

# Section 6: Interfaces

## Table of Contents

<b>OVERVIEW .....</b>	<b>3</b>
<b>INTEGRATION SOLUTION PATTERNS.....</b>	<b>3</b>
<b>Inbound .....</b>	<b>3</b>
Assign Attributes Syntax.....	4
Assign Attributes Types .....	5
Example from Assign Attributes.....	5
Validations on inbound messages .....	6
Pre Processing and Post Processing on AsFile .....	7
<b>Outbound Simple (Push).....</b>	<b>8</b>
Configuring Web Service Outbound Simple (Push) .....	8
WEBSERVICE Attributes.....	9
Web Service Business Rule .....	9
<b>Outbound Complex (Push).....</b>	<b>11</b>
<b>Outbound (Pull) .....</b>	<b>13</b>
<b>Custom Web Service .....</b>	<b>13</b>
<b>ADMINSERVERCYCLE.....</b>	<b>14</b>

## Overview

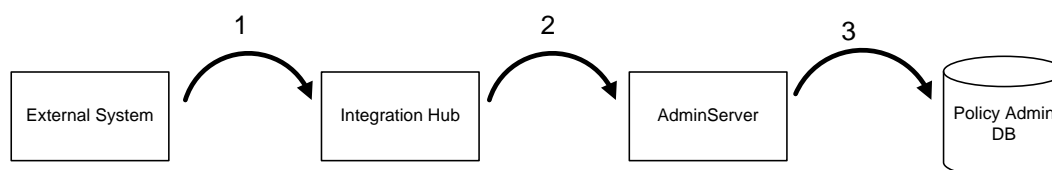
Integration with disparate systems represents one of the key challenges encountered during the deployment of the OIPA System. Enterprise Application Integration, (**EAI**), is the term used to describe the combination of separate applications into a co-operating federation of applications. An EAI solution is required to address the issue of integrating the OIPA System with disparate applications.

EAI solutions generally fall into one of two logical Integration Architectures: Direct point-to-point connections and Middleware-based. Our Integration Strategy employs the use of Web services along with a Middleware-based Integration Architecture to create a Service Oriented Architecture (**SOA**) that greatly simplifies the task of integrating the OIPA System with disparate systems.

As the remainder of this document will detail, using a Middleware-based Integration Architecture provides many benefits over direct point-to-point connections; while at the same time building a foundation for the future exploration of the benefits offered by the emerging discipline of Business Process Management.

## Integration Solution Patterns

### Inbound



1.1 Data delivery to hub

1.2 Web Service to AdminServer (ProcessFileReceived)

1.3 AdminServer delivers data to Policy Administration database

An example of this inbound message would be the addition of an activity to a policy in the admin system. Such a need might arise in the case of integration with a New Business system. When approved a message might be sent to the admin system to create a new copy of a policy.

The Web Service consumed in this event is called ProcessFileReceived. This scenario is sometimes referred to as the AsFile interface. AsFile is our integrated Web Service based import mechanism. Data is delivered through the ProcessFileReceived service. This data is then translated into table inserts through the use of the AssignAttributes rule along with an XSLT (both specific to the inbound message).

Through Rules -> File from the Main Menu screen you can access the "Files" rule screen to add the XSLT and XMLData for a specific ProcessFileReceived message.

File(s)

File Name Format: **ActivateDisbursements**

File Name Format:  File ID:  Version:

**XSLT:**

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <xsl:output indent="no" method="xml"/>
  <xsl:param name="DebugFlag">No</xsl:param>
  <xsl:param name="UpdateDisbursementsActivityGUID"></xsl:param>
  <xsl:param name="UpdateDisbursementsTransactionGUID"></xsl:param>
  <xsl:param name="UpdateDisbursementsPlanGUID"></xsl:param>
  <xsl:param name="UpdateDisbursementsProcessingOrder"></xsl:param>
  <xsl:param name="UpdateDisbursementsEffectiveDate"></xsl:param>
  <xsl:param name="SystemDate"></xsl:param>
  <xsl:param name="DisbursementGUIDMap" as="document-node()"></xsl:param>
  <xsl:param name="ActivityClientNumber"></xsl:param>
  <!-- Main Template -->
  <xsl:template match="ActivateDisbursements">
    <!-- Build Activity Xml For UpdateDisbursements Activity -->
    <xsl:variable name="updateDisbursementsFileSpecificXml">
      <xsl:apply-templates select="." mode="buildFileSpecificXml"/>
    </xsl:variable>
    <!-- Build AsXml -->
    <xsl:element name="AsXml">
      <!-- Build Activity XML -->
      <xsl:call-template name="buildActivity">
        <xsl:with-param name="TransactionGUID" select="UpdateDisbursementsTransactionGUID"></xsl:with-param>
      </xsl:call-template>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

**XMLData:**

```
<File>
  <AssignAttributes>
    <Attribute NAME="DebugFlag" TYPE="VALUE">No</Attribute>
    <Attribute NAME="UpdateDisbursementsActivityGUID" TYPE="GUID"></Attribute>
    <Attribute NAME="UpdateDisbursementsTransactionGUID" TYPE="SQL">SELECT AsTransac</Attribute>
    <Attribute NAME="UpdateDisbursementsPlanGUID" TYPE="SQL">SELECT AsTransac</Attribute>
    <Attribute NAME="UpdateDisbursementsProcessingOrder" TYPE="SQL">SELECT AsT</Attribute>
    <Attribute NAME="UpdateDisbursementsEffectiveDate" TYPE="SYSTEMDATE"></Attribute>
    <Attribute NAME="SystemDate" TYPE="SYSTEMDATE"></Attribute>
    <Attribute NAME="DisbursementGUIDMap" TYPE="SQLMAP">SELECT AsDisburseme</Attribute>
    <!-- Java Specific -->
    <Attribute NAME="ImportMapName" TYPE="VALUE">AsXml</Attribute>
    <Attribute NAME="ClassName" TYPE="VALUE">com.adminserver.utl.JibxListUtl</Attribute>
    <Attribute NAME="ActivityClientNumber" TYPE="VALUE">KRASULJAL</Attribute>
  </AssignAttributes>
  <PostInsert>
    <Object CLASS="com.adminserver.utl.AsFilePostInsertActivityProcessorUtl"></Object>
  </PostInsert>
</File>
```

Check Out New Close

## Assign Attributes Syntax

```
<AssignAttributes>
<Attribute NAME="PlanName" TYPE="XPATH">/Request/Transactions/Transaction/PlanName
</Attribute>
<Attribute NAME="PlanGUID" TYPE="SQL">
  SELECT PlanGuid FROM AsPlan WHERE PlanName = '[PlanName]'
</Attribute>
...
```

## Assign Attributes Types

- NAME - used to reference the value.
- TYPE - defines how the expression will be evaluated. Attributes are evaluated in order from top down, so attributes listed first can be used in expressions below them.
- GUID – sets the attribute to a newly generated GUID.
- VALUE – sets the attribute to the specified value.
- SYSTEMDATE – sets the attribute to the current system date.
- SEQUENCE – sets the attribute by calling `asc_NextSequenceInteger` and passing the NAME as a parameter.
- XPATH – sets the attribute to the result of the specified XPATH expression.
- XPATHSTRINGLIST – sets the attribute to a comma delimited list containing the resulting values of the XPATH.
- XPATHNUMBERLIST - sets the attribute to a comma delimited list containing the resulting values of the XPATH.
- SQL - sets the attribute to the result of the specified SQL statement.
- SQLMAP – sets the attribute to a 'key-value-pair' type collection of the resulting values of the SQL Statement.

## Example from Assign Attributes

- A request's XML contains `<PolicyIssueState>PA</PolicyIssueState>`.
- OIPA stores state values by using state codes found in `AsCode`.
- In order for the XSLT to map `<PolicyIssueState>PA</PolicyIssueState>` to `<IssueStateCode>38</IssueStateCode>`, a SQL map is helpful.
- The SQL map will be evaluated when the `AssignAttributes` XML is processed and then passed to the XSLT.
- The XSLT will then have access to `AsCode.CodeName` where `CodeName = AsCodeState` values to aid in transforming the request XML into `AsXml`.
- The `AsXml` will then be inserted into the database.

```
<Attribute NAME="AsStateCodeMap" TYPE="SQLMAP">
  SELECT ShortDescription, CodeValue
  FROM AsCode WHERE CodeName = 'AsCodeState'
</Attribute>
```

Evaluates to:

```
AL    01
AK    02
AZ    03
AR    04
```

The XSLT can then do something like this...

```
<xsl:variable name="State" select="/Request/PolicyIssueState"/>
<IssueStateCode>
  <xsl:value-of select="$AsStateCodeMap/Map[@KEY=$State]"/>
</IssueStateCode>
```

After being transformed by an XSLT, a client record in AsXml would look like this:

```
<AsXml>
  <AsClient>
    <ClientGuid>B1FC352B-88DB-4DC2-982C-A361E53257F9</ClientGuid>
    <TypeCode>02</TypeCode>
    <CompanyName></CompanyName>
    <LastName>Simpson</LastName>
    <FirstName>Homer</FirstName>
    </MiddleInitial>
    <Sex>M</Sex>
    <DateOfBirth>1962-05-05 00:00:00.000</DateOfBirth>
    <TaxId>444-44-4444</TaxId>
    <UpdatedGmt>2005-12-12 07:36:26.417</UpdatedGmt>
  </AsClient>
</AsXml>
```

Some columns in the database allow null values so the AsXml can omit them if needed. Also, some tags can be included but the XSLT will not always populate them with a value

## Validations on inbound messages

AsFile has the ability to perform validations in the XSLT portion of the configuration.

The following syntax can exist anywhere in the XSLT:

```
<xsl:if test="$variable = 'something is bad'">
  <xsl:variable name="Error1" select="I will throw a SOAP fault"/>
  <ValidationError ERRORSTATUSCODE="Err001">
    <xsl:value-of select="$Error1" />
  </ValidationError>
</xsl:if>
```

If the <xsl:if> expression evaluates to true, the <ValidationError> element will be present in the AsXml resulting from the XSLT.

If the one or more <ValidationError> elements exist in the AsXml, a SOAP fault will be thrown and the text within the element and the ERRORSTATUSCODE will be returned to the caller as part of the SOAP fault.

## Pre Processing and Post Processing on AsFile

AsFile can manipulate data before and/or after it is inserted into the database. This is done by calling specific types of Java classes, which are used for these operations. Pre and Post Insert operations are specified in the XmlData portion of a File's configuration, after the closing of the AssignAttributes element. Pre and Post insert operations are optional and are only used when there is a need to call other functionality in the application.

The following configuration will invoke the AsFile Post Insert Activity Processor after the records are inserted into the database

```
...
</AssignAttributes>
<PostInsert>
  <Object
    CLASS="com.adminserver.utl.AsFilePostInsertActivityProcessorUtl">
  </Object>
</PostInsert>
</File>
```

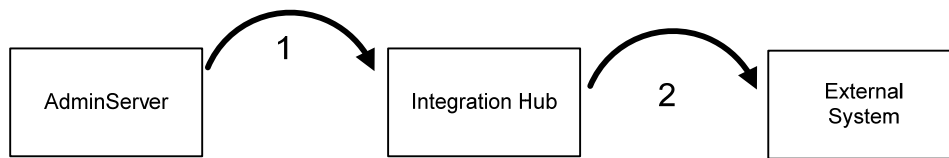
CLASS – specifies the fully qualified name of a Java class to invoke after the records are inserted into the database.

The architecture of the Pre and Post Insert functionality allows these classes to be dynamically instantiated at runtime.

### The flow is as follows

- AsFile receives a web service request.
- The appropriate File is looked up using the FileID specified by the request.
- Assign Attributes XML is processed.
- The XSLT maps the request XML to AsXml.
- If the AsXml contains Validation Errors, AsFile stops and a SOAP fault is returned to the caller.
- AsFile maps the AsXml to business objects.
- PreInsert operations are performed on the objects.
- The objects are inserted into the database.
- PostInsert operations are performed on the objects.
- The AsXml is returned to the caller.

## Outbound Simple (Push)



### 2.1 AdminServer initiating Web Service call

### 2.2 Formatted data delivered to downstream system (XML, Flat File, etc.)

The Outbound Simple scenario involves consuming a web service for the purpose of passing a relatively small (<10KB) message to an end point system. An example of this may be a Commission System, which may need to know of a new Contract – Producer agreement. Alternatively, the Commission System may be required to confirm appropriate licensing of a producer in a given state. Configuration of such an event is illustrated below.

This type of service is generated using a combination of a Math Variable (see *Transaction Configuration*) of TYPE="WEBSERVICE" along with a rule containing Web Service mapping information specific to the service being consumed.

The example below consumes a service named "SampleService" and passes in two variables – "Parameter1" and "Parameter2" to dynamically build the content of the message.

## Configuring Web Service Outbound Simple (Push)

Before declaring the Web Service math variable named "ServiceXML," two regular math variables of TYPE="VALUE" have been illustrated. Any type of math variable can be used here.

```

<MathVariable VARIABLENAME="Variable1" TYPE="VALUE">01</MathVariable>
<MathVariable VARIABLENAME="Variable2" TYPE="VALUE">Some String</MathVariable>
  
```

Next, the TYPE="WEBSERVICE" math variable is declared.

```

<MathVariable VARIABLENAME="ServiceXML" TYPE="WEBSERVICE" RULENAME="WebServices"
SERVICENAME="SomeService">
  <Parameters>
    <Parameter NAME="Parameter1">Variable1</Parameter>
    <Parameter NAME="Parameter2">Variable2</Parameter>
  </Parameters>
</MathVariable>
  
```



The following math variable named ServiceResult in this example illustrates how the configurer can query the result of a web service using an XPATH. Below the WebServices rule is explained in depth and describes how the XML results can be transformed into a more palatable format containing OIPA keys and other stored information related to the result.

```
<MathVariable VARIABLENAME="ServiceResult" TYPE="XML "
XPATH="/SomeXPath">ServiceXML</MathVariable>
```

## WEBSERVICE Attributes

### TYPE

This attribute indicates that the math variable is a web service. As a result the following two attributes are required for correct configuration.

### RULENAME

This points the processing engine at the rule that contains the details of how to build the message and consume this web service. In this case the rule is named “WebServices” but could just as easily have been named “Other Rule”. This option would not have made much sense.

### SERVICENAME

Within the rule many services can be nested. Below is a description of the WebServices business rule.

## Web Service Business Rule

This is a sample WebServices business rule built to match the above math variable and variable parameters. Within the tag WebServices can nest many “WebService” nodes for different service purposes. Each may be used by a different service request.

```
<WebServices>
  <WebService>
    <ServiceName>SampleService</ServiceName>
    <Parameters>
      <Parameter NAME="ParameterXML">
        <AsXML>
          <Target>AsXML</Target>
          <Param1>[Parameter1]</ Param1>
          < Param2>[ Parameter2]</ Param2>
        </AsXML>
      </Parameter>
    </Parameters>
    <RequestAppend>XML-Extensions</RequestAppend>
    <RequestTransform>XSLT-AdminServerToOther</RequestTransform>
    <ResponseAppend>XML-Extensions</ResponseAppend>
    <ResponseTransform>XSLT-OtherToAdminServer</ResponseTransform>
  </WebService>
```

**</WebServices>**

**<ServiceName>**

The name of the service being consumed by this request.

**<Parameters>**

Parameters node. Contains all variable and fixed values to be sent in the message.

**<Parameter>**

Contains the current state message.

**<AsXML>**

Opening tag for the current state message.

**<Target>**

Name for this message.

**<Param1>**

This is just a name chosen for this example. It could be any valid XML tag name. The value within is a variable substitution from the math variable "Parameter1" defined in the transaction above.

**<Param2>**

This is just a name chosen for this example. It could be any valid XML tag name. The value within is a variable substitution from the math variable "Parameter2" defined in the transaction above.

**<RequestAppend>**

This is the name of a business rule containing a list of XML tags used for reference by the transformation described in "RequestTransform" below. Basically, it appends the XML from this business rule to the current state message allowing the XSLT to reference code transformations for the purpose of sending codes understood by a downstream system.

**<RequestTransform>**

This is the name of a business rule containing the XSLT used to transform the current state message into a format understood by the service being consumed.

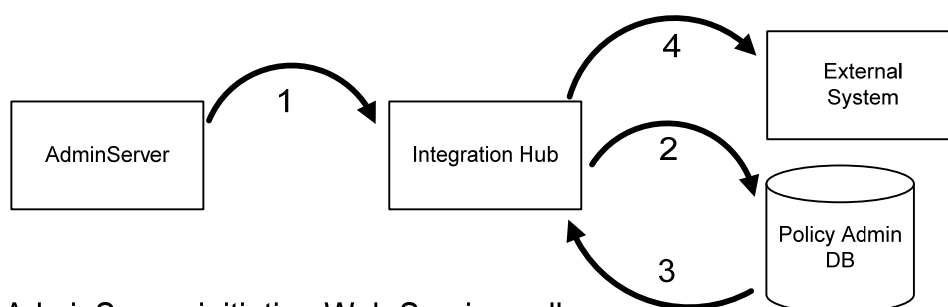
**<ResponseAppend>**

This is the name of a business rule containing a list of XML tags used for reference by the transformation described in "RequestAppend" below. Basically, it appends the XML from this business rule to the current state message allowing the XSLT to reference code transformations for the purpose of sending codes understood by a downstream system. It is usually the same rule referenced in "RequestAppend" so that code mappings are stored in a single location and as such are easier to maintain.

### <ResponseTransform>

This is the name of a business rule containing the XSLT used to transform the current state message into a format understood by the service being consumed.

## Outbound Complex (Push)



- 2.1 AdminServer initiating Web Service call
- 2.2 Integration Hub call to retrieve data (e.g. Stored Proc)
- 2.3 Query results returned from Policy Administration database
- 2.4 Formatted data delivered to downstream system (XML, Flat File, etc.)

This scenario represents the batch feed extract commonly associated with legacy endpoints or just larger message loads in general. For example, a General Ledger system would receive potentially too much traffic if it were to provide a Web Service for the purpose of receiving accounting information from all dependent systems. For this reason it typically receives a single file from each system detailing all accounting for the business day.

In this scenario the OIPA system depends on middleware to retrieve all information related to the batch feed when instructed to do so. First (1), OIPA notifies the middleware by means of a Web Service when ready for batch extract. Next (2), the middleware makes a call based on information in the message (1) to retrieve a recordset (3) from the AdminServer database. After transforming the retrieved data the middleware then delivers the batch data (4) to the endpoint system. This methodology takes advantage of the middleware's prowess in transformation and orchestration of such processes.

The work of creating the notification transaction (1) and gathering the data (2) is all done by the OIPA Configuration Team in conjunction with the OIPA Integration Team. All transformation logic is the work of the Middleware Team.

A sample transaction configuration for this scenario is illustrated below.

### Transaction Rule

```
<Transaction>
  <EffectiveDate STATUS="Disabled" TITLE="Effective Date" TYPE="SYSTEM"></EffectiveDate>
  <Fields>
    <Field>
      <Name>FieldName</Name>
      <Display>Display Name</Display>
      <DataType>Combo</DataType>
      <Query TYPE="SQL">asu_MissingActivities '1/1/2003', '[EffectiveDate]'</Query>
    </Field>
  </Fields>
  <Math>
    <MathVariables>
      <MathVariable VARIABLENAME="VariableName1"
TYPE="SQL">SELECT...'</MathVariable>
      <MathVariable VARIABLENAME="VariableName2"
TYPE="SQL">SELECT...'</MathVariable>
    </MathVariables>
  </Math>
</Transaction>
```

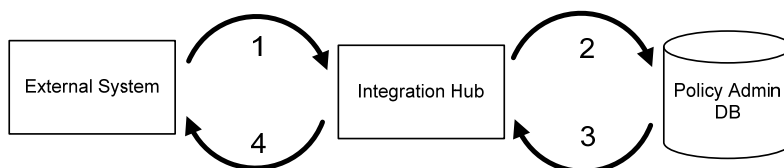
In this example the extract has one field as an input. This is likely a date indicating the date to be fed along with the stored procedure enabling the middleware to execute the query. Two math variables are also included in this example. These could also be passed as inputs to the stored procedure or potentially used to calculate a spawn date for the future pending copy of the extract transaction.

### Attached Business Rule (GenerateExtract)

```
<GenerateExtract>
  <ExtractName>ExtractNameHere</ExtractName>
  <Parameters>
    <Parameter NAME="ParameterName1">[ ParameterName1]</Parameter>
    <Parameter NAME=" ParameterName2">[ ParameterName2]</Parameter>
  </Parameters>
</GenerateExtract>
```

This rule denotes parameter values (taken from the associated transaction Math Variables) to be passed to the provider web service. A typical example of the parameters used in such a rule would be the name of a stored procedure along with any inputs to the stored procedure such as FromDate, ToDate, and PlanName. These rules are usually used as a method of notification to trigger an extract process where the notified middleware is to call such a stored procedure to retrieve all data required for a particular downstream system.

## Outbound (Pull)



- 3.1 Data delivery to hub
- 3.2 Integration Hub call to retrieve data (e.g. Stored Proc)
- 3.3 Query results returned from Policy Administration database
- 3.4 Formatted data delivered to downstream system (XML, Flat File, etc.)

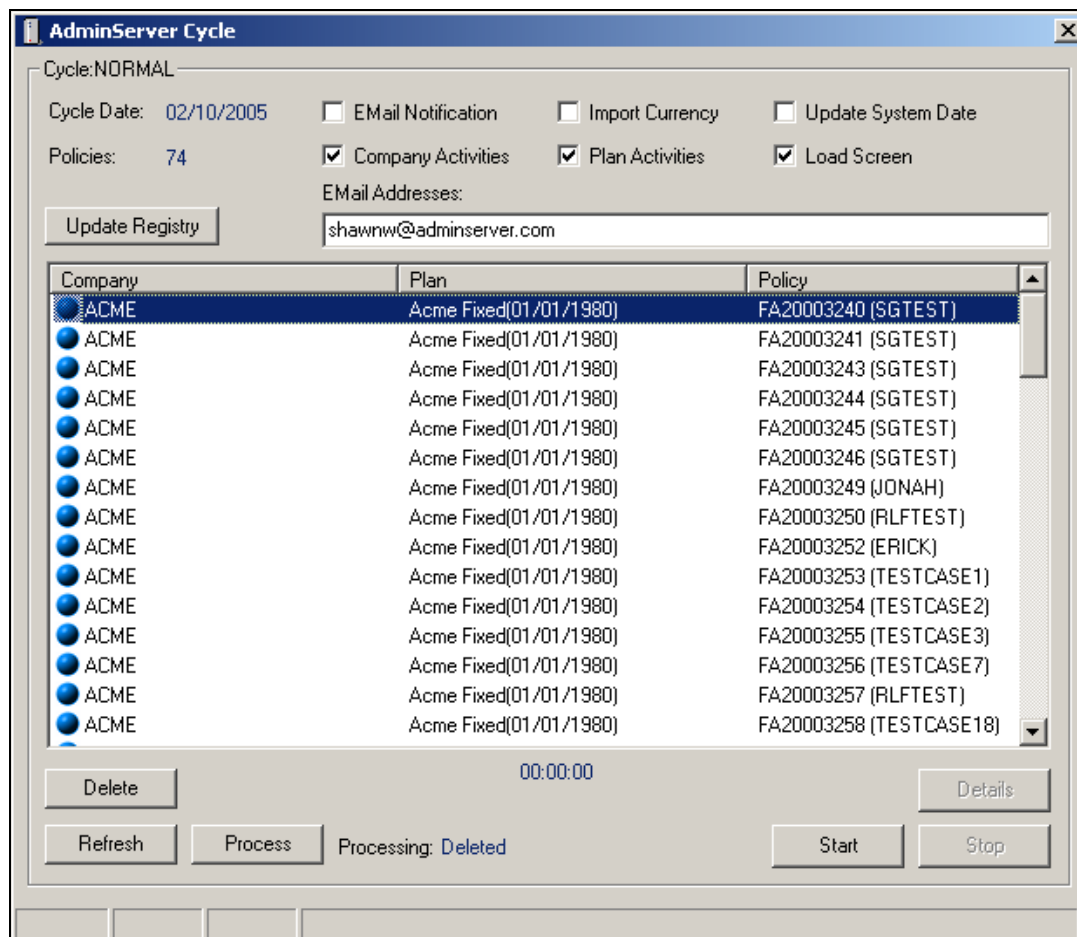
In this scenario there is little work for the Configuration or Integration Teams other than to provide the SQL queries necessary to gather the data. There are no transactions or web services to be designed and built. The external system simply instructs the middleware to call a set of pre-defined queries to retrieve some data from the OIPA database. The resultset is then returned to the middleware where it may or may not be transformed as needed. This call should be coordinated with the OIPA Integration Team to facilitate agreeable conditions and not to interfere with performance requirements of the database either during business hours or nightly processing.

## Custom Web Service

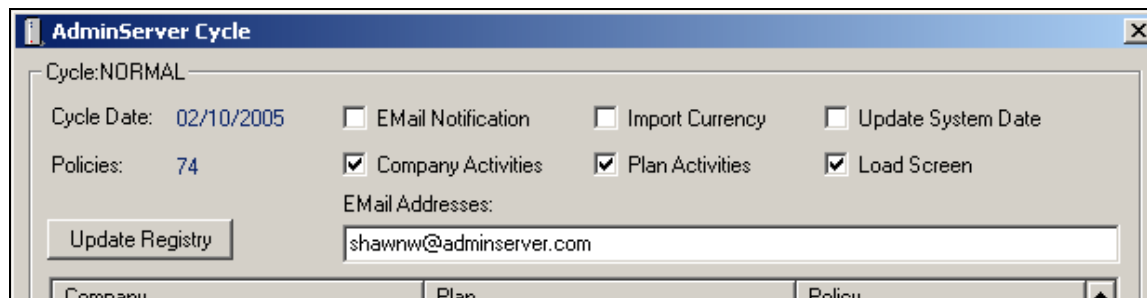
In cases where requirements can not be satisfied through the available methods there is always the option to have us build a custom web service. This is seldom necessary but the effort is comparable with that of configuring a complex web service through rules but usually slightly higher.

## AdminServerCycle

AdminServerCycle is the OIPA system's solution to batch processing. AdminServerCycle allows you to process company, plan and policy level activities in a graphical user interface (GUI). The GUI is designed to use a multi-thread approach to ensure quick processing and essential scalability.



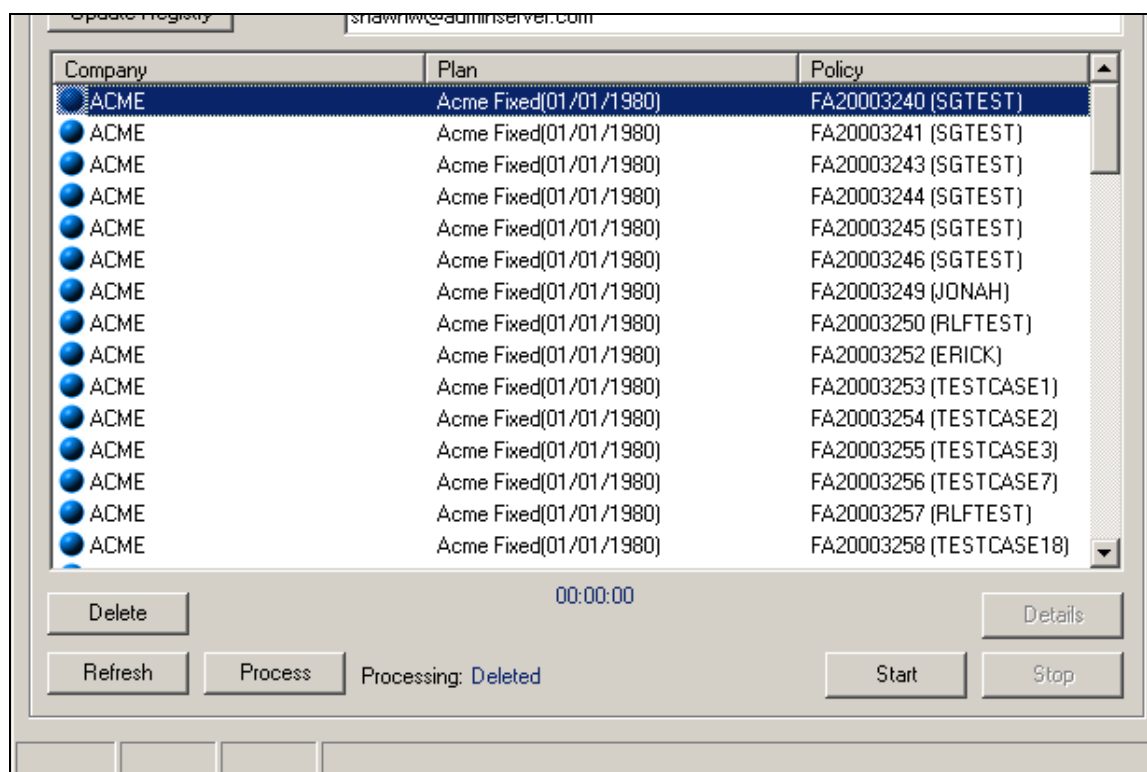
The AdminServerCycle GUI allows you to dynamically control batch processing. This is accomplished through the top portion of the GUI. As shown below these processing options include Email Notification, Import Currency, Update System Date, Company Activities, Plan Activities and Load Screen.



1. **Email Notification:** When checked, Email Notification will send various emails to the address in the Email Addresses text box. The various emails are specified in the registry under Software/AdminServer/Cycle/Email and restricted by other processing options. The super set of emails include: Cycle Start, Company Level(Prior) status, Plan Level(Prior) status, Batch Started, Cycle Paused, Batch Ended, Plan Level(After) status, Company Level(After) status, Cycle Ended and Cycle System Errors.
2. **Import Currency:** Saves different forms of currency and an exchange list into the OIPA system. Import Currency is rules driven via the DefaultCurrency business rule.
3. **Update System Date:** After the AdminServerCycle has completed the System Date will increment. The date can increment one calendar day or use different criteria in the table AsSequence. If cycle.nextSystemDate in the CycleUtil.properties file is set to "Calendar" then the OIPA SystemDate will always be set to the date on the server. Otherwise, the date will be set to the node /Flow/NextDate in AsSequence.XMLData where SequenceDate is equal to today's SystemDate.
4. **Company Activities:** If selected Cycle will process both company level activities before and after policy level activities (i.e. Company Level(Prior) and Company Level(After)).
5. **Plan Activities:** If selected Cycle will process both plan level activities before and after policy level activities (i.e. Plan Level(Prior) and Plan Level(After)).
6. **LoadScreen:** This option will toggle the display of policy in the policy table. The loading of this list can take some time. When cycle's load of policies is large you want to avoid displaying the policy list.

The option you have chosen can be saved via the Update Registry button at the following location in the registry: Software\AdminServer\Cycle. All activities that run during cycle will run under the OIPA client number SYSADMIN and any spawns during cycle will be spawned by SYSADMIN.

The AdminServerCycle GUI gives you several options on processing as well as information about what is being processed.



1. **Policy Table:** When the GUI is first opened this table will display all policies with pending activities that have an activity effective date less then or equal to the system date.
  - a. The bullet to the left of the policies shows different colors for different statuses.
    - i. Blue: Policy has not been processed.
    - ii. Green: Policy has successfully processed all pending activities.
    - iii. Red: A business rule error has occurred during processing.
2. **Delete Button:** Removes the highlighted policies from the table. This will prevent the policy from being processed when cycle is started.
3. **Refresh Button:** Repopulates the Policy Table as if it was just open. It will remove all successfully processed activities, repopulate all deleted policies and set all errant policies back to pending.
4. **Process Button:** Begins the cycle process.

Within Policy Level Activities, the first process is to insert all policies and pending activities from the policy table on the AdminServerCycle's main window into the table AsCycle with the cycle status set to pending.

Next, an assign thread pool is launched, and each of the pending activities is inserted into a work queue. The cycle application then binds a number of threads to each server as specified in the Cycle.properties files, and begins making remote calls into these servers to process all pending activities on the specified server.



The Cycle.properties file has the following format:

```
java.naming.factory.initial = org.jnp.interfaces.NamingContextFactory
java.naming.provider.url.machine = jnp://win2000-83:1099/,1
```

```
cycle.emailAddresses = carll@adminserver.com
cycle.companyActivities = Yes
cycle.emailNotification = yes
cycle.importCurrency = No
cycle.loadScreen = yes
cycle.planActivities = No
cycle.updateSystemDate = Yes
cycle.cycleClientNumber = SHAWNW
```

### **java.naming.factory.initial**

This property indicates the name of the class used for the initial naming context, and is specific to the application server that cycle will be driving.

### **java.naming.provider.url.machine**

This is a delimited list of server addresses and thread counts. The URL format conforms to the specification of the remote call protocol used to drive the application server. In this example, you connect to the machine win2000-83, which is running the JBoss application server. It will run two threads. The general format is:

```
<protocol>://<machine1>:<port>/,<threadcount>;<protocol>://<machine2>:<port>/,<threadcount>
```

### **cycle.emailAddresses**

The email address to send cycle notifications to.

### **cycle.companyActivities**

Whether or not to include/process company level activities.

### **cycle.emailNotification**

Whether or not to send email notifications.

### **cycle.importCurrency**

Whether or not to import currency.

### **cycle.loadScreen**

Whether or not to initialize the cycle table screen on startup.

**cycle.planActivities**

Whether or not to run plan level activities.

**cycle.updateSystemDate**

Whether or not to update the system date when cycle completes.

**cycle.cycleClientNumber**

The application user whose credentials will be used to run the cycle.