util"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util-3.0.xsd

"

>

<util:list id="**beanImports**">

<value>**classpath:/ca/bc/gov/demo/Demo.sf.xml**</value>

</util:list>

<bean

id="**demo**"

class="**ca.bc.gov.demo.DemoPlug-in**"

p:**dataSource**-ref="**demoDataSource**"

scope="**prototype**" />

</beans>

This file may only contain bean definitions for the plug-ins and an optional **beanImports** list.

The bean definitions must have the id attribute equal to the business application plug-in name defined in the plug-in class. The class must be the fully qualified class name of the plug-in class. The scope on the bean must be set to prototype. This ensures that a new instance of the plug-in be created on each request to get the bean. The p:<propertyName>[[5]](#footnote-5) or p:<propertyName>-ref attribute styles can be used to inject dependencies into the bean when it is created.

If the plug-in requires other beans such as JDBC data sources or a caching service that are expensive to create these must be defined in **plug-in resource spring** configuration files. The plug-in archetype includes a blank spring file for this purpose. The **beanImports** list includes a reference to each **plug-in resource spring** file used by the plug-ins. This is used instead of the spring **import** mechanism so that the beans defined in those files are only instantiated on the worker nodes. Within the **plug-in definition spring** file no regular spring imports can be used, spring imports can however be used in the **plug-in resource spring** files. Additional **beanImports** can be added if required.

### Class annotations

The instance of the plug-in is executed within a single thread so does not need to be synchronized. Any services that it uses must however be thread safe as they will be used by multiple instances of the plug-in in different threads.

### Plug-in Resource Beans

A new instance of the plug-in class is created for each request processed in a job. Therefore the plug-in class should not perform any complex instantiation in the constructor. Instead any resources (such as data sources) or a caching service should be defined as singleton beans in the **plug-in resource spring** files.

The plug-in bean would then have a property (set/get method pair) for each resource bean it uses.

private DataSource dataSource;

private void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

private DataSource getDataSource() {

return dataSource;

}

The com/mycompany/demo/Demo.sf.xml **plug-in resource spring** file would define the bean as shown below. Note that this example introduces the JdbcDataSourceFactoryBean. This can be used to create a pooling data source without needing to know the underlying database data source used. By changing the URL to an Oracle JDBC URL it will create an Oracle data source. The config properties are set on the data source instance. Consult the database vendor's documentation for the available parameters.

<bean

id="demoDataSource"

class=" com.revolsys.jdbc.io.JdbcDataSourceFactoryBean"

p:url="**jdbc:postgresql://localhost:5432/postgres**"

p:username="**demo**"

p:password="**\*\*\*\*\*\*\*\***"

lazy-init="true"

>

<property

name="config"

>

<map>

<entry

key="initialConnections"

value="**0**" />

<entry

key="maxConnections"

value="**10**" />

</map>

</property>

</bean>

To use the data source factory you'll need to include one of the following dependencies in your plug-in. Note that the version uses a property obtained from the cpf-parent module that is the parent of all plug-ins created by the archetype. Therefore this value will never need to change.

**PostgreSQL**

<dependency>

<groupId>com.revolsys.open</groupId>

<artifactId>com.revolsys.open.gis.**postgresql**</artifactId>

<version>${com.revolsys.open.version}</version>

</dependency>

**Oracle**

<dependency>

<groupId>com.revolsys.open</groupId>

<artifactId>com.revolsys.open.gis.**oracle**</artifactId>

<version>${com.revolsys.open.version}</version>

</dependency>

Finally the demoDataSource bean will be injected into the dataSource property on the demo bean definition in the **plug-in definition spring** file.

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

<beans

xmlns="http://www.springframework.org/schema/beans"

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

xmlns:util="http://www.springframework.org/schema/util"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/util

http://www.springframework.org/schema/util/spring-util-3.0.xsd

"

>

<util:list id="**beanImports**">

<value>**classpath:/com/mycompany/demo/Demo.sf.xml**</value>

</util:list>

<bean

id="**demo**"

class="**ca.bc.gov.demo.DemoPlug-in**"

p:**dataSource**-ref="**demoDataSource**"

scope="**prototype**" />

</beans>

Any required dependencies can be injected using the basic approach shown above.

### Customization Properties

Plugins can customize certain aspects of the files generated by the CPF for structured results. For example the URL to the style document to be used in KML documents.

The customization properties can be defined using an optional properties map bean in the ca.bc.gov.open.cpf.plugin.sf.xml file.

The map can contain a default entry, or an entry for each business application name. The value for each entry is a map of the customization properties. If the property is not specified for a business application then the value from the default entry is used.

<util:map id=*"****properties****"*>

<entry key=*"****default****"*>

<map>

<entry

key=*"****kmlPlaceMarkNameAttribute****"*

value=*"****fieldA****"* />

</map>

</entry>

<entry key=*"****demo****"*>

<map>

<entry

key=*"****kmlPlaceMarkNameAttribute****"*

value=*"****fieldB****"* />

</map>

</entry>

</util:map>

Customization properties can also be specified on a plugin class or on the result object for plugins that return a list of results. This allows the property to be calculated for each result. For example the kmlStyleUrl may vary based on the type of result. The following example show the optional method that must be implemented for plugins that wish to define per result configuration properties.

public Map<String,Object> getCustomizationProperties {

Map<String,Object> properties = new HashMap<String,Object>();

properties.put("...", ...);

return properties;

}

The following customization properties are supported.

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| kmlDocumentName | String | The text to use as the name of the KML Document. **Don't use on a per result basis.** |
| kmlDocumentDescription | String | The text to use as the description of the KML Document. **Don't use on a per result basis.** |
| kmlPlacemarkDescription | String | The text to use as the description of the KML Placemark created for each result. |
| kmlSnippet | String | The text to use as the snippet of the KML Placemark created for each result. |
| kmlStyle | String | A list of one or more KML Style tags containing a custom style for the whole document. **Don't use on a per result basis.** External styles should be preferred. |
| kmlStyleUrl | String | The text to use as the styleUrl of the KML Placemark created for each result. |

See the KML reference guide for details on how to use the KML attributes.

<https://developers.google.com/kml/documentation/kmlreference>

### Plug-in Job Parameters & Structured Request Parameters

A plug-in can implement parameters that the user can specify when submitting a job. Parameters can either be global for all the requests in a job or specific to an individual request within the job.

Parameters are implemented as Java bean properties on the class. The plug-in will implement a set property method for each parameter. The parameter name is the name of the Java bean property. Job and Request Parameters can only use the following data types byte, short, int, long, float, double, boolean, String, URL. Request parameters can also use JTS Point, LineString, Polygon, MultiPoint, MultiLineString and MultiPolygon.

Parameters for the whole job use the ca.bc.gov.open.cpf.plugin.api.JobParameter annotation and parameters for each request use the ca.bc.gov.open.cpf.plugin.api.RequestParameter annotation. Both annotations can be specified on a single set property method to indicate that the parameter can be specified either on the job or on a request. The ca.bc.gov.open.cpf.plugin.api.Required annotation is used to mark the parameter as being required. If a value is not specified for a required parameter the scheduler will exclude that request from being processed and an error returned.

Both job and request parameter annotations also support the description, index, length and scale attributes. The description is used as Input help text on the form. The index defines the order of the parameters on the HTML form. The length is the size of the field (number of characters or digits). The scale is the number of decimal places.

The ca.bc.gov.open.cpf.plugin.api.DefaultValue annotation can be used to specify the default value to be displayed on the CPF job submission form or to be set on the plugin if the user did not specify the value. The value must be specified as a string. The CPF will attempt to convert this string to the data type of the parameter.

The ca.bc.gov.open.cpf.plugin.api.AllowedValues annotation can be used to specify the list of allowed values that can be used for the parameter. The field on the job submission form will be displayed as a select list containing these values.

private int value;

@RequestParameter(

index=2,

description="The value to calculate the square for (1-4)"

)

@JobParameter

@DefaultValue("2")

@AllowedValues(value = {"1", "2", "3", "4"})

@Required

public void setValue(int value) {

this.value = value;

}

The ca.bc.gov.open.cpf.plugin.api.RequestParameter annotation cannot be used for plug-ins which accept per request input data perRequestInputData=true.

See the following class for an example.

<http://apps.bcgov/wsvn/CITZ.cpf/api-source/trunk/cpf-plug-ins-samples/src/main/java/ca/bc/gov/open/cpf/plug-in/samples/WebMapImage.java>

### Per request input data

Plug-ins that accept per request input data perRequestInputData=true must implement the following methods to set the content type of the binary blob and URL which can be used to access the binary blob of the input data for that request. The input stream for the URL can be read in the execute method of the plug-in. When getting the input stream for the URL any HTTP redirects must be followed to get the final content to read.

private URL inputDataUrl;

private String inputDataContentType;

public void setInputDataUrl(final URL inputDataUrl) {

this.inputDataUrl = inputDataUrl;

}

public void setInputDataContentType (final String inputDataContentType) {

this.inputDataContentType = inputDataContentType;

}

See the following class for an example.

[http://apps.bcgov/wsvn/CITZ.cpf/api-source/trunk/cpf-plug-ins-samples/src/main/java/ca/bc/gov/open/cpf/plug-in/samples/Digest.java](https://apps.gov.bc.ca/int/wsvn/CITZ.cpf/api-source/trunk/cpf-plugins-samples/src/main/java/ca/bc/gov/open/cpf/plugin/samples/Digest.java)

### Structured Result Attributes

Plug-ins that do not accept per request result data perRequestResultData=false can return structured result attributes. Result attributes are implemented as Java bean properties on the class. The plug-in will implement a get or is property method for each result attribute. The result attribute name is the name of the Java bean property. Parameters can only use the following data types byte, short, int, long, float, double, boolean, String, URL, JTS Point, LineString, Polygon, MultiPoint, MultiLineString and MultiPolygon.. The result attributes will be converted by the CPF to the correct representation in the output format or a string representation of the value before being returned to the user.

The ca.bc.gov.open.cpf.plugin.api.ResultAttribute annotation marks a get or is property method as being a result attribute. Result attribute annotations also support the description, index, length and scale attributes. The description is used as Input help text on the business application description page. If the description is omitted and there was a job or request parameter of the same name then the description will be taken from the parameter. The index defines the order of the attributes in the result file. The length is the size of the field (number of characters or digits). The scale is the number of decimal places.

private int square;

@ResultAttribute(

index=2,

description="The square of a value"

)

public int getSquare() {

return square;

}

See the following class for an example.

<http://apps.bcgov/wsvn/CITZ.cpf/api-source/trunk/cpf-plug-ins-samples/src/main/java/ca/bc/gov/open/cpf/plug-in/samples/UserInfo.java>

### List of Structured Result Attributes

Plug-ins that do not accept per request result data perRequestResultData=false can return a list of structured result attributes if the plug-in returns multiple results instead of a single result.

To return a list of results the plug-in must implement a get method that returns a list of a specified value object (e.g. ResultObject). The method must also have the annotation ca.bc.gov.open.cpf.plugin.api.ResultList. The plug-in class must not have any methods marked with the ResultAttribute annotation.

private List<ResultObject>

@ResultList

public List<ResultObject> getResults() {

return results;

}

The value object class contains the result attributes to return for each result in the list. Pick an appropriate name for the value object (e.g. Address). The value object class must have one or more get methods with the ca.bc.gov.open.cpf.plugin.api.ResultAttribute annotation. These are the attributes that are returned to the client. The same rules apply as per structured result attributes.

public class ResultObject

private String attribute1;

@ResultAttribute

public int getAttribute1 () {

return attribute1;

}

See the following classes for an example.

<http://apps.bcgov/wsvn/CITZ.cpf/api-source/trunk/cpf-plug-ins-samples/src/main/java/ca/bc/gov/open/cpf/plug-in/samples/BcGeocoder.java>

<http://apps.bcgov/wsvn/CITZ.cpf/api-source/trunk/cpf-plug-ins-samples/src/main/java/ca/bc/gov/open/cpf/plug-in/samples/BcGeocoderResult.java>

### Structured Data Geometry Request Parameters and Result Attributes

The CPF includes spatial support for structured input data and result data files. This enables plug-ins to define job parameters, request parameters and result attributes to use Java Topology Suite (JTS)[[6]](#footnote-6) Geometry objects. The CPF reads the input data files (e.g. ESRI Shapefile in a Zip file) and converts the values into JTS Geometry objects for processing by the business application. After processing the JTS Geometry objects are converted back into the requested output spatial file format. The CPF also handles projection of the geometries if required.

The CPF can use Extended Well-Known Text[[7]](#footnote-7) (EWKT) geometry strings for geometries in the input data files submitted by the user or the result data files generated by the CPF. This can be used in non-spatial file formats or where there are multiple geometries in a spatial file format that only supports a single geometry. These EWKT geometries will be converted to JST Geometry objects. EWKT geometries are also used for job parameters as they are passed as HTML form parameters. We however don't recommend using geometries as job parameters.

#### Geometry Job & Request Parameters

Adding a geometry job parameter or request parameter is exactly the same as any other business application parameter. The parameter type must be a JTS Geometry class or subclass. The only difference is that the GeometryConfiguration attribute above can also be specified. If the configuration is not set on the method then the annotation from the class is used.

The following example shows a geometry parameter with a geometry configuration. The plug-in expects the geometry in BC Albers (3005). If the user submits a WGS84 (4326) geometry it will be projected to BC Albers before being passed to the plug-in. The number of axis is 3 so the geometry will always have x, y, z values with Double.NaN if the source geometry was 2D. The scaleFactorXy is 1000, which indicates a 1mm precision (1/1000 = 0.001), the source geometry will be rounded to this precision. The scaleFactorZ is 1, which indicates a 1m precision, the source geometry will be rounded to this precision. As validate is true the geometry will be checked using the OGC is valid predicate. If the validation fails the plug-in will not be executed. Finally the geometry is marked as being the primary geometry.

@RequestParameter

@GeometryConfiguration(

srid = **3005**,

numAxis = **3**,

scaleFactorXy = **1000**,

scaleFactorZ = **1**,

validate = **true**,

primaryGeometry = **true**)

public void setGeometry(final Geometry geometry) {

this.geometry = geometry;

}

#### Geometry Result Attributes

Adding a geometry result attribute is exactly the same as any other business application result attributes. The return type must be a JTS Geometry[[8]](#footnote-8) class or subclass. The only difference is that the GeometryConfiguration attribute above can also be specified. If the configuration is not set on the method then the annotation from the class is used.

The following example shows a geometry result attribute with a geometry configuration. The plug-in indicates that the user will get the geometry in BC Albers (3005). If the plug-in returns a WGS84 (4326) geometry it will be projected to BC Albers before being returned to the use. The number of axis is s so the geometry will only have x, y, if the plug-in returned a 3D geometry the z-value will be removed. The scaleFactorXy is 1000, which indicates a 1mm precision (1/1000 = 0.001). The geometry will be rounded to this precision. The scaleFactorZ is 1, which indicates a 1m precision. The geometry will be rounded to this precision. As validate is true the geometry will be checked using the OGC is valid predicate. If the validation fails the plug-in will not be executed. Finally the geometry is marked as being the primary geometry.

**NOTE:** All geometries created by a business application must have the SRID set on the geometry object.

The user when creating a job can override the srid, numAxis, scaleFactorXy and scaleFactorZ. The required conversion will be performed by the CPF. The business application can implement any one of the following methods to receive the values requested by the user. This might be useful if the application wants to perform it's own projection or reduce work by not creating z-values if they are not required.

public void setResultNumAxis(final int resultNumAxis) {

this.resultNumAxis = resultNumAxis;

}

public void setResultScaleXy(final double resultScaleXy) {

this.resultScaleXy = resultScaleXy;

}

public void setResultScaleZ(final double resultScaleZ) {

this.resultScaleZ = resultScaleZ;

}

public void setResultSrid(final int resultSrid) {

this.resultSrid = resultSrid;

}

#### Geometry Processing

The CPF provides access to an extended version of the JTS GeometryFactory, use the following import to access the geometry factory. The geometry factory provides projection support, precision models and controls in the number of axis.

import com.revolsys.gis.cs.GeometryFactory;

In general a geometry factory should be obtained using the getFactory static methods on the geometry factory class. This ensures that only one instance of a factory is created where the same parameters are used. The following methods are available.

**// No coordinate system, 3D, floating precision**

getFactory()

***// No coordinate system, 3D, precise x, y, floating z***

getFactory(double scaleXy)

**// Get the geometry factory from an existing geometry**

getFactory(Geometry geometry)

**// Srid coordinate system, 3D, floating precision**

getFactory(int srid)

**// Srid coordinate system, 2D, precise x, y**

getFactory(int srid, double scaleXy)

**// Srid coordinate system, 3D, precise x, y, z**

getFactory(int srid, double scaleXy, double scaleZ)

**// Srid coordinate system, num axis, floating precision**

getFactory(int srid, int numAxis)

**// Srid coordinate system, num axis, precise x, y, z**

getFactory(int srid, int numAxis, double scaleXy, double scaleZ)

The following example shows how to create a geometry from a WKT or EWKT string. If the EWKT string includes a SRID the geometry will use read using that SRID and then projected to the SRID of the geometry factory.If the SRID was not specified the geometry will be assumed to be in the coordinate system of the geometry factory's SRI. The return type of the WKT to geometry conversion will be auto-casted to the type of the variable the result is assigned to. Use Geometry as the type if it is not possible to guarantee that the WKT is that geometry type.

GeometryFactory geometryFactory = GeometryFactory.getFactory(3005, 1.0);

String wkt = "SRID=4326;POLYGON((-122 50,-124 50,-124 51,-122 51,-122 50))";

Polygon polygon = geometryFactory.createGeometry(wkt);

System.out.println(polygon);

// POLYGON((1286630 561884,1143372 555809,1140228 667065,1280345 673006,1286630 561884))

Geometries can also be created from a double array of coordinates and from lists of other geometries. There are two many methods to list here, but the following gives a basic set to start with. When dealing with double arrays the array must contain only the number of axis supported by the geometry factory. For example a 2D geometry will have x1,y1...,xN,yN values and a 2D x1,y1,z1...,xN,yN,zN.

**// Create a geometry copy, converting the geometry to the srid and precision of the**

**// factory**

Geometry createGeometry(Geometry geometry)

**// Create a point**

Point createPoint(double... coordinates)

**// Create a lineString**

LineString createLineString(double... coordinates)

**// Create a linearRing**

LinearRing createLinearRing(double... coordinates)

**// Create a polygon, exterior ring followed by 0 or more interior rings**

**// rings can be double[], LineString, LinearRing, CoordinatesList or CoordinateSequence**

Polygon createPolygon(List<?> rings)

**// Create a multi point, points can be double[], Coordinate, Coordinates, Point,**

**// CoordinatesList or CoordinateSequence**

MultiPoint createMultiPoint(List<?> points)

**// Create a multi line string, lines can be double[], LineString, CoordinatesList or**

**// CoordinateSequence**

MultiLineString createMultiLineString(List<?> lines)

**// Create a multi polygon, polygons can be Polygon or a sub List for each polygon**

**// (see createPolygon)**

MultiPolygon createMultiPolygon(List<?> polygons)

### Per request result data

Plug-ins that accept per request input data perRequestResultData=true cannot return result attributes. They can only return the binary content of the result data. The plug-in must implement the following methods. The resultData output stream can be used to write the result binary blob data. The resultDataContentType indicates to the plug-in the type of result data it should generate.

private OutputStream resultData;

private String resultDataContentType;

public void setResultData(final OutputStream resultData) {

this.resultData = resultData;

}

public void setResultDataContentType(final String format) {

this.resultDataContentType = format;

}

See the following class for an example.

<http://apps.bcgov/wsvn/CITZ.cpf/api-source/trunk/cpf-plug-ins-samples/src/main/java/ca/bc/gov/open/cpf/plug-in/samples/WebMapImage.java>

### Execute method

The plug-in must implement a public void execute() method. When the CPF has set all the parameters on the plug-in it invokes the execute method so that the plug-in can perform the request using the parameters and generate the result.

The following is a simple example of an execute method that takes the value request field, calculates the square and stores this in the square result attribute.

public void execute() {

this.square = this.value \* this.value;

}

The following is an example of a per request input data execute method. The plug-in gets the input stream from the inputDataUrl, performs some processing on the data and stores the result in the digest result attribute.

try {

MessageDigest digester = MessageDigest.getInstance("MD5");

InputStream in = **this.inputDataUrl**.openStream();

byte[] buffer = new byte[4096];

for (int count = in.read(buffer); count != -1; count = in.read(buffer)) {

digester.update(buffer, 0, count);

}

byte[] data = digester.digest();

**this.digest** = new String(Hex.encodeHex(data));

} catch (NoSuchAlgorithmException e) {

throw new IllegalArgumentException("Cannot find digest algorithm "

+ algorithmName, e);

} catch (IOException e) {

throw new RuntimeException("Cannot read input data", e);

}

The following is an example of a per request result data execute method. The plug-in creates query string parameters from the plug-in request parameters and creates a connection to a web service. The plug-in then writes the response from the web service to the result data.

public void execute() {

Map<String, Object> parameters = new LinkedHashMap<String, Object>();

parameters.put("SERVICE", "WMS");

parameters.put("VERSION", "1.1.1");

parameters.put("REQUEST", "GetMap");

parameters.put("LAYERS", **this.layers**);

parameters.put("STYLES", **this.styles**);

parameters.put("CRS", **this.crs**);

parameters.put("BBOX", **this.bbox**);

parameters.put("WIDTH", **this.width**);

parameters.put("HEIGHT", **this.height**);

parameters.put("FORMAT", **this.resultDataContentType**);

parameters.put("EXCEPTIONS", "INIMAGE");

String url = UrlUtil.getUrl(**this.wmsUrl**, parameters);

try {

InputStream in = new URL(url).openStream();

try {

FileUtil.copy(in, **this.resultData**);

} finally {

FileUtil.closeSilent(**this.resultData**);

FileUtil.closeSilent(in);

}

} catch (MalformedURLException e) {

throw new IllegalArgumentException(url + " is not valid URL");

} catch (IOException e) {

throw new RuntimeException("Unable to get map", e);

}

}

## Plug-in Security

The CPF provides a Security service that allows a plug-in to query information about the user, check group memberships and check permission to access resources.

### Security Groups

The CPF uses groups of users to manage the security policies for plug-ins. All policies are associated with security groups as opposed to individual users.

See the CPF Admin Guide for more detail on managing groups using the CPF admin application.

There are three types of user group.

The USER\_TYPE user groups are virtual groups that indicate the type of user. In the BC Government infrastructure the following USER\_TYPE groups are supported.

|  |  |
| --- | --- |
| Name | Description |
| BCGOV\_ALL | BC Government All Users |
| BCGOV\_BUSINESS | BC Government External Business Users |
| BCGOV\_EXTERNAL | BC Government External Users |
| BCGOV\_INDIVIDUAL | BC Government External Individual Users |
| BCGOV\_INTERNAL | BC Government Internal Users |
| BCGOV\_VERIFIED\_INDIVIDUAL | BC Government External Verified Individual |

The GLOBAL user groups are created and managed by the CPF security administrator using the admin application link shown below. These are shared across all the plug-ins in the system. There are no global groups installed by default.

<http://localhost/cpf/secure/admin/userGroups/>

Each plug-in can also have their own user groups that are specific to that plug-in and not shared between plug-ins. Module specific groups must have the <Module Name>\_ prefix (e.g. the partner group for the BGEO module would be BGEO\_PARTNER).

1. Open <http://localhost/cpf/admin/> using a web browser. Use the username CPF\_ADMIN and password cpf\_2009.
2. On the menu of the left select Modules.
3. Click on the name of the module in the table, this will open the module view page.
4. Click the [+] next to User Groups to view the groups available for the module, it will include USER\_TYPE, GLOBAL and module specific groups.
5. Click the name of a group to view it and then click Edit to change the group name or description. Editing is only possible for module specific groups.
6. Or click Add to add a new module user group.
7. On the View user group page click the [+] next to User Accounts for Group to add or remove users from the group. Adding users to a group is only possible for module specific groups.

### Security Permissions

The CPF allows security permissions to be granted to groups of users. Security permissions grant members of a group permission to perform a specified action on a resource. Negative permissions that deny access to a resource are not supported.

|  |  |  |
| --- | --- | --- |
| Attribute | Example | Description |
| resourceClass | reportName | The resourceClass is used to categorize the type of resource that the policy applies to. This could be the same name as the plug-in parameter that contains the resource identifier to check. |
| resourceId | Demo Report | The resourceId is the identifier of the resource the |
| action | View | The name of the action that can be performed on the resource. Use common names such as View, Edit for the action name. |

The special value 'All' can be used as a wildcard for the resourceClass, resourceId or action. When enforcing permissions the 'All' value will match any requested value for that attribute when the plug-in uses the security service to check if a user can access a resource.

Security permissions can be granted

1. Open <http://localhost/cpf/admin/> using a web browser. Use the username CPF\_ADMIN and password cpf\_2009.
2. On the menu of the left select Modules.
3. Click on the name of the module in the table, this will open the module view page.
4. Click the [+] next to User Groups to view the groups available for the module, it will include USER\_TYPE, GLOBAL and module specific groups.
5. Click the name of a group to view it.
6. On the View user group page click the [+] next to User Group Permissions to add, view and edit permissions.

### Security Configuration File

In addition to manually defining security groups and permissions using the CPF Admin Application a plug-in can include a configuration file in the plug-in jar. The following must be created as a JSON document.

src/main/resources/META-INF/ca.bc.gov.open.cpf.plugin.UserGroups.json

The User Groups config file is a JSON list of JSON objects. Each object represents a user group to be created or to create permissions for. Only groups starting with the <Module Name>\_ prefix will be created. The USER\_TYPE or GLOBAL groups must already exist for permissions to be added to them.

|  |  |
| --- | --- |
| Attribute | Description |
| name | The name of the group to create or grant permissions to (e.g. DEMO\_PARTNER). |
| permissions | A JSON list of JSON objects containing the permissions to grant. See table below for list of permission attributes. |

The following table shows the JSON attributes for a permission object.

|  |  |  |
| --- | --- | --- |
| Attribute | Example | Description |
| resourceClass | reportName | The resourceClass is used to categorize the type of resource that the policy applies to. This could be the same name as the plug-in parameter that contains the resource identifier to check. |
| resourceId | Demo Report | The resourceId is the identifier of the resource the |
| action | View | The name of the action that can be performed on the resource. Use common names such as View, Edit for the action name. |

The following example shows a group called DEMO\_PARTNER that will be created if it does not exist. The members of that group will be granted the permission to View the Demo Report. Members of the BCGOV\_INTERNAL group will be granted permission to View All reports.

[

{

"name": "**DEMO\_PARTNER**",

"permissions": [

{

"resourceClass" : "**reportName**",

"resourceId" : "**Demo Report**",

"action" : "**View**"

}

]

},

{

"name": "**BCGOV\_INTERNAL**",

"permissions": [

{

"resourceClass" : "**reportName**",

"resourceId" : "**All**",

"action" : "**View**"

}

]

}

]

## Plug-in Configuration

Plug-ins can include a default configuration file that contains any configuration that will change from environment to environment. Such as database connection URLs and passwords. The configuration file provided in the application is a template that is used to populate the CPF configuration database with initial values for the administrator to edit with the values for that environment.

**NOTE:** Plug-ins must fully document all supported configuration properties in their readme file and other documentation where appropriate.

**NOTE:** Plug-ins **must not** require a configuration file to be created on the file system and they **must not** require manual editing of source code for specific environments. The same code must be compiled only once for use in all environments.

If a plug-in requires configuration properties the following JSON file must be created in the maven module for the project.

src/main/resources/META-INF/ca.bc.gov.open.cpf.plugin.ConfigProperties.json

[

{

"name": "**name1**",

"type": "**string**",

"value": "**value1**"

},

{

"name": "**name2**",

"type": "**int**",

"value": "**value2**"

},

]

The configuration file is a JSON list containing JSON objects for each property. Each configuration JSON object represents one configuration property and must include the following attributes.

| Attribute | Description |
| --- | --- |
| name | The name of the property (or sub property). The name can either be placeholders[[9]](#footnote-9) in the spring configuration file or can override[[10]](#footnote-10) bean properties from the spring configuration file. The recommendation is to reduce the number of placeholders and use overrides instead. Include default values in the spring bean files. |
| type | The Java data type (e.g. String, byte, short, int, long, float, double, boolean) of the value. Must be the same type as the Java set property method. |
| value | The string value of the property, it must be possible to convert this value to the Java data type specified in the type attribute. |

Consider the example of a data source defined with the following bean definition.

<bean

id="demoDataSource"

class=" com.revolsys.jdbc.io.JdbcDataSourceFactoryBean"

p:url="**jdbc:postgresql://localhost:5432/postgres**"

p:username="**demo**"

p:password="**\*\*\*\*\*\*\*\***"

lazy-init="true"

>

<property

name="config"

>

<map>

<entry

key="initialConnections"

value="**0**" />

<entry

key="maxConnections"

value="**10**" />

</map>

</property>

</bean>

To override the URL and the password the following properties would be created.

[

{

"name": "demoDataSource.url",

"type": "string",

"value": " **jdbc:postgresql://localhost:5432/demo**"

},

{

"name": "demo.timeout",

"type": "int",

"value": "10"

},

]

Once a module has been deployed the configuration properties can be modified using the CPF admin application. See the CPF Admin Guide for more details.

1. Open <http://localhost/cpf/admin/> using a web browser. Use the username CPF\_ADMIN and password cpf\_2009.
2. On the menu of the left select Modules.
3. Click on the name of the module in the table, this will open the module view page.
4. Click the [+] next to Module Config Properties to view the current properties.
5. Click the id of a property to view it and then click Edit to change the value.
6. Or click Add to add a new property.

## Build the plug-in

The plug-in is built using maven. Use the following command to create a clean build that is deployed to your local maven repository.

mvn clean install

The plug-in must be deployed to a maven repository so that the CPF can download the plug-in. If on a developers workstation a local maven cache can be used instead of this step.

mvn deploy[[11]](#footnote-11)

For BC Government projects the plug-in is built and deployed to the Maven repository via the Jenkins tool. See the example README.txt for more details.

<http://delivery.apps.gov.bc.ca/gov/cis/>

## Deploy a Plug-in

Plug-ins are deployed dynamically using the CPF admin application. Follow this procedure to deploy a new model.

1. Open <http://localhost/cpf/admin/> using a web browser. Use the username CPF\_ADMIN and password cpf\_2009.
2. On the menu of the left select Modules
3. Click the Add button on the Modules page.
4. Enter the project acronym as the module name (e.g. demo), it will be converted to upper case.
5. Enter in the maven module Id for your plug-in in the format <groupId>:<artifactId>:<version>. For example **com.myapplication**:**demo**:**1.0.0-SNAPSHOT**.
6. Click the Save button
7. The module view page will be displayed.
8. Refresh the page if 'Started' is a cross and no 'Module error' is displayed to confirm that the module was loaded successfully.

## Testing a Plug-in

The business application plug-in can be tested by creating a test harness, using the CPF web site or using a CPF client to connect to the CPF web services.

### CPF Web Site

If the CPF is deployed to /cpf on localhost the list of business applications can be found using the following page.

<http://localhost/cpf/ws/apps/>

Click on the link for a business application to get the list of resources for that application. Depending on the plug-in the following resources maybe available.

|  |  |
| --- | --- |
| Resource | Description |
| specification | The HTML page describing the parameters and result attributes of the business application. NOTE page is not yet finished. |
| instant | If the plug-in allows instant mode there will be a form that allows a user to submit a single request and get the results back immediately. |
| single | The single request form allows the parameters for a single request to be entered using form fields instead of requiring a file of structured input data. Unlike the instant mode the request is processed via a cloud job. |
| multiple | The multiple request form contains fields for the job parameters and accepts a file upload or a URL for the request data fields. |

Use one of the above forms to create a new job and then download the results to confirm that the plug-in works as expected.

### Test Harness

The BusinessApplicationPluginExecutor class allows developers to execute a single request against their plug-in without deploying it to the CPF infrastructure. The executor converts the input parameters to a JSON object and back again to simulate what happens in the internal CPF processing.

To create a test harness construct an instance of ca.bc.gov.open.cpf.plugin.api.BusinessApplicationPluginExecutor and then use one of the following methods to invoke the plug-in.

/\*\*

\* Execute a {@link BusinessApplicationPlugin} which accepts and returns

\* structured data.

\*

\* @param businessApplicationName The name of the business application to

\* execute.

\* @param inputParameters The input parameters to the business application.

\* @return The result parameters from the business application.

\*/

public Map<String, Object> execute(

String businessApplicationName,

Map<String, Object> inputParameters

);

/\*\*

\* Execute a {@link BusinessApplicationPlugin} which accepts and returns

\* structured data.

\*

\* @param businessApplicationName The name of the business application to

\* execute.

\* @param inputParameters The input parameters to the business application.

\* @return The result parameters from the business application.

\*/

public List<Map<String, Object>> executeList(

String businessApplicationName,

Map<String, Object> inputParameters

);

/\*\*

\* Execute a {@link BusinessApplicationPlugin} which accepts structured data

\* and returns opaque data.

\*

\* @param businessApplicationName The name of the business application to

\* execute.

\* @param inputParameters The input parameters to the business application.

\* @param resultDataContentType The content type of the result data.

\* @param resultData The output stream that the plug-in can use to write the

\* result data to.

\*/

public void execute(

String businessApplicationName,

Map<String, Object> inputParameters,

String resultDataContentType,

OutputStream resultData

);

/\*\*

\* Execute a {@link BusinessApplicationPlugin} which accepts opaque data and

\* returns structured data.

\*

\* @param businessApplicationName The name of the business application to

\* execute.

\* @param jobParameters The global job parameters to the business application.

\* @param inputDataUrl The URL to the opaque input data.

\* @return The result parameters from the business application.

\*/

public Map<String, Object> execute(

String businessApplicationName,

Map<String, Object> jobParameters, URL inputDataUrl

);

/\*\*

\* Execute a {@link BusinessApplicationPlugin} which accepts and returns

\* opaque data.

\*

\* @param businessApplicationName The name of the business application to

\* execute.

\* @param jobParameters The global job parameters to the business application.

\* @param inputDataUrl The URL to the opaque input data.

\* @param resultDataContentType The content type of the result data.

\* @param resultData The output stream that the plug-in can use to write the

\* result data to.

\*/

public void execute(

String businessApplicationName,

Map<String, Object> jobParameters,

URL inputDataUrl,

String resultDataContentType,

OutputStream resultData

);

### CPF Client

CPF plug-ins deployed to a CPF web server can also be tested using the CPF Java or Javascript clients. See the CPF Client Java User Guide or CPF Client Javascript User Guide for more details.

1. <http://www.springsource.org/spring-core#documentation> [↑](#footnote-ref-1)
2. Change /projects to the directory you use for development [↑](#footnote-ref-2)
3. <http://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html#Dependency_Exclusions> [↑](#footnote-ref-3)
4. <http://en.wikipedia.org/wiki/Java_annotation> [↑](#footnote-ref-4)
5. <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-p-namespace> [↑](#footnote-ref-5)
6. <http://tsusiatsoftware.net/jts/main.html> [↑](#footnote-ref-6)
7. The Well-Known text WKT encoding is described at <http://en.wikipedia.org/wiki/Well-known_text>. The SRID can also be specified using the PostGIS EWKT SIRD=<srid>; prefix, see <http://postgis.org/documentation/manual-1.5/ch04.html#EWKB_EWKT>. [↑](#footnote-ref-7)
8. Plug-ins can use the Geometry and GeometryCollection classes as a return type. However some file formats (ESRI Shapefile) can only support a single specific type of geometry (e.g. Point, LineString, Polygon). Therefore if the plug-in is only going to return a Point geometry, use the Point class as the return type. [↑](#footnote-ref-8)
9. <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-factory-placeholderconfigurer> [↑](#footnote-ref-9)
10. <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-factory-overrideconfigurer> [↑](#footnote-ref-10)
11. [http://maven.apache.org/plug-ins/maven-deploy-plug-in/](http://maven.apache.org/plugins/maven-deploy-plugin/) [↑](#footnote-ref-11)