ORACLE
INSURANCE

ORACLE
INSURANCE

Oracle® Insurance Policy Administration

Extensibility

Version 10.0.0.0

Documentation Part Number: E40981_01

October, 2013

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

License Restrictions

Warranty/Consequential Damages Disclaimer

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

Warranty Disclaimer

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

Restricted Rights Notice

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are “commercial computer software” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Hazardous Applications Notice

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate failsafe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Third Party Content, Products, and Services Disclaimer

This software or hardware and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Table of Contents

INTRODUCTION

Extensibility is a critical factor when selecting an architecture that can be enhanced with the speed of your organizations' needs. The ability to extend the system and hook in new system capability without making major infrastructure changes is necessary for system maintainability and for avoiding early obsolescence. The Oracle Insurance Policy Administration (OIPA) system provides several mechanisms for extensibility. Currently, all extensions are implemented as Java classes that are injected into specific points or levels in the OIPA infrastructure. Extension developers need only implement the requisite Java interfaces in order to access this powerful OIPA feature.

Note:	Creating custom extensions should be viewed as a means to enhance the functionality of OIPA only when there are no means to achieve the same using OIPA's built-in configuration ability and when there are no such planned enhancements announced for near-future versions of the software.

Customer Support

If you have any questions about the installation or use of our products, please visit the My Oracle Support website: https://support.oracle.com, or call (800) 223-1711.

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

---

[1] Coding custom extensions requires a great deal of knowledge about the internal data structures and logical processes used by OIPA. Improperly used, such extensions can adversely affect the operational accuracy and/or efficiency of the system. For that reason, Oracle recommends that, with a few exceptions, (redirecting the Login page start method to accomplish single sign-on, for example), extensions only be used when critical functionality cannot be attained by the use of OIPA's native configuration language and such functionality is not planned for a future version of OIPA that is to be released in a time frame suitable to the customer.

Further, because there is no guarantee that the internal code structures utilized by the extension framework will not change from one release of OIPA to the next, when it is deemed necessary to code an extension, Oracle also recommends that the customer informs Oracle or Oracle's implementation partner about such usage so that the customer may be advised of any internal change that might affect their custom extension code or its usage.

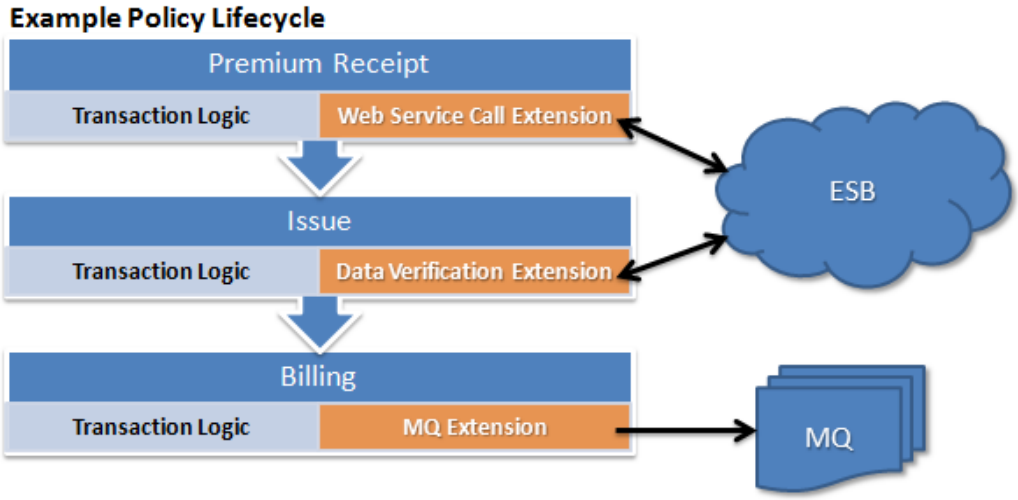
Overview

Primarily, there are three types of extensions: Anonymous, Feature Specific and Named.

1. The **Anonymous** extensions allow for invocation of custom logic using pre-defined methodologies.
2. The **Feature Specific** extensions allow for extending specific OIPA features.
3. The **Named** extensions allow for custom logic at pre-defined points through the lifecycle of specific system events. Named extensions allow for fine-tuned customization.

Anonymous Extensions

Anonymous extensions are provided with data from running transactions, making them powerful tools for integration. Anonymous extensions can execute when an activity is run or when the FileReceived web service is invoked. In the example below, the policy lifecycle includes the OIPA transactions Premium Receipt, Issue and Billing. The first two transactions illustrate how the system can perform messaging over an enterprise service bus (ESB). The last transaction, Billing, illustrates MQ series integration.



Feature Specific Extensions

Feature Specific extensions provide the ability to enhance a specific OIPA feature. There is only one Feature Specific extension at this time called External Client Integration which can be used to enhance OIPA's client and role management feature to interact with an external agent database.

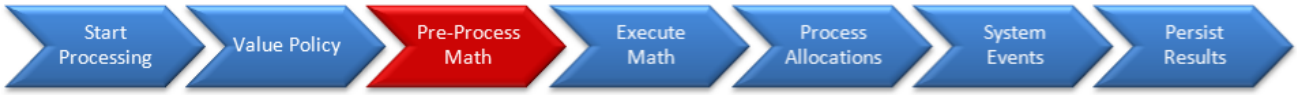
Named Extensions

When fine-tuned control over the application's lifecycle is required, Named extensions can be employed. Named extensions are provided through the Extensibility Framework, which is discussed later in this document.

Below is a simplified rules engine processing example that illustrates how Named extensions can provide pre- or post-processing, or can replace a processing step altogether.

1. Default implementation

2. Pre-processing **added** via an extension



3. Post-processing **added** via an extension

4. Math processing lifecycle step is **replaced** with custom math via an extension

ANONYMOUS EXTENSIONS

There are several facilities in place that enable extensibility using Anonymous extensions. They are in the Math processing of a transaction rule, via the ExternalProcess business rule and through the FileReceived Web Service. Each mechanism is explained in detail below. Anonymous Extensions are not included in a configuration file. Their invocation is governed by configuration of OIPA rules.

Math Type Process

Some tasks may require interaction with custom Java code during the math processing of rules. OIPA has a dedicated math variable type for this purpose, which can be added to any XML math section that contains the `<MathVariable>` element. The math variable type is `PROCESS`. This has the advantage of allowing for business rule logic to dictate which extension should execute and which parameters should be passed. Refer to the OIPA XML Configuration Guide for further detail about the math variable type.

Example XML Configuration

This example illustrates the `PROCESS` type `<MathVariable>` and the available attributes and parameter passing.

```
<MathVariable NAME="VariableName"
              TYPE="PROCESS"
              NAMESPACE="com.example.package"
              OBJECT="ClassName"
              ISARRAY="YES|NO"
              DATATYPE="DataType*">
   <Parameters>
      <Parameter NAME="InputVar" TYPE="INPUT">Literal|VariableName</Parameter>
      <Parameter NAME="OutputVar" TYPE="OUTPUT">VariableName</Parameter>
   </Parameters>
</MathVariable>
```

Note:	Data type can be one of the following: TEXT, INTEGER, DECIMAL, BOOLEAN, DATE.

Java Implementation Details

The above example requires that a class, `com.example.package.ClassName`, implement the interface `IProcessableObject`. This class must be present on the class path of the application's class loader. The `IProcessableObject` interface is defined as follows:

```
public interface IProcessableObject {

   public Object execute( Map<String, Object> inputParameterMap,
                          Map<String, Object> outputParameterMap )
                          throws Exception;
}
```

The implementation will receive two `java.util.Map` instances. The first, `inputParameterMap`, contains all of the values defined as input parameters in the Math XML, keyed on the parameter name. Similarly, the

outputParamaterMap contains all of the values defined as output parameters in the Math XML, keyed on the parameter name. Output parameters will be copied back to the original variable name specified as the value of the parameter. In other words, the *value* of the output variables will be passed into the extension. Any changes to that value will be applied to the variable that is provided by the parameter definition. In essence, the variable is "passed by reference" to the extension.

By supporting output parameters, the extension can return any number of values. By default, the extension returns a single value, which is stored in the variable given by the VARIABLENAME attribute from the XML Configuration example above.

Example Use of Math Extension

Assume there is a service available in the enterprise that will validate that a postal code provided by the user matches the city also provided by the user. The extension will return a Boolean value indicating whether or not the postal code/city pair matches. It will also return the valid city name for the given postal code in the event that the city and postal code do not match.

Example XML Configuration

```
<MathVariable NAME="PostalCodeMV"
              TYPE="FIELD" DATATYPE="TEXT">Policy:IssuePostalCode</MathVariable>
<MathVariable NAME="CityMV"
              TYPE="FIELD" DATATYPE="TEXT">Policy:IssueCity</MathVariable>
<MathVariable NAME="CorrectCityMV"
              TYPE="VALUE" DATATYPE="TEXT"></MathVariable>
<MathVariable NAME="PostalCodeAndCityMatch"
              TYPE="PROCESS"
              NAMESPACE="com.example.package"
              OBJECT="VerifyPostalCode"
              ISARRAY="NO"
              DATATYPE="BOOLEAN">
   <Parameters>
      <Parameter NAME="PostalCode" TYPE="INPUT">PostalCodeMV</Parameter>
      <Parameter NAME="City" TYPE="INPUT">CityMV</Parameter>
      <Parameter NAME="CorrectedCity" TYPE="OUTPUT">CorrectCityMV</Parameter>
   </Parameters>
</MathVariable>
```

Extension Pseudo-Code

```
class VerifyPostalCode implements IProcessableObject {
   public Object execute( Map <String, Object> inputVariableMap,
                          Map <String, Object> outputVariableMap ) throws Exception {
      String postalCode = ( String )inputVariableMap.get( "PostalCode" );
      String city = ( String )inputVariableMap.get( "City" );
      VerificationResult result = ExternalService.verifyPostalCode( postalCode, city );

      if( result.isValid() ) {
         return true;
      }
      else {
         outputVariableMap.put( "CorrectedCity", result.getCorrectedCity() );
         return false;
      }
   }
}
```

ExternalProcess Business Rule

Every transaction in OIPA can have business rules associated with it. These business rules are responsible for applying logic during the transaction lifecycle. Typical business rules will perform actions at a higher level than the transaction itself, such as generating a suspense account or writing data to fields (i.e. policy, role, client, etc.).

In order to facilitate greater flexibility, OIPA includes a special business rule that is capable of calling custom Java code called `ExternalProcess`.

Implementing `ExternalProcess` requires the following:

1. A business rule named `ExternalProcess` attached to the transaction. The business rule will describe the Java class to be executed.
2. The business rule name must exist in the TransactionBusinessRulePacket business rule in the order it should be executed relative to other attached business rules.
3. An extension class that implements the `IApeExtension` interface.

Example XML Configuration

```
<ExternalProcess>
   <Process>
      <Assembly>com.example.extension</Assembly>
      <Object>ExternalProcessImpl</Object>
   </Process>
   <Parameters>
      <Parameter NAME="ParameterName">ParameterValue</Parameter>
   </Parameters>
</ExternalProcess>
```

Java Implementation Details

The previous example requires that a class, `com.example.extension.ExternalProcessImpl`, implement the interface `IApeExtension`. This class must be present on the class path of the application's class loader. The `IApeExtension` interface is defined as follows:

```
public interface IApeExtension {
    public void process( IActivityBll activityBll, Map <String, String> parameterCollection );
    public void processUndo( IActivityBll activityBll, Map <String, String>
    parameterCollection );
}
```

The `process(..)` method is executed during forward processing. The `processUndo(..)` method is executed during reversal.

The `activityBll` parameter exposes the necessary surface area for extension. It's critical to understand that activities are executed as a single unit of work. That is, if any operation fails, then no changes will have been made to the system. To enable this, insert, change and delete operations are exposed by `IActivityBll`. All data changes should be made through these mechanisms, as they ensure proper transactional integrity.

Note:	Direct modification of the database can lead to undesirable or inaccurate results.

FileReceived Web Service

The FileReceived Web Service, sometimes referred to as AsFile, exposes extensions before and after the insert operation occurs. The FileReceived lifecycle is illustrated below, along with the associated insert extension opportunities.

The basic lifecycle for the FileReceived Web Service is:

1. The FileReceived Web Service receives a request via a SOAP message.
2. The AsFile entry is looked up using the FileID specified in the request.
3. The math in the AsFile entry's XMLData is processed.
4. The attributes in the AssignAttributes section of the AsFile entry's XMLData are processed.
5. The XSLT maps the request XML to AsXml.
6. The transformed AsXml is mapped to data objects.
7. PreInsert operations are performed on the objects.
8. Objects are inserted into the database.
9. PostInsert operations are performed on the objects.
10. Output XSLT is loaded from AsFileOutput based on the attributes in AssignAttributes.
11. Response XML is built.
12. If the ValidationError section is configured in the XSLT, then a SOAP fault is created (with embedded response XML) and sent to the caller. Otherwise, response AsXml is returned.

Example XML Configuration

```
<File>
  <RequestType>...</RequestType>
  <Math ID="MathVariablePrefix">
    <MathVariables>
      <MathVariable>...</MathVariable>
       ...
    </MathVariables>
  </Math>
  <AssignAttributes>...</AssignAttributes>
  <PreInsert>
    <Object CLASS="com.example.extension.PreInsertExtension">
      <Parameters>
        <Parameter NAME="Name">Value</Parameter>
        <Parameter NAME="Name">Value</Parameter>
          ...
      </Parameters>
    </Object>
    <Object CLASS="com.example.extension.PreInsertExtension2" />
  </PreInsert>
  <PostInsert>
    <Object CLASS="com.example.extension.PostInsertExtension" />
    <Object CLASS="com.example.extension.PostInsertExtension2" />
      ...
```

```
    </PostInsert>
  </File>
```

Java Implementation Details

The `PreInsert` and `PostInsert` extensions must implement the `IFilePreInsertProcessorBll` and `IFilePostInsertProcessorBll`, respectively. If the `Parameters` element is present, the specified parameters will be passed to the extension. The text of the `Parameter` elements should contain either a constant or the name of an attribute from the `AssignAttributes` section. This allows for the passage of data to the defined extensions.

The `PreInsert` interface is defined as:

```
public interface IFilePreInsertProcessorBll {
	public <T extends AdminServerPersistentDcl> ArrayList <T> process( ArrayList <T> dclList,
	String requestXml, Map <String, String> parameterMap ) throws AsExceptionUtl;
}
```

The `PostInsert` interface is defined as:

```
public interface IFilePostInsertProcessorBll {
	public <T extends AdminServerPersistentDcl> String process( ArrayList <T> dclList, String
	requestXml, Map <String, String> parameterMap ) throws AsExceptionUtl;
}
```

FEATURE SPECIFIC EXTENSIONS

External Client Integration

OIPA may be integrated with external systems that can provide client information. The external client integration feature provides support to:

- Display external client data on the OIPA Role screen.
- Attach external clients to OIPA policies.

Rules Configuration

This section describes how to configure business rules to implement the OIPA external client functionality.

1. Make sure that External Client Plan security is enabled (checked) in the Rules Palette Plan Page Security.
2. Using the Rules Palette Admin Explorer, add codes for new external roles to the AsCodeRole code set in the AsCode table—navigate to **Administration | Code Names**. If using the OIPA client search option, also add codes for external client types to the AsCodeClientType code set. The table below shows sample values for both codes:

Code Name	Code Value	Short Description	Long Description	System Indicator
AsCodeRole	71	External Custom Agent	External Custom Agent	Unchecked
AsCodeRole	72	External Agent	External Agent	Unchecked
AsCodeClientType	06	External Client	External Client	Unchecked

3. If needed, insert authorizations for the ExternalClient page to be able to see the ExternalClientDetail popup page that is used to view the details of the external client.
4. For the ExternalClientDetailsScreen rule, create a company-level override to configure which external client role codes and fields are associated with each defined external role. The information in the rule is used:
   - To validate a set of keys identifying a client in the external system; the keys are returned when a new external role is created.
   - To configure the External Client Detail popup page, which is used to view and, in some cases, to edit external client details on the Policy or Segment role page.

   The external client fields configured by the rule may be classified as following:
   - Key fields that identify a client in an external system. They are retrieved from an external system when a new external role is created, and stored in the AsExternalClient/AsExternalClientField tables in the OIPA database. Later, they are used as keys in requests to an external system whenever OIPA needs to obtain external client data. The key fields are indicated by a “No” value in

the ExternalSource tag, which has its KEY attribute set to “Yes”. When a new external role is created, OIPA validates returned key values for a selected external client against this rule to make sure keys are not empty.

- OIPA-specific fields that are configured and exist in the OIPA application only. They are stored in the OIPA database. Values for these types of fields can be entered and modified on the External Client Detail popup page. The OIPA-specific external client fields are indicated by a “No” value in the ExternalSource tag, which has its KEY attribute set to “No” (default value) in the ExternalClientDetailsScreen rule.
- External fields that are always returned from an external system and never stored in the OIPA database. They are indicated by a “Yes” value in the ExternalSource tag in the ExternalClientDetailsScreen rule. Fields of this type can be displayed on the External Client Detail popup page, but cannot be modified.

Example:

```
<ExternalClientDetailScreen>
    <Client ROLECODE="71">
        <Fields>
            <Field>
                <Name>CLID</Name>
                <Display>External ID</Display>
                <DataType>Text</DataType>
                <ExternalSource KEY="Yes">No</ExternalSource>
            </Field>
            <Field>
                <Name>Address</Name>
                <Display>Address</Display>
                <DataType>Text</DataType>
                <ExternalSource>Yes</ExternalSource>
            </Field>
            <Field>
                <Name>PaymentMethod</Name>
                <Display>Methods</Display>
                <DataType>Combo</DataType>
                <Query TYPE="SQL">SELECT CodeValue,LongDescription FROM AsCode … </Query>
                <DefaultValue>01</DefaultValue>
                <ExternalSource>No</ExternalSource>
            </Field>
        </Fields>
    </Client>
    . . .
</ExternalClientDetailScreen>
```

5. For the PolicyScreen rule or Segment rule (for segment roles):

   a. Create a plan-level override.
   b. Add external client roles to the configuration.
   c. Identify the name of a class that implements the IExternalClientRowRetriever interface that must be called to retrieve a row of external client data from an external system.

```
<Role NAME="External Agent" EXTERNAL="Yes">
    <ExternalClientRowRetriever>com.adminserver.externalClient.ExternalClientRows
    </ExternalClientRowRetriever>
    <RoleCode>72</RoleCode>
    <RoleCount>2</RoleCount>
    <RolePercent>100</RolePercent>
    <ClientType>06</ClientType>
</Role>
```

6. For an implementation that uses the OIPA client search screen, the ClientSearchScreen rule needs to include configuration of the client search screen for the defined external client types. The name of a class that implements the IExternalClientSearch interface should be specified using the ExternalClientSearchRetriever tag.

   Example:

```
<Client TYPECODE="06" EXTERNAL="Yes">
    <ExternalClientSearchRetriever>com.adminserver.externalClient.ExternalClientSearch
    </ExternalClientSearchRetriever>
    <Search>
        <Fields>
            <Field>
                <Name>LastName</Name>
                <Display>Last Name</Display>
                <DataType>Text</DataType>
            </Field>
            <Field>
                <Name>FirstName</Name>
                <Display>First Name</Display>
                <DataType>Text</DataType>
            </Field>
        </Fields>
    </Search>
    <Results DEFAULTRESULTS="Yes">
    . . .

</Client>
```

7. If the Custom screen option is implemented, the role configuration in the Policy or SegmentName rule must include the CustomScreen tag with the name of a class that implements the IExternalClientSearchScreen interface. See the bolded information in the configuration below.

```
<Role NAME="External Custom Agent" EXTERNAL="Yes">
    <ExternalClientRowRetriever>com.adminserver.externalClient.ExternalClientRows
    </ExternalClientRowRetriever>
    <CustomScreen>com.adminserver.externalClient.CustomScreen</CustomScreen>
    <RoleCode>71</RoleCode>
    <RoleCount>2</RoleCount>
    <RolePercent>100</RolePercent>
</Role>
```

Implementing Java Interfaces

In addition to the business rule configuration, integration with an external client system requires implementation of several OIPA interfaces by a Java developer. The external classes that implement the interfaces described in this section should be deployed in a shared library, as with other extension classes. Please refer to the OIPA Deployment Installation instructions for your particular application server for information on how to configure shared libraries.

Retrieving a Row of External Client Data

The `IExternalClientRowRetriever` interface should be implemented to retrieve client data from an external system. The returned data are then displayed as a row in the role summary table on the OIPA Role screen. The

same interface is also used to retrieve external client information displayed on the External Client Detail screen. The name of the implementation class (or the name of the Spring bean if the implementation class is registered as a Spring component) should be specified for each external role in the business rules as described in the **Rules Configuration** section above.

```
/**
 * Interface to retrieve a row of external client data. It is used to retrieve external
 * client data displayed in the role summary table Also, used to display external data
 * on the edit external client details popup screen
 * (for the fields marked with ExternalSource tag set to Yes)
 */
public interface IExternalClientRowRetriever {

   /**
    * Retrieves a row of external client data. Besides returning values to match
    * the columnDcl list, the implementation may return value a named "ClientName"
    * that will be used to display name of the client
    *
    * @param keyVariableDclList
    *           list of keys that identify external client
    * @param columnDclList
    *           list of columnDcl objects that defines returned values
    * @param locale
    *           current user locale
    * @return
    */
   public List <VariableDcl> retrieveRow( List <VariableDcl> keyVariableDclList,
                     List <ColumnDcl> columnDclList, Locale locale )
                     throws ExternalClientException;
}
```

The `retrieveRow` method of the `IExternalClientRowRetriever` interface has the following parameters:

- A list of `VariableDcl` objects containing data to identify a client in an external system. This contains keys that have been retrieved from an external system when a client has been selected. The key data are stored in the OIPA database in the AsExternalClient/AsExternalClientField tables and used when a client needs to be identified in the external system. A `VariableDcl` object is a POJO (Plain Old Java Object) bean with the following properties:

  ```
  String variableName;
  String dataType;
  Object value;
  String currencyCode;
  ```

  The values of the `dataType` property are listed below:

  ```
  "01", //Date
  "02", //Text
  "03", //Integer
  "04", //Float
  ```

  The object type of the `value` property should match the type of the `VariableDcl` instance. The `currencyCode` property should be set for the currency values.

- A list of `ColumnDcl` objects describing what values should be returned from the method. The list consists of columns configured in the RoleScreen business rule (for the role view), as described in the **Rules Configuration** section above, that have the Group attribute set to either CLIENT or CLIENTFIELD. A `ColumnDcl` object is a POJO bean that has the following properties, which areimportant in the context of the `retrieveRow` method:

  ```
  private String             dataType;
  private String             name;
  ```

  The `dataType` property may have the following values:

  ```
  public static final String DATATYPE_Date    = "Date";
  public static final String DATATYPE_Money   = "Money";
  public static final String DATATYPE_Integer = "Integer";
  public static final String DATATYPE_Decimal = "Decimal";
  public static final String DATATYPE_Percent = "Percent";
  public static final String DATATYPE_Text    = "Text";
  ```

  The locale of the current user can be used to localize returned values if required.

The `retrieveRow` method retrieves a list of `VariableDcl` objects that represents search results in an external client system based on the key values passed in the first parameter. The elements in the list should match the elements requested in the second parameter of the method. Additionally, the method may add a `VariableDcl` object with the name “ClientName” to the list. This value is used by OIPA to display the client name in the left side navigation bar on the role screen.

In case of failure, the method may throw an instance of the `ExternalClientException` exception class, derived from the `Exception` class, that has the following constructors:

```
public class ExternalClientException extends Exception {

    public ExternalClientException( String message ){. . .

public ExternalClientException( Throwable cause ) {. . .

    public ExternalClientException( String message, Throwable cause ) {. . .
```

Here is a simple example of the interface implementation:

```
@Component
@Scope("singleton")
public class ExternalClientRows implements IExternalClientRowRetriever {

   public List <VariableDcl> retrieveRow( List <VariableDcl> keyVariableDclList,
List <ColumnDcl> columnDclList, Locale locale )
throws ExternalClientException {

      if( keyVariableDclList == null || keyVariableDclList.size() != 1 ) {
         throw new ExternalClientException( "Missing or redundant keys" );
      }
      VariableDcl keyVarDcl = keyVariableDclList.get( 0 );
      int key = TypeHelperUtl.getInt( keyVarDcl.getValue() );
      ExClientDcl exClientDcl = ExClientDal.findClients(key);
      if( exClientDcl == null ) {
         throw new ExternalClientException( "Could not find an external client:" + key );
      }
      List <VariableDcl> results = new ArrayList <VariableDcl>();
      for( ColumnDcl columnDcl : columnDclList ) {
         String value = "";
         if( "Name".equalsIgnoreCase( columnDcl.getName() ) ) {
            value = exClientDcl.getLastName() + ", " + exClientDcl.getFirstName();
         }
         else if( "LastName".equalsIgnoreCase( columnDcl.getName() ) ) {
            value = exClientDcl.getLastName();
         }
         else if( "FirstName".equalsIgnoreCase( columnDcl.getName() ) ) {
            value = exClientDcl.getFirstName();
         }
         else if( "Address".equalsIgnoreCase( columnDcl.getName() ) ) {
            value = exClientDcl.getAddress();
         }
         else if( "TaxId".equalsIgnoreCase( columnDcl.getName() ) ) {
            value = exClientDcl.getTaxId();
         }
         VariableDcl varDcl = new VariableDcl( columnDcl.getName(), value,
FieldDataType.TEXT.getTypeCode() );
         results.add( varDcl );
      }
      VariableDcl varDcl = new VariableDcl( "ClientName",
            exClientDcl.getLastName() + ", " + exClientDcl.getFirstName(),
            FieldDataType.TEXT.getTypeCode() );
      results.add( varDcl );
      return results;
   }
}
```

Retrieving External Client Search Results

The `IExternalClientSearch` interface should be implemented for external roles that are configured to use the OIPA client search tab on the Role screen, which is used to attach external clients to OIPA policies. This interface provides a method to query an external system using a set of search criteria. The search criteria, and the columns of search results returned, are configured similarly to the search configuration for clients stored in the OIPA system. The name of the implementation class (or the name of the Spring bean if the implementation class is registered as a Spring component) should be specified for each external role in the business rules as described in the **Rules Configuration** section above.

```
/**
 * Interface to retrieve results of an external client search. This interface is
 * implemented when using OIPA client search screen to retrieve results of
 * the external client search
 */
public interface IExternalClientSearch {

   /**
    * Returns client rows found by an external search for the given search criteria
    *
    * @param criteriaFieldDataDclList
    *           list of criteria for external client search collected on
    *           the OIPA client search screen
    * @param columnDclList
    *           list of columnDcl objects that defines returned values
    * @param pageNumber
    *           page number of the search results
    * @param pageSize
    *           size of the page of the search results
    * @param locale
    *           current user locale
    * @return
    */
   public RowDataListDcl <ExternalClientSearchRowDcl> retrieveClientSearchResults(
            List <IFieldDataDcl> criteriaFieldDataDclList,
            List <ColumnDcl> columnDclList,
            Integer pageNumber,
            Integer pageSize,
            Locale locale
         ) throws ExternalClientException;
}
```

The `retrieveClientSearchResults` method of the `IExternalClientSearch` interface has the following parameters:

- A list of search criteria represented by objects that implement the `IFieldDataDcl` interface. The following methods of the interface are the most important in the context of the external client search:

```
public interface IFieldDataDcl {

   /**
    * Gets the value of the field
    */
   public Object getValue();

   /**
    * Gets the fieldName value
    */
   public String getFieldName();
```

```
    /**
     * Returns the value associated with the column: DateValue
     */
    public Date getDateValue();

    /**
     * Returns the value associated with the column: FieldTypeCode
     */
    public String getFieldTypeCode();

    /**
     * Returns the value associated with the column: FloatValue
     */
    public Double getFloatValue();

    /**
     * Returns the value associated with the column: IntValue
     */
    public Integer getIntValue();

    /**
     * Returns the value associated with the column: TextValue
     */
    public String getTextValue();

    /**
     * Returns the currencyCode
     */
    public String getCurrencyCode();
     . . .
}
```

The values of the `fieldTypeCode` property are the same as the type values for the `VariableDcl` objects:

```
"01", //Date
"02", //Text
"03", //Integer
"04", //Float
```

The developer must extract search criteria name/value pairs from the list, and pass them on to an external system search call.

- A list of `ColumnDcl` objects describing what values should be returned from the method. The list consists of columns configured in the ClientSearchScreen business rule as described in the **Rules Configuration** section above.

Note:	The next two parameters can be used only if an external system supports paginated search. Otherwise, all external rows found by the search may be returned. This could affect the external search performance.

- Page number of the search results.
- The size of the search result page.
- The locale of the current user, which could be used to localize the returned values if desired.

The `retrieveClientSearchResults` method retrieves a list of `ExternalClientSearchRowDcl` objects. The list contains the search results from the external client system that satisfy the search criteria specified. The elements in the list should match the elements requested in the second parameter of the method. The returned list is of type `RowDataListDcl.` Besides a list of objects, this list also contains the total number of rows of search results. The total number of rows may be different from the number of rows returned if the external system supports pagination of search results.

```
public class RowDataListDcl <T> extends ArrayList <T> {

        public int getTotalRowCount()
        . . .
```

The `ExternalClientSearchRowDcl` objects returned in the list represent individual rows of search results. The `ExternalClientSearchRowDcl` class has the following properties that are important in the context of an external client search:

```
public class ExternalClientSearchRowDcl extends RowDcl {

   /**
    * List of external client keys that identify client in an external system
    */
   List <VariableDcl> externalKeyVariableDclList;
   /**
    * Client name
    */
   String             clientName;
    . . .

public class RowDcl {

   List <CellDcl>      rowData;
    . . .
```

The `externalKeyVariableDclList` list should be populated with the keys that identify rows of data in an external system. When an external client is chosen by a user to map to an external role, the key data will be stored in the OIPA database. Later, it may be used to retrieve the external client information. The `clientName` property may be set so that it can be used in OIPA to display a selected client name. The `rowData` property is used to display client data in the search result table. Each `CellDcl` object represents a value that will be displayed in a single cell in the search result table. The type of a `CellDcl` object should match the type of a corresponding `ColumnDcl` object in the list passed in as the second parameter of the method. This is shown in the following table:

ColumnDcl and CellDcl types

ColumnDcl Type	CellDcl Type/Constructor
Money	MoneyCellDcl(Double data, CurrencyDcl currencyDcl)
Date	DateCellDcl(Date data)
Integer	IntegerCellDcl(Integer data)
Percent	PercentCellDcl(Double data)
Decimal	DecimalCellDcl(Double data)
Text	TextCellDcl(String data)

For money values, an instance of the `CurrencyDcl` class may be created using the following constructor:

```
public CurrencyDcl( String currencyCode,
    String currencyName,
    Integer displayRoundPlaces,
    Integer currencyRoundPlaces,
    String currencyRoundMethod )
```

In case of a failure, the method may throw an instance of the `ExternalClientException` exception class.

Here is a simple example of the interface implementation:

```
public class ExternalClientSearch implements IExternalClientSearch {

   /*
    * An exmaple implementation that always returns a single client
    * regardless of search criteria. It also ignores a list of
    * columns
    **/
   public RowDataListDcl <ExternalClientSearchRowDcl> retrieveClientSearchResults(
            List <IFieldDataDcl> criteriaFieldDataDclList,
            List <ColumnDcl> columnDclList,
            Integer pageNumber, Integer pageSize, Locale locale
         ) throws ExternalClientException {

      RowDataListDcl <ExternalClientSearchRowDcl> results =
         new RowDataListDcl <ExternalClientSearchRowDcl>();
      ExternalClientSearchRowDcl rowDcl = new ExternalClientSearchRowDcl();
      rowDcl.getRowData().add( new TextCellDcl( "LastName, FirstName" ) );
      rowDcl.getRowData().add( new TextCellDcl( "123-23-8970") );
      rowDcl.getRowData().add( new TextCellDcl( "100 Street Town, PA 19003" ) );
      rowDcl.setClientName( "LastName, FirstName" );
      List <VariableDcl> varDclList = new ArrayList <VariableDcl>();
      VariableDcl varDcl = new VariableDcl( "CLID", "12345",
            FieldDataType.TEXT.getTypeCode(), null );
      varDclList.add( varDcl );
      rowDcl.setExternalKeyVariableDclList( varDclList );
      results.add( rowDcl );
      results.setTotalRowCount( 5 );
      return results;
   }
}
```

Implementing Custom External Client Search Page

The `IExternalClientSearchScreen` interface should be implemented for external roles that are configured to use a custom Client Search screen included on the OIPA Role screen. This interface provides methods for OIPA to find an external page Java Server Faces (JSF) include, retrieve an external client selection made on the custom screen, and map it to an external role. OIPA does not make any assumptions regarding the custom search screen, but simply includes it within the OIPA Role screen. Then it requests the selection from the custom screen through the interface. The name of the implementation class (or the name of the Spring bean if the implementation class is registered as a Spring component) should be specified for each external role in the business rules as described in the **Rules Configuration** section above.

```
/**
 * Interface is required to implement custom client search screen. This is a stateful
 * interface. An instance of an object that implements this interface is created
 * and kept in the role page bean (request scope) maintaining the state
 * of the current search for a user
 */
public interface IExternalClientSearchScreen {

   /**
    * Returns URL to an external JSF include. The include file should be deployed
    * on a web server and accessible through HTTP. Please remember to change
    * the include file extension to txt
    *
    * @return
    */
   public String getScreenUrl();

   /**
    * Retrieves the current selection of an external client search screen.
    * The state of the search is maintained by storing an instance of
    * this object in the role page bean
    *
    * @return
    */
   public ExternalClientSearchRowDcl retrieveSelection() throws ExternalClientException;
}
```

The `getScreenUrl` method of the interface returns a URL to a JSF subpage that will be included in the tab normally occupied by the client search on the OIPA Role screen. The custom JSF include should be deployed as a web application and available to the OIPA system through HTTP protocol.

Note:	Please make sure that the file extension of the resource file is changed from JSPX to TXT. In this case, an include resource is retrieved through HTTP “as is” and included in the Role screen before the entire page is processed by the JSF engine to render the content. If the extension of the include resource remains JSPX, the resource file will be processed by the JSF engine prior to returning it through HTTP. The generated content (HTML) then will be included in the Role screen instead of JSF code.

The `retrieveSelection` method of the `IExternalClientSearchScreen` interface should return the selection that the user has made on the custom screen. An instance of the implementation class is kept as a property of the IceFaces bean, with the name “rolePage,” which is a request scope bean. This conveniently preserves the

state of the instance that can access and retrieve a custom selection. The result is returned as an object of type `ExternalClientSearchRowDcl` (described above for the `IExternalClientSearch` interface).

In case of a failure, the `retrieveSelection` method should throw an instance of the `ExternalClientException` exception class.

Here is a simple example of the interface implementation:

```
public class CustomScreen implements IExternalClientSearchScreen {

   public CustomScreen() {
      super();
   }

   public String getScreenUrl() {
      return "http://localhost:8080/includes/ExternalClientSearch.txt";
   }

   public ExternalClientSearchRowDcl retrieveSelection() throws ExternalClientException {

      // The selection result is hardcoded in this implementation.
      // Normally, it should be retrieved from the current selection
      // of the table that displays external search results.
      ExternalClientSearchRowDcl rowDcl = new ExternalClientSearchRowDcl();
      rowDcl.getRowData().add( new TextCellDcl( "LastName, FirstName" ) );
      rowDcl.getRowData().add( new TextCellDcl( "123-30-9876" ) );
      rowDcl.getRowData().add( new TextCellDcl( "100 Street Town, PA 19003" ) );
      rowDcl.setClientName( "LastName, FirstName" );
      List <VariableDcl> varDclList = new ArrayList <VariableDcl>();
      VariableDcl varDcl = new VariableDcl( "CLID", "12345",
            FieldDataType.TEXT.getTypeCode(), null );
      varDclList.add( varDcl );
      rowDcl.setExternalKeyVariableDclList( varDclList );
      return rowDcl;
   }
}
```

Developing a Custom Page

The OIPA role page, both for policy and segment roles, may be configured to include a custom client search page allowing OIPA customers to customize external client search to satisfy business requirements that can't be satisfied with the standard configurable OIPA client search. It has been discussed in the previous sections how to configure business rules and create a Java implementation of the `IExternalClientSearchScreen` interface. This section will discuss how to create an actual Java Server Face (JSF) page that will be included on the Client Search tab of the OIPA Role screen.

Typically, a custom search should display a set of search criteria and a table with search results that allows the selection of a single row of results before the user clicks the "Add" button to create a role mapped to the selected external client. Please note, the "Add" buttons belongs to the OIPA role page and not the custom include subpage. The selection is returned to OIPA through the `retrieveSelection` method of the `IExternalClientSearchScreen` interface as described in the previous section. The `IExternalClientSearchScreen` interface is stateful, and an instance of the implementation class is kept in the `customScreen` property of the IceFaces `rolePage` bean that has the request scope.

Below is an example of the `IexternalClientSearchScreen` interface implementation.

```
public class CustomScreen implements IExternalClientSearchScreen {

   public CustomScreen() {
      super();
   }

   public String getScreenUrl() {
      return "http://localhost:8080/includes/ExternalClientSearch.txt";
   }

   public ExternalClientSearchRowDcl retrieveSelection() throws ExternalClientException {

      // The selection result is hardcoded in this implementation
      // Normally, it should be retrieved from the current selection
      // of the table that displays external search results
    . . .
      return rowDcl;
   }

   public String getTitleCustom() {
      return "Custom Screen";
   }

   public String getTitleCustomContent() {
      return "External Client Search";
   }
}
```

The custom JSF include may then look like this

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:ice="http://www.icesoft.com/icefaces/component"
        xmlns:ui="http://java.sun.com/jsf/facelets"
        xmlns:as="http://www.adminserver.com/jsf">
   <ui:decorate template="../includes/StandardBoxTemplate.txt">
      <ui:param name="style"
 value="popupBoxContainer"/>
      <ui:define name="standardBoxTitle">
         <ice:outputText value="#{rolePage.customScreen.titleCustom}"
          styleClass="boxTitle"/>
      </ui:define>
      <ui:define name="standardBoxContent">
         <ice:outputText value="#{rolePage.customScreen.titleCustomContent}"/>
      </ui:define>
   </ui:decorate>
</f:view>
```

The simplest way to implement a more complex custom include page is to add properties needed in the JSF subpage to the `CustomScreen` class. For example, an implementation could have a list that contains the search results, along with a property that is bound to the row selector of the table that displays search results on the custom page. Then the `retrieveSelection` method would need to return the selected row based on these properties.

It is important to remember that a custom JSF include should be deployed in a web server as a text resource, not as a JSF file. This will allow retrieving the resource through HTTP protocol and including it in the JSF Role page before generating the Role screen content.

NAMED EXTENSIONS[1]

The Extensibility Framework provides a mechanism by which system event lifecycles can be extended. The various lifecycle steps are given specific names in the system, so they have to be accessed using those names. Custom Java code can be added before or after a lifecycle step, or it can replace the lifecycle step altogether. The Extensibility Framework is present at the transaction processing, Web Service and user interface levels.

The Extensibility Framework maps custom Java classes to named points in the system. These names vary based on the context, and are described in detail later in this document. A single Java class can be mapped to multiple extension points and, conversely, a single extension point can have multiple Java classes mapped to it.

The Extensibility Framework allows for wildcards in the specification of the extension point name. For instance, `Activity.*` will call a custom Java class for every extension point starting with `Activity`. If "`*`" is used by itself, then all extension points will be processed by the Java class.

Wildcard Options

Specification	Example	Explanation
<Qualifier>.<Name>	`Activity.StartProcessing`	Only the StartProcessing event will be intercepted.
<Qualifier>.*	`Activity.*`	All extension points starting with Activity will be intercepted.
*	*	All extension points will be intercepted.

Lifecycle flow control is managed through the two methods exposed by every extension point interface, `processPre(..)` and `processPost(..)`. The `processPre(..)` method is executed prior to the lifecycle event. When this method returns true, it indicates that the lifecycle step should execute and when it returns false it indicates that that the lifecycle step and `processPost(..)` should be skipped. This allows for lifecycle steps to be overridden by custom code. The `processPost(..)` method is called after the lifecycle step has completed. If that lifecycle step generated any data, it will be present in the context map as "`Result`".

It's important to note that extensions are not thread-safe. That means, for performance, a single instance is created and continually executed. The use of member variables in an extension is discouraged unless proper locking is in place.

All extension points contain context data in the form of a `java.util.Map`. The contents of this map will vary depending on the extension point. This map is shared between the `processPre(..)` and `processPost(..)`

[1] Coding custom named extensions requires a great deal of knowledge about the internal data structures and logical processes used by OIPA. Improperly used, such extensions can adversely affect the operational accuracy and/or efficiency of the system. For that reason, Oracle recommends that, with a few exceptions, (redirecting the Login page start method to accomplish single sign-on, for example), named extensions only be used when critical functionality cannot be attained by the use of OIPA's native configuration language and such functionality is not planned for a future version of OIPA that is to be released in a time frame suitable to the customer.
Further, because there is no guarantee that the internal code structures utilized by the extension framework will not change from one release of OIPA to the next, when it is deemed necessary to code a named extension, Oracle also recommends that the customer informs Oracle or Oracle's implementation partner about such usage so that the customer may be advised of any internal change that might affect their custom extension code or its usage.

methods, but it is not shared with other extension points. Inter-extension communication can be achieved through the use of some context data mechanism, depending on the implementation. This could be in the form of a `ThreadLocal` variable or an HTTP/Servlet request context.

Each extension point can have one or more extensions registered to it. These extensions are executed in the order in which they appear in the XML configuration.

Example XML Configuration

```
<extensions>
    <extensionPoint type="com.example.extension.ExtensionPointClassName"
    extensionPointName="ExtensionPoint.Name">
        <register extension="com.example.extension.ExtensionClassName1" />
        <register extension="com.example.extension.ExtensionClassName2" />
    </extensionPoint>
</extensions>
```

Shared Rules Engine

The Shared Rules Engine (SRE) can be extended through the use of the extensibility framework. The SRE powers activity processing, and can be extended at many critical lifecycle points.

Example XML Configuration

```
<extensionPoint type="com.adminserver.sre.extensibility.SreExtensionPoint"
    extensionPointName="Activity.*">
    <register extension="com.example.SreExtension" />
</extensionPoint>
```

All extensions written for the `SreExtensionPoint` must implement `ISreExtension`.

```
    public interface ISreExtension {
       public boolean processPre( SreExtensionContext extensionContext );
       public void processPost( SreExtensionContext extensionContext );

}
```

SRE Extension Points

Forward Processing

Lifecycle Step	Extension Point	Available Variables
ActivityBll.Process	Activity.InitializeProcessing	ClientNumber, ActivityProcessType
ActivityProcessorBll.Process	Activity.StartProcessing	InputVariableMap, ApplicationCallback, ActivityProcessType
DoSuspense	Activity.StartSuspense	Activity, Transaction
ProcessSuspense	Activity.ProcessSuspense	Activity
ProcessMultiSuspense	Activity.ProcessMultiSuspense	Activity
DoValuation	Activity.StartValuation	Activity, Transaction
PolicyValuation.Value	Activity.ValuePolicy	ValuationInformation, Activity
DoMath	Activity.ProcessMath	Activity
DoBusinessLogic	Activity.StartBusinessLogic	Activity
ProcessRule	Activity.ProcessBusinessRule	BusinessRule, RuleOption, Activity
DoAssignment	Activity.StartAssignment	Activity, Transaction
ProcessAssignments	Activity.ProcessAssignments	AssignmentList, Activity, ExpressionValidator
ProcessAssignment	Activity.ProcessAssignment	Assignment, Activity
DoDisbursement	Activity.StartDisbursement	Activity, Transaction
ProcessDisbursements	Activity.ProcessDisbursements	DisbursementDetails, Activity
ProcessBalanced	Activity.ProcessBalancedDisbursements	DisbursementDetails, Activity, DisbursementData
ProcessUnbalanced	Activity.ProcessUnBalancedDisbursements	DisbursementDetails, Activity, DisbursementData
DoAccounting	Activity.StartAccounting	Activity
ProcessAccounting	Activity.ProcessAccounting	Activity
DoSpawn	Activity.ProcessSpawn	Activity, Transaction
PrepareValuationChanges	Activity.CompleteValuation	Activity,
PrepareActivityChanges	Activity.ProcessActivityChanges	Activity
DoWrite	Activity.Persist	ActivityProcessResult
DoWriteOnSystemError	Activity.SystemError	Exception

Reverse Processing

Lifecycle Step	Extension Point	Available Variables
ActivityBll.Process	Activity.InitializeProcessing	ClientNumber, ActivityProcessType
UndoProcessor.Process	*Activity.StartUndoProcessing*	*InputVariableMap, ApplicationCallback*
DoBusinessLogicForNuvPending	*Activity.StartNuvPendingBusinessLogic*	*Activity, Transaction*
Rule.ProcessNuvPending	*Activity.ProcessBusinessRuleNuvPending*	*BusinessRule, Activity*
ProcessValuationForUndo	*Activity.ProcessValuationForUndo*	*Activity, UndoActivityStatusCode*
DoAccounting	Activity.StartAccounting	Activity,
ProcessAccounting	Activity.ProcessAccounting	Activity
DoBusinessLogicForUndo	*Activity.StartUndoBusinessLogic*	*Activity, Transaction*
Rule.ProcessUndo	*Activity.ProcessBusinessRuleUndo*	*BusinessRule, Activity*
DoWrite	Activity.Persist	ActivityProcessResult
DoWriteOnSystemError	Activity.SystemError	Exception

Scheduled Valuation

Lifecycle Step	Extension Point	Available Variables
GenerateScheduledValuation	ScheduledValuation.GenerateScheduledValuation	ValuesRequestTask

FileReceived

The FileReceived Web Service can be extended through the use of the extensibility framework by implementing one of two interfaces.

Example XML Configuration

```
<extensionPoint type="com.adminserver.webservice.extensibility.FileReceivedExtensionPoint"
        extensionPointName="FileReceived.*">
        <register extension="com.example.FileReceivedExtension" />
</extensionPoint>
```

All extensions written for the `FileReceivedExtensionPoint` must implement `IFileReceivedExtension`.

```
public interface IFileReceivedExtension {
   public boolean processPre( FileReceivedExtensionContext extensionContext );
   public void processPost( FileReceivedExtensionContext extensionContext );
}
```

File Received Extension Points

Lifecycle Step	Extension Point	Available Variables
FileReceived	FileReceived.StartProcessingFileReceived	FileId, IncomingXml
RetrieveFileProcessDcl	FileReceived.StartRetrievingFileRecord	FileId
FindByFileId	FileReceived.FindRecord	FileProcessData
PopulateFileProcessDcl	FileReceived.CreateDataCarrier	XmlHelperUtility, FileProcessData
ProcessAssignAttributes	FileReceived.ProcessAssignAttributes	FileProcessData
ProcessRequest	FileReceived.StartProcessingRequest	FileProcessData
TransformtoAsXml	FileReceived.TransformToXml	FileProcessData
ValidateAsXml	FileReceived.ValidateXml	AsXmlDocument
MapXmlToObject	FileReceived.Deserialize	FileProcessData
ProcessImportedObject	FileReceived.StartProcessingDataCarriers	FileProcessData, PendingImportedObject, AsXmlDocument
PerformPreInsert	FileReceived.StartPreInsert	FileProcessData, PendingInsertObjects
RetrieveDclList	FileReceived.StartRetrievingDataCarriers	PendingImportedObject
BuildDclListFromAsXml	FileReceived.BuildDataCarriersList	PendingImportedObject
DoPreInsert	FileReceived.PreInsert	FileProcessData, PendingInsertObjects
PerformInsert	FileReceived.InsertData	PendingInsertObjects
PerformPostInsert	FileReceived.StartPostInsert	FileProcessData, PendingInsertObjects, AsXmlDocument
DoPostInsertProcessing	FileReceived.StartPostInsertProcessing	FileProcessData, PendingInsertObjects, AsXmlDocument
DoPostInsert	FileReceived.PostInsert	FileProcessData, PendingInsertObjects
BuildResultString	FileReceived.StartBuildingResult	FileProcessData
LoadOutputXslt	FileReceived.LoadOutputXslt	FileProcessData

Example XML Configuration

```
<extensionPoint type="com.adminserver.webservice.extensibility.WebserviceExtensionPoint"
    extensionPointName=" XsltHelper.XmlTransform">
    <register extension="com.example.WebServiceExtension" />
</extensionPoint>
```

All extensions written for the `WebserviceExtensionPoint` must implement `IWebserviceExtension.`

```
public interface IWebserviceExtension {
    public boolean processPre( WebserviceExtensionContext extensionContext );
    public void processPost( WebserviceExtensionContext extensionContext );
}
```

This extension point allows for the overriding of the default XSLT transformer at two stages during processing of the FileReceived web service.

1. When Input Xml is transformed to AsXml.
2. When an Xslt is applied to the response Xml.

This extension will receive the following parameters in the context map:

SourceDocument	XML Document for transformation.
Parameters	The parameters available for passing to the transformer. These are values of attributes set using AssignAttribute in AsFile

In the `processPost` method, the context map will contain `TransformedXml`, which holds the result of the Xslt transformation.

User Interface

The extension point names for the user interface are convention based. They take the form of `<PageName>.<StartProcessing/Process[Action]>`. An example would be `Policy.ProcessSave`.

Example XML Configuration

```
<extensionPoint type="com.adminserver.pas.uip.extensibility.UipExtensionPoint"
        extensionPointName="*">
        <register extension="com.example.UipExtension" />
</extensionPoint>
```

All extensions written for the `UipExtensionPoint` must implement `IUipExtension`.

```
public interface IUipExtension {
    public boolean processPre( UipExtensionContext extensionContext );
    public void processPost( UipExtensionContext extensionContext );
}
```

The UI extension point names can be discovered by writing a "catch-all" extension point and logging the extension point names. For example the following steps can be followed to discover available named extension points on the client screen.

1. Create a Java class called **OIPAExtension**.java. This class can be given any name.
2. Add the following code to the class.

```
public class OIPAExtension implements IUipExtension {
    public boolean processPre( UipExtensionContext extensionContext ){
        boolean result = true;
        // Use the System.out.println() or java logger to print all extension points using
        extensionContext
        String extensionName =  extensionContext.getExtensionPoint().getName();
        System.out.println(" UI Client Page Extension Name : " + extensionName);
    }

    @SuppressWarnings("boxing")
    public void processPost( UipExtensionContext extensionContext ) {
    }
}
```

3. Generate **OIPAExtension**.jar and deploy it to the application server. In WebSphere, include the absolute path of the **OIPAExtension**.jar in sharedlib. In WebLogic, drop the **OIPAExtension**.jar within WEB-INF/lib of the OIPA war included in the OIPA media pack.
4. Add the following to the extensions xml configuration.

```
<extensionPoint type="com.adminserver.pas.uip.extensibility.UipExtensionPoint"
        extensionPointName="*">
        <register extension="com.example.OIPAExtension" />
</extensionPoint>
```

5. Login to OIPA application.
6. Navigate to the Client Screen.
7. The JVM console output will list all the extension point names.

Follow the same steps for different screens.