Informatica® ActiveVOS
9.2.4.6

7. APIs, SDKs, and Services

Informatica ActiveVOS 7. APIs, SDKs, and Services
9.2.4.6
March 2020

Publication Date: 2023-11-08

Table of Contents

Preface

This module contains information about the Informatica Business Process Manager Software Development Kit (SDK) with its application programming interfaces (APIs).

CHAPTER 1

Overview, Downloading, and Installing

You can download the Informatica Business Process Manager Software Development Kit (SDK) with its application programming interfaces (API) for developing custom applications .

`http://www.activevos.com/dev/sdks/com.activevos.api.zip`

When you unzip the SDK, it includes the following folders:

Directory	Description
.settings	Files required for using the Java project in Eclipse
AdminSDK	Process Server Administration API. Contains operations to communicate with the Server.
common	Java dependencies for builds and runtime of jax-ws clients
Identity Service	Identity Service API. Contains operations to get users and groups stored in a member directory tied to Process Server services.
orchestration	Contains two sample BPEL orchestration projects for use in Process Developer as well as a deployment archive for a third sample project
src	Empty folder for your source code
WSHT4J	Process Server Web Services Human Task (WS-HT) API. Contains operations for creating a custom BPEL for People application.
XML-JSON-Binding	Process Developer XML-JSON API. Contains documents and samples of HTML form building using the Process Central technologies.

Process Server Administration API

The Process Server provides an administration interface that is accessible via WSDL-defined web service invocations or directly using Java classes. The AdminSDK folder includes projects and documentation describing how to interact with the API using both techniques.

Process Server Identity Service API

Process Server provides an identity service to enable processes to look up users and groups in an enterprise directory service. The Identity Service folder describes the identity service API and includes a few examples on using the API via SOAP and indirectly via Java code using JAX-WS (IdentityService4J api).

Screenflow Programming SDK

The Platform SDK contains a business user tool for workflow development called the Guide Designer. The guide designer has a simple interface for creating steps such as screens and automated actions and they can use and modify data in a host system. The guides that you create are applications that run from within a web browser connecting to either an internal or external server. The browser can be within a mobile device such as an iPhone or within a PC or a computer created by Apple.

The Host Provider SDK and associated Service Programming Interface (SPI) allow developers to expose data and operations in their host system to the Designer. At the heart of the system are reusable services that are defined using the powerful Process Developer environment and deployed to the Process Server environment. These services implement the SPI necessary for providing a hosting environment (Host Provider) and are described to the Process Servr using a simple descriptor mechanism.

WS-HumanTask API

Process Server implements the operations described by the OASIS WS-HumanTask (WS-HT) API task client specifications to manage human tasks as well as interact with specific tasks. In addition to WS-HT API, the Process Server also provides an extension API to the OASIS WS-HT API with enhanced functionality. The documents in the WSHT4J folder introduce you to the Informatica implementation of the WS-HT specification and Process Server Extension APIs. It also includes a few examples describing the use of the API via SOAP and indirectly via Java code using JAX-WS (WSHT4J API)

We recommend that you read the `WS-HumanTask Architecture` document to get an appreciation of WS-HT. This document will help you understand concepts necessary to the use of the API.

XML-JSON-Binding API

Process Server exposes all its processes and services via simple XML and JavaScript Object Notation (JSON) bindings in addition to SOAP, REST and JMS. The XML (and JSON) binding makes it easy for application developers already familiar with XML/JSON-based REST application development to invoke processes and obtain responses from them.

Using XML or JSON bindings frees developers from the need to obtain and learn SOAP libraries to build applications that leverage Process Server service-based processes. This approach allows JavaScript developers to use various libraries such as jQuery to build process-enabled applications.

Process Server uses the JSON binding for Process Central request and task forms. This implementation provides functions that are available for use by developers for process-enabled application projects.

XML-JSON-Binding and Process Central API

Process Central is a client application packaged with Process Server that contains Process Request Forms, Tasks, and Reports. Users of Process Central can start a process that has been deployed to the server, work on tasks that are part of a running process, and view reports pertinent to tasks.

SDK Installation Instruction

Copy the SDK archive to your file system and launch Process Developer. Import the archive as follows:

1. In Process Developer, select File > Import.
2. Expand the General section, select Existing Projects into Workspace, and select Next.
3. Select the Select Archive File radio button and browse to the location of the archive.
4. Select the zip file and select **Finish**.
5. A project named `com.activevos.api` opens in your Project Explorer, showing the contents listed in the table above.

Note: This project contains two nested orchestration projects. We highly recommend that you import the orchestration projects into your workspace separately if you plan to modify them. Eclipse does not treat a child project as an exclusive project and your modified files will not be contained in the child project.

APIs and SDKs

The Informatica Business Process Manager Software Development Kit (SDK) provides application programming interfaces (API) for developing custom applications. Along with the APIs, the SDK project includes common libraries, resources, example code, and documentation to help you develop your application.

API or SDK	Description
Informatica Business Process Manager SDK	The Informatica Business Process Manager Software Development Kit (SDK) provides application programming interfaces (API) for developing custom applications. Along with the APIs, the SDK project includes common libraries, resources, example code, and documentation to help you develop your application. The components of this SDK are: - **AdminSDK**: Process Server Administration API. Contains operations to communicate with the Server. - **Identity Service**: Contains operations to get users and groups stored in a member directory tied to Process Server services. - **WSHT4J**: Web Services Human Task (WS-HT) API. Contains operations for creating a custom BPEL for People application. - **XML-JSON Binding**: XML-JSON API. Contains documents and samples of HTML form building using the Process Central technologies.
Alert for Faulting Processes Service	For many reasons, an activity may throw an uncaught fault. Process Server provides the ability to suspend a process on an uncaught fault, and if desired, you can also trigger an alert. For details, search for *Alert Service* elsewhere in this help.
Automated Step Operations	These operations are part of Guide Designer, which is described later in this table.
Data Access Service	You can interact directly with a data source within a BPEL process by using this service. This service lets you create an invoke activity that executes one or more SQL statements on a specified data source and receives a result set as the response. For details, see *Data Access Service* elsewhere in this help.
Email Service	Process Server supports an email service. This means you can create a process that sends email messages to a list of recipients using an invoke activity. Many types of processes can benefit from an email delivery activity, including use an Alert Service. For details, see *Email Service* elsewhere in this help.
Event Action Service	An event-action process receives and handles the events you define in other processes. For details, see *Creating an Event-Action BPEL Process* elsewhere in this help.
Execute Report Service	This service lets you create a BPEL process that sends a request to the server to take a snapshot of a deployed report, save the report as a PDF, MS Word, or other format, and return it to the process. Within your BPEL process, you provide programming logic to email the report, use it in a People activity, or send it to an endpoint based on an invoked service. You can schedule the process to run daily (or other frequency) to have a daily report emailed. For details, see *Reporting Service* elsewhere in this help.
Human Task Service	These are the core WS-HT operations
Identity Search Service	An identity service provides a way to look up users and groups based on a defined set of attributes. Process Server support for an identity service is based on the Lightweight Directory Access Protocol (LDAP), JDBC, or a file-based service. For details, see *Identity Service* elsewhere in this help.

API or SDK	Description
Migration Service	The migration service allows fine grain control over migrating running process instances to a newer (or different) version of a process definition. The simplest way to migrate running processes is to specify this choice in the Process Deployment Descriptor of the newer version being deployed, on the General Deployment Options page. For details, see "Migration Service" elsewhere in this help.
Monitor Alert Service	Process Server provides engine monitoring, and if desired, you can trigger an alert to notify an administrator when an engine stops running or when a monitored property has a warning or error condition. For details, see *Monitoring Alert Service* elsewhere in this help.
OAuth Service	If your business process needs a system whose resources can be accessed using OAuth authentication, you can use the OAuth system service to allow delegated access to private resources. For details, see *Using an OAuth REST-Based System Service* elsewhere in this help.
REST Service	You can create web service interaction activities in BPEL that can handle messages based on the Representational State Transfer (REST) architecture rather than WSDL operations. This means that you do not need your own WSDL file in order to create certain types of BPEL processes. You may also access partner REST services from a BPEL process. For details, see *Using a REST-based Service* elsewhere in this help.
Retry Invoked Service	At times an endpoint may not reply when it is invoked in a BPEL process. In normal circumstances, the invoke faults. To avoid this, you can attach a policy to the endpoint reference in the PDD file to indicate that additional retry attempts are allowed at a specified interval for the end point. Alternately, you can name a service in the policy section of the PDD file whose job it is to provide a retry interval, that you may want to calculate dynamically, and do additional work programmatically. The service could also specify an alternate endpoint to try when the invoke is faulting or suspended. For details, see "Retry-Policy Service" elsewhere in this help.
Guide Designer Automated Step Service	For details, see *About Guide Designer Host Provider SDK* elsewhere in this help.
Server Logging Service	Process Server provides a Server Log which captures the details of server events, such as configuration changes and errors in running processes. You can create a BPEL process to interact with the Server Log so that you can add your own message to the log programmatically. This interaction provides a way to troubleshoot your running processes. For details, see *Server Log Service* elsewhere in this help.
Shell Command Service	You can create an invoke activity that calls a POJO service to execute a shell command script. You can, for example, call a script that updates a database or adds a new user to an identity service. To use the shell command system service, you must create and locate your script in a working directory that is specified within the input message for the invoke activity. For details, see *Shell Command Invoke Service* elsewhere in this help.
Task Custom Notification	This process lets you create a custom process to execute for a people activity's notification deadline. For details, see *Human Tasks* elsewhere in this help.

CHAPTER 2

Administration API

This chapter includes the following topics:

Sample Messages

The Administration Service is exposed as a web service. If you have Process Server deployed on your localhost running on port 8080, the WSDL for the service can be found at: `http://localhost:8080/active-bpel/services/ActiveBpelAdmin?wsdl`.

The Administration Service exposes the operations shown in the table below. You can use the SOAP-UI or a similar tool (such as the Web Services Explorer included in the Process Developer) to interact with the Administration Service. For most operations, a sample SOAP request and response are shown. Some operations, such as the operations concerning breakpoints, are exposed for remote debugging and sample messages are not provided.

Operation	Description
`AddAttachment`	Adds an attachment to the variable specified at a variable path in a process.
`AddBreakpointListener`	Adds a listener for engine breakpoint notification events.
`DeployBPR`	Deploys a BPR file.
`GetAPIVersion`	Returns the API version for the BPEL administration service.
`GetConfiguration`	Gets the current engine configuration as XML.
`GetProcessCount`	Returns a count of processes currently running on the BPEL engine.
`GetProcessDef`	Returns the process definition (BPEL XML) for a process.
`GetProcessDetail`	Returns the process detail for the a process ID or null if the process does not exist on the server.

Operation	Description
GetProcessDigest	Returns the message digest code of the deployed BPEL file within a process.
GetProcessList	Returns a list of processes currently running on the BPEL engine.
GetProcessLog	Returns the process log for the a process if logging is enabled on the server.
GetProcessState	Returns the state of the process specified by a process ID.
GetServerLogList	Returns the list of logs.
GetVariable	Returns the data for the variable being referenced by the variable path.
IsInternalWorkManager	Returns True if using the native ActiveBPEL WorkManager and False if using a server provided WorkManager.
RemoveAttachments	Removes one or more attachments for the variable specified by a variable path in a process.
RemoveBreakpointListener	Remove a listener for engine breakpoint notification events.
RemoveEngineListener	Removes a listener from receiving engine notification events.
RemoveProcessListener	Removes the passed listener from list of those notified of process events for a PID.
ResumeProcess	Resumes the business process identified by a PID.
ResumeProcessContainer	Resumes the business process identified by a PID for the passed suspended location container.
ResumeProcessObject	Resumes the business process identified by a PID for the passed suspended location.
RetryActivity	Retries the activity associated with the passed location path or its enclosing scope.
SetConfiguration	Sets properties for the engine configuration.
SetCorrelationSetData	Sets the correlation set data for a PID and location path.
SetPartnerLinkData	Sets the partner link data for a PID and location path.
SetVariable	Sets the variable specified by a variable path in a process.
SuspendProcess	Suspends the business process identified by an PID.
TerminateProcess	Terminates the business process identified by an PID.
UpdateBreakpointList	Updates the list of breakpoints defined by the user for remote debugging.

AddAttachment

This operation adds an attachment to the variable specified by its variable path in a process. A sample SOAP request looks similar to the following message.

Note: The attachment is a base 64 encoded document that is added to the request as a SOAP attachment.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
          2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:addAttachmentDataInput>
       <act:pid>202</act:pid>
       <act:variablePath>/process/variables/variable[@name='TestSuspendRequest']
           </act:variablePath>
    </act:addAttachmentDataInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: The response contains one or more role names

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:addAttachmentDataOutput
    xmlns:ns2=
       "http://schemas.active-endpoints.com/activebpeladmin/2007/01/
                activebpeladmin.xsd"
    xmlns:ns3=
       "http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
    xmlns:ns4=
      "http://docs.active-endpoints/wsdl/activebpeladmin/2007/01/
                activebpeladmin.wsdl">
   <ns2:attachmentId>2</ns2:attachmentId>
   <ns2:attachmentAttributes>
      <ns2:attribute>
        <ns2:attributeName>Attachment-Created-At</ns2:attributeName>
        <ns2:attributeValue>1250256682654</ns2:attributeValue>
     </ns2:attribute>
     <ns2:attribute>
       <ns2:attributeName>Content-Id</ns2:attributeName>
       <ns2:attributeValue>1</ns2:attributeValue>
     </ns2:attribute>
     <ns2:attribute>
        <ns2:attributeName>Content-Transfer-Encoding</ns2:attributeName>
        <ns2:attributeValue>binary</ns2:attributeValue>
     </ns2:attribute>
     <ns2:attribute>
        <ns2:attributeName>Content-Type</ns2:attributeName>
        <ns2:attributeValue>application/octet-stream</ns2:attributeValue>
     </ns2:attribute>
     <ns2:attribute>
        <ns2:attributeName>X-Size</ns2:attributeName>
        <ns2:attributeValue>118</ns2:attributeValue>
     </ns2:attribute>
   </ns2:attachmentAttributes>
  </ns2:addAttachmentDataOutput>
 </S:Body>
</S:Envelope>
```

CompleteActivity

This operation steps resume the process and marks the activity associates with the passed location path as complete. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/2007/01/
              activebpeladmin.xsd">
  <soapenv:Header />
  <soapenv:Body>
    <act:completeActivityInput>
       <act:pid>202</act:pid>
       <act:locationPath>/process/sequence/extensionActivity/suspend
            </act:locationPath>
    </act:completeActivityInput>
```

```
  </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body/>
</S:Envelope>
```

DeployBPR

This operation deploys a BPR file. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
          2007/01/activebpeladmin.xsd">
  <soapenv:Header />
  <soapenv:Body>
      <act:deployBprInput>
         <act:bprFilename>riskAssessment.bpr</act:bprFilename>
         <act:base64File>base 64 encoded bpr file string inserted here</act:base64File>
      </act:deployBprInput>
  </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
    <ns2:deployBprOutput
       xmlns:ns2=
         "http://schemas.active-endpoints.com/activebpeladmin/2007/01/
            activebpeladmin.xsd"
       xmlns:ns3=
         "http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
       xmlns:ns4=
         "http://docs.active-endpoints/wsdl/activebpeladmin/2007/01/
            activebpeladmin.wsdl">
    <ns2:response><![CDATA[
      <deploymentSummary numErrors="0"
              numWarnings="1">
        <globalMessages>[riskAssessment.bpr] [riskAssessment.wsdl]
           Replaced resource mapped to location hint:
           project:/loan_approval_integrated/wsdl/riskAssessment.wsdl
           [riskAssessment.bpr] [loanRequest.xsd] Replaced resource mapped to location
           hint: project:/loan_approval_integrated/schema/loanRequest.xsd
           [riskAssessment.bpr] [riskAssessmentPLT.wsdl] Replaced resource mapped to
location
           hint: project:/loan_approval_integrated/wsdl/riskAssessmentPLT.wsdl
           [riskAssessment.bpr] [loanMessages.wsdl] Replaced resource mapped to location
           hint: project:/loan_approval_integrated/wsdl/loanMessages.wsdl
       </globalMessages>
       <deploymentInfo deployed="true" numErrors="0" numWarnings="1"
           pddName="riskAssessment.pdd" planId="26"
          runningProcessDisposition="maintain"
          version="1.0">
         <log>[riskAssessment.bpr]
          [riskAssessment.pdd] WARNING: Some rendering images refereneced in
          metadata document do not exist. Rendering images will not be deployed.
          If you have modified the process layout, please try re-creating the BPRD
          deploy script.
          [riskAssessment.bpr] [riskAssessment.pdd] Passed validation and was stored
          in database.
        </log>
      </deploymentInfo>
     </deploymentSummary>]]></ns2:response>
    </ns2:deployBprOutput>
 </S:Body>
</S:Envelope
```

GetAPIVersion

This operation returns the API version for the BPEL administration service. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
          2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:getAPIVersionInput/>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response contains the version of the administration API

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
    <ns2:getAPIVersionOutput
          xmlns:ns2=
            "http://schemas.active-endpoints.com/activebpeladmin/2007/01/
               activebpeladmin.xsd"
          xmlns:ns3=
               "http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
          xmlns:ns4=
            "http://docs.active-endpoints/wsdl/activebpeladmin/2007/01/
               activebpeladmin.wsdl">
        <ns2:response>6.0</ns2:response>
    </ns2:getAPIVersionOutput>
 </S:Body>
</S:Envelope>
```

GetConfiguration

This operation returns the current engine configuration as XML. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act=
    "http://schemas.active-endpoints.com/activebpeladmin/2007/01/
          activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
     <act:getConfigurationInput/>
 </soapenv:Body>
</soapenv:Envelope>
```

A sample SOAP response contains the server configuration is can be found in the topic titled Sample SOAP response for a server configuration.

GetProcessCount

This operation returns a count of processes currently running on the BPEL engine. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/2007/01/
           activebpeladmin.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header/>
 <soapenv:Body>
    <act:getProcessCountInput>
       <act:filter>
          <act:listStart>0</act:listStart>
          <act:maxReturn>500</act:maxReturn>
          <act:processCompleteEnd xsi:nil="true" />
          <act:processCompleteStart xsi:nil="true" />
          <act:processCreateEnd xsi:nil="true" />
          <act:processCreateStart xsi:nil="true" />
          <act:processName></act:processName>
          <act:processState>0</act:processState>
       </act:filter>
    </act:getProcessCountInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response contains the process count

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
    <ns2:getProcessCountOutput
      xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/2007/01/
           activebpeladmin.xsd"
      xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
      xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/2007/01/
           activebpeladmin.wsdl">
      <ns2:response>7</ns2:response>
   </ns2:getProcessCountOutput>
 </S:Body>
</S:Envelope>
```

GetProcessDef

This operation returns the process definition (BPEL XML) for an process. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/2007/01/
           activebpeladmin.xsd">
  <soapenv:Header/>
  <soapenv:Body>
    <act:getProcessDefInput>
       <act:pid>101</act:pid>
    </act:getProcessDefInput>
  </soapenv:Body>
</soapenv:Envelope>
```

A sample SOAP response that contains the process definition for a process with the process ID of 101 is in the topic titled Sample SOAP Response That Contains a Process Definition.

GetProcessDetail

This operation returns the process detail for a process ID or null if the process does not exist on the server. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/2007/01/
           activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:getProcessDetailInput>
       <act:pid>101</act:pid>
    </act:getProcessDetailInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response contains the process detail for a process with a process id of 101

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
   <ns2:getProcessDetailOutput
       xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/
              2007/01/activebpeladmin.xsd"
       xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
       xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/
              2007/01/activebpeladmin.wsdl">
     <ns2:response>
       <ns2:ended>2009-08-12T09:27:34.347-04:00</ns2:ended>
       <ns2:name xmlns:ns5="http://docs.active-endpoints.com/activebpel/
                sample/bpel/while/
                 2006/09/while.bpel">ns5:While</ns2:name>
       <ns2:processId>101</ns2:processId>
       <ns2:started>2009-08-12T09:27:33.347-04:00</ns2:started>
       <ns2:state>3</ns2:state>
       <ns2:stateReason>-1</ns2:stateReason>
    </ns2:response>
   </ns2:getProcessDetailOutput>
```

```
 </S:Body>
</S:Envelope>
```

GetProcessDigest

This operation returns the message digest code of the deployed BPEL file for a process. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:getProcessDigestInput>
       <act:pid>101</act:pid>
    </act:getProcessDigestInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response contains the process digest for a process with the process id of 101

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
    <ns2:getProcessDigestOutput
        xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd"
        xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
        xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/
           2007/01/activebpeladmin.wsdl">
     <ns2:digest>Xgp1J8IDUUGrhJQv1i3QvA==</ns2:digest>
   </ns2:getProcessDigestOutput>
 </S:Body>
</S:Envelope>
```

GetProcessList

This operation returns a list of processes currently running on the BPEL engine. The list of processes returned can be controlled by a filter. The comments in the sample SOAP request shown below provide values for the `processState` and `advancedQuery` elements.

Note: You can easily create a valid value for the `xmlns="http://www.w3.org/1999/xhtml">advancedQuery` element by using the expression builder of the Advanced Query option on the Active Processes page of the Process Console.

The following SOAP message shows two ways to return a list of completed processes.

- Setting the `act:processState` element to 2
- Setting the `advancedQuery` element to `getProcessProperty("State")=3`.

More information on using the process filter can be found by searching for Active Processes elsewhere in this help.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
              2007/01/activebpeladmin.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Header />
 <soapenv:Body>
    <act:getProcessListInput>
       <act:filter>
          <act:listStart>0</act:listStart>
          <act:maxReturn>500</act:maxReturn>
          <act:processCompleteEnd xsi:nil="true"/>
          <act:processCompleteStart xsi:nil="true"/>
          <act:processCreateEnd xsi:nil="true"/>
          <act:processCreateStart xsi:nil="true"/>
          <act:processName></act:processName>
             <!-- property codes for processState
             0 - Any
```

```
                  1 - Running
                  2 - Completed
                  3 - Faulted
                  4 - Completed and Faulted
                  5 - Suspended
                  6 - Suspended(Faulting)
                  7 - Suspended(Activity)
                  8 - Suspended(Manual)
                  9 - Compensatable
                  10 - Suspended(Invoke Recovery)
                  11 - Running and Suspended
                  -->
               <act:processState>2</act:processState>

               <!-- property codes for the "State" property as used in the
               getProcessProperty("State") = '1' advancedQuery
               1 - Running
               2 - Suspended
               3 - Completed
               4 - Faulted
               5 - Compensatable
               -->
               <act:advancedQuery>getProcessProperty("State")=3</act:advancedQuery>
            </act:filter>
      </act:getProcessListInput>
   </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response containing a list of completed processes

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
  <ns2:getProcessListOutput
     xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/
              2007/01/activebpeladmin.xsd"
     xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
     xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/
              2007/01/activebpeladmin.wsdl">
    <ns2:response>
       <ns2:totalRowCount>2</ns2:totalRowCount>
       <ns2:completeRowCount>true</ns2:completeRowCount>
       <ns2:rowDetails>
          <ns2:item>
             <ns2:ended>2009-08-27T09:04:04.338-04:00</ns2:ended>
             <ns2:name xmlns:ns5="TerminateRunningProcesses">
                 ns5:TerminateRunningProcesses</ns2:name>
             <ns2:processId>110</ns2:processId>
             <ns2:started>2009-08-27T09:02:22.150-04:00</ns2:started>
             <ns2:state>3</ns2:state>
             <ns2:stateReason>2</ns2:stateReason>
          </ns2:item>
          <ns2:item>
             <ns2:ended>2009-08-21T10:44:23.858-04:00</ns2:ended>
             <ns2:name xmlns:ns5="http://www.example.org/helloWorld"
                       >ns5:helloWorld</ns2:name>
             <ns2:processId>1</ns2:processId>
             <ns2:started>2009-08-21T10:44:23.655-04:00</ns2:started>
             <ns2:state>3</ns2:state>
             <ns2:stateReason>-1</ns2:stateReason>
          </ns2:item>
       </ns2:rowDetails>
       <ns2:empty>false</ns2:empty>
    </ns2:response>
  </ns2:getProcessListOutput>
 </S:Body>
</S:Envelope>
```

GetProcessLog

This operation returns the process log for a process if logging is enabled on the server. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
    <soapenv:Body>
       <act:getProcessLogInput>
          <act:pid>101</act:pid>
       </act:getProcessLogInput>
    </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response contains the process log for a process with the process ID of 101

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
 <ns2:getProcessLogOutput
     xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd"
     xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
     xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/
           2007/01/activebpeladmin.wsdl">
    <ns2:response>[101][2009-08-12 09:27:33.550] Process : Executing [/process]
[101][2009-08-12 09:27:33.550] Flow : Executing [/process/flow]
[101][2009-08-12 09:27:33.550] Receive : Executing
            [/process/flow/receive]
[101][2009-08-12 09:27:33.565] Message received on /process/flow/receive
[101][2009-08-12 09:27:33.581] Receive : Completed normally [/process/flow/receive]
[101][2009-08-12 09:27:33.581] Link L1:
        status is true [/process/flow/links/link[@name='L1']]
[101][2009-08-12 09:27:33.581] Assign InitializeVariables: Executing
        [/process/flow/assign[@name='InitializeVariables']]

    ...

[101][2009-08-12 09:27:33.628] Reply : Executing [/process/flow/reply]
[101][2009-08-12 09:27:34.347] Replied to message from /process/flow/reply
[101][2009-08-12 09:27:34.347] Reply : Completed normally [/process/flow/reply]
[101][2009-08-12 09:27:34.347] Flow : Completed normally [/process/flow]
[101][2009-08-12 09:27:34.347] Process : Completed normally [/process]<
           /ns2:response>
    </ns2:getProcessLogOutput>
 </S:Body>
</S:Envelope>yyyy
```

You can find a more complete example in the topic titled Sample Soap Response.

GetProcessState

This operation returns the state of the process specified by a process ID. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:getProcessStateInput>
       <act:pid>101</act:pid>
    </act:getProcessStateInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response contains the process state for process 101

GetServerLogList

This operation returns the list of logs. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
</soapenv:Envelope>
```

Example: A sample SOAP response to this messasge

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
</soapenv:Envelope>
```

GetVariable

This operation returns the data for the variable being referenced by the variable path. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/2007/01/
            activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:getVariableDataInput>
       <act:pid>101</act:pid>
    <act:variablePath>/process/variables/variable[@name='orderTotal']
           </act:variablePath>
    </act:getVariableDataInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response containing the contents of the orderTotal variable for a process with process id of 101

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getVariableDataOutput
             xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/
                    2007/01/activebpeladmin.xsd"
             xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
             xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/
                    2007/01/activebpeladmin.wsdl">
         <ns2:response>&lt;variable dataIncluded="yes"
               hasAttachments="false" hasData="true"
               name="orderTotal"
               type="{http://www.w3.org/2001/XMLSchema}float"
               version="68">6985.8594&lt;/variable></ns2:response>
    </ns2:getVariableDataOutput>
 </S:Body>
</S:Envelope>
```

RemoveAttachments

This operation removes one or more attachments for a variable a process. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:removeAttachmentDataInput>
       <act:pid>202</act:pid>
       <act:variablePath>/process/variables/variable[@name='TestSuspendRequest']
                 </act:variablePath>
       <act:itemNumbers>
           <!--Zero or more repetitions:-->
           <act:itemNumber>1</act:itemNumber>
       </act:itemNumbers>
    </act:removeAttachmentDataInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 <S:Body>
       <ns2:removeAttachmentDataOutput
            xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/
                 2007/01/activebpeladmin.xsd"
            xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
            xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/
                 2007/01/activebpeladmin.wsdl">
          <ns2:response/>
      </ns2:removeAttachmentDataOutput>
 </S:Body>
</S:Envelope>
```

ResumeProcess

This operation resumes the business process identified by the passed PID. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:act=
         "http://schemas.active-endpoints.com/activebpeladmin/
            2007/01/activebpeladmin.xsd">
  <soapenv:Header/>
  <soapenv:Body>
     <act:resumeProcessInput>
       <act:pid>203</act:pid>
     </act:resumeProcessInput>
  </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body/>
</S:Envelope>
```

ResumeProcessContainer

This operation resumes the business process identified by the passed PID for the passed suspended location container. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
              2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:resumeProcessContainerInput>
       <act:pid>303</act:pid>
       <act:location>/process/sequence/scope[@name='MyScope']</act:location>
    </act:resumeProcessContainerInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body/>
</S:Envelope>
```

ResumeProcessObject

This operation resumes the business process identified by a PID for a suspended location. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
  <soapenv:Header/>
     <soapenv:Body>
        <act:resumeProcessObjectInput>
```

```
            <act:pid>303</act:pid>
            <act:location>/process/sequence/scope[@name='MyScope']/flow/sequence/
               extensionActivity/suspend</act:location>
         </act:resumeProcessObjectInput>
      </soapenv:Body>
   </soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body/>
</S:Envelope>
```

RetryActivity

This operation retries the activity associated with a location path or its enclosing scope. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
  <soapenv:Header/>
  <soapenv:Body>
     <act:retryActivityInput>
         <act:pid>303</act:pid>
         <act:locationPath>/process/sequence/scope[@name='MyScope']/flow/
            sequence/empty[@name='EmptyActivity_1']</act:locationPath>
          <act:atScope>true</act:atScope>
     </act:retryActivityInput>
  </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body/>
</S:Envelope>
```

SetConfiguration

This operation sets the engine configuration properties.

Note: You should retrieve the current configuration using the `GetConfiguration` operation and make changes to the message returned by that operation prior to setting the configuration.

Example: A sample SOAP response to this message:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body/>
</S:Envelope>
```

SetCorrelationSetData

This operation sets the correlation set data for a process id and location path. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
              2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:setCorrelationDataInput>
       <act:pid>301</act:pid>
       <act:locationPath>/process/correlationSets/correlationSet[@name='CS1']</
                 act:locationPath>
       <act:data>
          <![CDATA[
          <property name="QuoteRefId"
              namespaceURI="http://docs.active-endpoints.com/activebpel/demo/
              quoteRequest/2008/06/
              QuoteRequest.wsdl"
          value="301a"/>
```

```
          ]]>
        </act:data>
     </act:setCorrelationDataInput>
  </soapenv:Body>
 </soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
    <setCorrelationDataInputResponse
         xmlns="http://schemas.active-endpoints.com/activebpeladmin/
              2007/01/activebpeladmin.xsd"/>
 </soapenv:Body>
</soapenv:Envelope>
```

SetPartnerLinkData

This operation sets the partner link data for a process ID and location path. A sample SOAP is looks similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
  <soapenv:Header/>
  <soapenv:Body>
    <act:setPartnerLinkDataInput>
       <act:pid>301</act:pid>
       <act:partnerRole>true</act:partnerRole>
       <act:locationPath>/process/partnerLinks/partnerLink[@name='Rules_Service']</
                 act:locationPath>
          <act:data>
            <![CDATA[
            <partnerRole>
              <wsa:EndpointReference xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/
                03/addressing"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:ns5="http://www.active-endpoints.com/demo/wsdl/QuoteRules/
                      2008/06/QuoteRules.wsdl"
                  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
              <wsa:Address>http://localhost:9080/active-bpel/services/
                      QuoteRulesService2</wsa:Address>
              <wsa:ServiceName PortName="QuoteRulesPort">ns5:QuoteRulesService</
                      wsa:ServiceName>
            </wsa:EndpointReference>
          </partnerRole>
          ]]>
       </act:data>
    </act:setPartnerLinkDataInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
       <setPartnerLinkDataInputResponse
          xmlns="http://schemas.active-endpoints.com/activebpeladmin/
                 2007/01/activebpeladmin.xsd"/>
    </soapenv:Body>
</soapenv:Envelope>
```

SetVariable

This operation sets the variable specified by a variable path a process with the given data. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
                  2007/01/activebpeladmin.xsd">
    <soapenv:Header/>
    <soapenv:Body>
       <act:setVariableDataInput>
          <act:pid>302</act:pid>
          <act:variablePath>/process/variables/
            variable[@name='TestSuspendRequest']</act:variablePath>
          <act:variableData>
          <![CDATA[
             <wsdl:part xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
                <TestSuspendRequest xmlns="urn:testSuspendRequest">
                   <pid>302</pid>
                </TestSuspendRequest>
             </wsdl:part>
          ]]>
          </act:variableData>
       </act:setVariableDataInput>
   </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
     <ns2:setVariableDataOutput
        xmlns:ns2="http://schemas.active-endpoints.com/activebpeladmin/
             2007/01/activebpeladmin.xsd"
        xmlns:ns3="http://schemas.active-endpoints.com/logging/2009/05/logging.xsd"
        xmlns:ns4="http://docs.active-endpoints/wsdl/activebpeladmin/
             2007/01/activebpeladmin.wsdl">
      <ns2:response/>
    </ns2:setVariableDataOutput>
 </S:Body>
</S:Envelope>
```

SuspendProcess

This operation suspends the business process identified by a PID. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
                 2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:suspendProcessInput>
       <act:pid>101</act:pid>
    </act:suspendProcessInput>
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this message

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body/>
</S:Envelope>
```

TerminateProcess

This operation terminates the business process identified by a PID. A sample SOAP request is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:act="http://schemas.active-endpoints.com/activebpeladmin/
           2007/01/activebpeladmin.xsd">
 <soapenv:Header/>
 <soapenv:Body>
    <act:terminateProcessInput>
       <act:pid>101</act:pid>
    </act:terminateProcessInput>
```

```
 </soapenv:Body>
</soapenv:Envelope>
```

Example: A sample SOAP response to this messasge

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body/>
</S:Envelope>
```

Other Server Admin APIs

Catalog API

The elements described in the following table are contained within an `<wsdl:portType name="IAeCatalogManagement">`

Operation Name	Operation	Description

deleteItem	`<wsdl:operation name="deleteCatalog">`
	`<wsdl:input message="tns:deleteCatalog" />`
	`<wsdl:output message="tns:deleteCatalogResponse" />`
	`<wsdl:fault`
	`message="tns:AdminAPIFault"`
	`name="AdminAPIFault" />`
	`</wsdl:operation>`
		Deletes an item in a catalog.

getItem	`<wsdl:operation name="getItem">`
	`<wsdl:input message="tns:GetItem" />`
	`<wsdl:output message="tns:GetItemResponse" />`
	`<wsdl:fault message="tns:AdminAPIFault"`
	`name="AdminAPIFault" />`
	`</wsdl:operation>`
		Retrieves an intem within a catalog.

insertItem	`<wsdl:operation name="insertItem">`
	`<wsdl:input message="tns:InsertItem" />`
	`<wsdl:output message="tns:InsertItemResponse" />`
	`<wsdl:fault message="tns:AdminAPIFault"`
	`name="AdminAPIFault" />`
	`</wsdl:operation>`
		Inserts an item into a catalog.

searchCatalog	`<wsdl:operation name="searchCatalog">`
	`<wsdl:input message="tns:SearchCatalog" />`
	`<wsdl:output message="tns:SearchCatalogResponse" />`
	`<wsdl:fault message="tns:AdminAPIFault"`
	`name="AdminAPIFault" />`
	`</wsdl:operation>`
		Searches a catalog for an item.

updateItem	`<wsdl:operation name="updateItem">`
	`<wsdl:input message="tns:UpdateItem" />`
	`<wsdl:output message="tns:UpdateItemResponse" />`
	`<wsdl:fault message="tns:AdminAPIFault"`
	`name="AdminAPIFault" />`
	`</wsdl:operation>`
		Updates an item in a catalog.

Contribution API

The elements described in the following table are contained within an `<wsdl:portType name="IAeContributionManagement">`

Operation Name	Operation	Description

deleteContribution	`<wsdl:operation name="deleteContribution">`
	`  <wsdl:input  message="tns:DeleteContribution"/>`
	`  <wsdl:output`
	
	`message="tns:DeleteContributionResponse"/>`
	`  <wsdl:fault message="tns:AdminAPIFault"`
	`  name="AdminAPIFault" />`
	`</wsdl:operation>`
		Deletes a contribution.

deletePlan	`<wsdl:operation name="deletePlan">`
	`  <wsdl:input  message="tns:DeletePlan" />`
	`  <wsdl:output message="tns:DeletePlanResponse" />`
	`  <wsdl:fault message="tns:AdminAPIFault"`
	`  name="AdminAPIFault" />`
	`</wsdl:operation>`
		Deletes a plan.

getContributionDetail	`<wsdl:operation name="getContributionDetail">`
	`  <wsdl:input message="tns:GetContributionDetail" />`
	`  <wsdl:output`
	
	`message="tns:GetContributionDetailResponse" />`
	`</wsdl:operation>`
		Obtains information about a contribution.

getDeployedProcessDetail	`<wsdl:operation name="getDeployedProcessDetail">`
	`  <wsdl:input`
	`message="tns:getDeployedProcessDetail" />`
	`  <wsdl:output`
	
	`message="tns:getDeployedProcessDetailResponse" />`
	`  <wsdl:fault message="tns:AdminAPIFault"`
	`  name="AdminAPIFault" />`
	`</wsdl:operation>`
		Retrieves information about a deployed process

getDeploymentLog	`<wsdl:operation name="getDeploymentLog">`
	`  <wsdl:input  message="tns:GetDeploymentLog" />`
	`  <wsdl:output`
	
	`message="tns:GetDeploymentLogResponse" />`
	`  <wsdl:fault message="tns:AdminAPIFault"`
	`  name="AdminAPIFault" />`
	`</wsdl:operation>`
		Retrieves a contribution's deployment log.

getDeployedProcessVersionDetail	`<wsdl:operation`
	`name="getDeployedProcessVersionDetail">`
	`  <wsdl:input`
	`message="tns:getDeployedProcessVersionDetail" />`
	`  <wsdl:output`
	
	`message="tns:getDeployedProcessVersionDetailResponse"`
	`/>`
	`  <wsdl:fault message="tns:AdminAPIFault"`
	`  name="AdminAPIFault" />`
	`</wsdl:operation>`
		Retrieves information about a version of a deployed process

Operation Name	Operation	Description
purgeOffline	<wsdl:operation name="purgeOffline"> <wsdl:input message="tns:PurgeOffline" /> <wsdl:output message="tns:PurgeOfflineResponse" /> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Tells Process Server to purge the contribution when the application is offline.
searchContribution	<wsdl:operation name="searchContributions"> <wsdl:input message="tns:SearchContributions" /> <wsdl:output message="tns:SearchContributionsResponse" /> </wsdl:operation>	Looks for a contribution.
searchDeployedProcesses	<wsdl:operation name="searchDeployedProcessses"> <wsdl:input message="tns:searchDeployedProcessses" /> <wsdl:output message="tns:searchDeployedProcesssesResponse" /> </wsdl:operation>	Looks for a deployed process.
searchDeployedServices	<wsdl:operation name="searchDeployedServices"> <wsdl:input message="tns:searchDeployedServices" /> <wsdl:output message="tns:searchsearchDeployedServicesResponse" /> </wsdl:operation>	Looks for a service definition
searchIndexedProperties	<wsdl:operation name="searchIndexedProperties"> <wsdl:input message="tns:searchIndexedProperties" /> <wsdl:output message="tns:searchIndexedPropertiesResponse" /> </wsdl:operation>	Looks for an indexed property.
searchPartnerDefinitions	<wsdl:operation name="searchPartnerDefinitions"> <wsdl:input message="tns:searchIndexedProperties" /> <wsdl:output message="tns:searchsearchIndexedPropertiesResponse" / > </wsdl:operation>	Looks for a partner definition.
searchTaskProperties	<wsdl:operation name="searchTaskProperties"> <wsdl:input message="tns:searchTaskProperties" /> <wsdl:output message="tns:searchTaskPropertiesResponse" /> </wsdl:operation>	

Operation Name	Operation	Description
setContributionOnline	<wsdl:operation name="setContributionOnline"> <wsdl:input message="tns:SetContributionOnline" /> <wsdl:output message="tns:SetContributionOnlineResponse" /> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Places a contribution online.

setPlanOnline	<wsdl:operation name="setPlanOnline">
	  <wsdl:input message="tns:SetPlanOnline" />
	  <wsdl:output
	  message="tns:SetPlanOnlineResponse" />
	  <wsdl:fault message="tns:AdminAPIFault"
	  name="AdminAPIFault" />
	</wsdl:operation>
		Places a plan online.

takeContributionDetail	<wsdl:operation name="takeContributionOffline"> <wsdl:input message="tns:TakeContributionOffline" /> <wsdl:output message="tns:TakeContributionOfflineResponse" /> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Obtains contribution details.
takePlanOffline	<wsdl:operation name="takePlanOffline"> <wsdl:input message="tns:TakePlanOffline"/> <wsdl:output message="tns:TakePlanOfflineResponse"/> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Places the plan offline.
takePlansOffline	<wsdl:operation name="takePlasOffline"> <wsdl:input message="tns:TakePlansOffline"/> <wsdl:output message="tns:TakePlansOfflineResponse"/> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Places all plans offline.
updatePlan	<wsdl:operation name="updatePlan"> <wsdl:input message="tns:updatePlan"/> <wsdl:output message="tns:updatePlanResponse"/> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Updates a plan.

Process Monitoring API

The elements described in the following table are contained within an `<wsdl:portType name="IAeProcessManagement">` element.

Operation Name	Operation	Description

getAlarmList	`<wsdl:operation name="getAlarmList">`
	`  <wsdl:input  message="tns:GetAlarmList" />`
	`  <wsdl:output message="tns:GetAlarmListResponse" />`
	`  <wsdl:fault  message="tns:ApiFault"`
	`name="apiFault" />`
	`</wsdl:operation>`
		Returns a list of the process's alarms.

getApiVersion	`<wsdl:operation name="getApiVersion">`
	`  <wsdl:input  message="tns:GetApiVersion" />`
	`  <wsdl:output message="tns:GetApiVersionResponse" />`
	`  <wsdl:fault  message="tns:ApiFault"`
	`name="apiFault" />`
	`</wsdl:operation>`
		Returns the process's API version.

getMessageReceiveList	`<wsdl:operation name="getMessageReceiverList">`
	`  <wsdl:input  message="tns:GetMessageReceiverList" />`
	`  <wsdl:output`
	`message="tns:GetMessageReceiverListResponse" />`
	`  <wsdl:fault  message="tns:ApiFault"`
	`name="apiFault" />`
	`</wsdl:operation>`
		Returns a list of received messages.

getProcessList	`<wsdl:operation name="getProcessList">`
	`  <wsdl:input  message="tns:GetProcessList" />`
	`  <wsdl:output message="tns:GetProcessListResponse" />`
	`  <wsdl:fault  message="tns:ApiFault"`
	`name="apiFault" />`
	`</wsdl:operation>`
		Returns a list of processes.

getProcessLog	`<wsdl:operation name="getProcessLog">`
	`  <wsdl:input  message="tns:GetProcessLog" />`
	`  <wsdl:output message="tns:GetProcessLogResponse" />`
	`  <wsdl:fault  message="tns:ApiFault"`
	`name="apiFault" />`
	`</wsdl:operation>`
		Returns the process log.

getProcessState	`<wsdl:operation name="getProcessState">`
	`  <wsdl:input  message="tns:GetProcessState" />`
	`  <wsdl:output`
	`message="tns:GetProcessStateResponse" />`
	`  <wsdl:fault  message="tns:ApiFault"`
	`name="apiFault" />`
	`</wsdl:operation>`
		Returns the process state.

Scheduler API

The Scheduler API lets you capture the following information:

- Disabled
- End Time
- Last Execution Date
- Last Process ID

- Last Process State
- Next Execution Date
- Plan ID
- Process Schedule
- Schedule Date
- Schedule Every
- Schedule Frequency
- Schedule Name
- Schedule Restriction
- Schedule Time
- Service Name
- Start Time
- Trigger Info

The elements described in the following table are contained within an `<wsdl:portType name="IAeScheduleManagement">` element.

Operation Name	Operation	Description

createDatabaseMaintenanceSchedule	`<wsdl:operation name="createDatabaseMaintenanceSchedule">`
	`  <wsdl:input message="tns:createDatabaseMaintenanceSchedule" />`
	`  <wsdl:output message="tns:createDatabaseMaintenanceScheduleeResponse" />`
	`  <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" />`
	`</wsdl:operation>`
		Creates a maintenance schedule for a database.

createSchedule	`<wsdl:operation name="createSchedule">`
	`  <wsdl:input message="tns:CreateSchedule" />`
	`  <wsdl:output message="tns:CreateScheduleResponse" />`
	`  <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" />`
	`</wsdl:operation>`
		Creates a schedule.

deleteSchedule	`<wsdl:operation name="deleteSchedule">`
	`  <wsdl:input message="tns:DeleteSchedule" />`
	`  <wsdl:output message="tns:DeleteScheduleResponse" />`
	`  <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" />`
	`</wsdl:operation>`
		Deletes a schedule.

editSchedule	`<wsdl:operation name="editSchedule">`
	`  <wsdl:input message="tns:EditSchedule" />`
	`  <wsdl:output message="tns:EditScheduleResponse" />`
	`  <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" />`
	`</wsdl:operation>`
		Changes items within a schedule.

getAllSchedules	`<wsdl:operation name="getAllSchedules">`
	`  <wsdl:input message="tns:GetAllSchedules" />`
	`  <wsdl:output message="tns:GetAllSchedulesResponse" />`
	`</wsdl:operation>`
		Returns information on all schedules.

Operation Name	Operation	Description
getDatabaseMaintenanceSchedule	<wsdl:operation name="getDatabaseMaintenanceSchedule"> <wsdl:input message="tns:getDatabaseMaintenanceSchedule" /> <wsdl:output message="tns:getDatabaseMaintenanceScheduleResponse" /> </wsdl:operation>	Returns information on the maintenance schedule for your databases.
runDatabaseMaintenance	<wsdl:operation name="runDatabaseMaintenance"> <wsdl:input message="tns:runDatabaseMaintenance" /> <wsdl:output message="tns:runDatabaseMaintenanceResponse" /> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Initiates the maintenance for a database.
runNow	<wsdl:operation name="runNow"> <wsdl:input message="tns:RunNow" /> <wsdl:output message="tns:RunNowResponse" /> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Tells a scheduled item that it should run now instead of at its scheduled time.
setScheduleStatus	<wsdl:operation name="setScheduleStatus"> <wsdl:input message="tns:SetScheduleStatus" /> <wsdl:output message="tns:SetScheduleStatusResponse" /> <wsdl:fault message="tns:AdminAPIFault" name="AdminAPIFault" /> </wsdl:operation>	Sets the status for a schedule.

URN Mapping API

Within Process Server, you can use system mapping and user-defined mappings for a URN. For information on URN mappings, look for URN Mappings elsewhere in this help.

The elements described in the following table are contained within an `<wsdl:portType name="IAeURNManagement">`

Operation Name	Operation	Description
deleteURN	<wsdl:operation name="deleteURN"> <wsdl:input message="tns:DeleteURN" /> <wsdl:output message="tns:DeleteURNResponse" /> </wsdl:operation>	Deletes a URN.
getAllURNDefinitions	<wsdl:operation name="getAllUrnDefinitions"> <wsdl:input message="tns:GetAllUrnDefinitions" /> <wsdl:output message="tns:GetAllUrnDefinitionsResponse" /> </wsdl:operation>	Returns information on all URN definitions.

Operation Name	Operation	Description
getUrnDefinitions	<wsdl:operation name="getUrnDefinitions"> <wsdl:input message="tns:getUrnDefinitions" /> <wsdl:output message="tns:getUrnDefinitionsResponse" /> </wsdl:operation>	Returns information on one URN.
upsertURN	<wsdl:operation name="upsertURN"> <wsdl:input message="tns:UpsertURN" /> <wsdl:output message="tns:UpsertURNResponse" /> </wsdl:operation>	Upserts (inserts; switches to update if already exists) a URN.

Sample Business Process

Included in the SDK is a sample orchestration project that demonstrates the use Administration API (a system service) by getting a list of running processes and then terminating them. You can work with this project by navigating to the orchestration folder and importing and existing project in to your workspace.

The process will receive a `getProcessList()` request and return a list of processes based on the filter criteria present in the request message. The process suspends so that you can review the contents of the `getProcessListOutput` process variable. Using the Process Console, you can then edit the variable and delete any items in the list that you don't want to terminate. After saving the variable, you can resume the process, which then iterates over the list of processes returned and terminates them. Here is an figure showing this process

Sample Java Client

Included in the SDK is a Java client sample that demonstrates how to return a list of active processes using a process filter. The source code can be found in the AdminSDK project `src-examples` package. After you import this project into Process Developer, you can run it as-is or modify the process filters to see how they impact the list of processes returned.

To run the application simply right-click on the `JAXWSClient.java` file and select the **Run As > Java Application** command from the menu.

Note: If you don't have the **Project | Build Automatically** option set, you will need to build this project before running it or after making changes to the source code.

```
////////////////////////////////////////////////////////////////////////////
// Copyright 2009 Active Endpoints, Inc
//
// Licensed under the terms of the Active Endpoints, Inc License (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
// http://www.activevos.com/activevos-evaluation-eula.php
//
```

```
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//////////////////////////////////////////////////////////////////////////
import javax.xml.ws.WebServiceRef;
import javax.xml.namespace.QName;
import example.*;
public class JAXWSClient {
 @WebServiceRef(wsdlLocation="http://localhost:8080/active-bpel/services/
           ActiveBpelAdmin?wsdl")
 static ActiveBpelAdmin service = new ActiveBpelAdmin();
 public static void main(String[] args) {
   try {
     JAXWSClient client = new JAXWSClient();
     client.doTest(args);
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
 public void doTest(String[] args) {
   try {
    System.out.println("Retrieving the port from the following service: " + service);

     IAeAxisActiveBpelAdmin port = service.getActiveBpelAdmin();
     System.out.println("Invoking the getProcessList() operation on the port.");
     AesProcessFilterType aesPFT = new AesProcessFilterType();
     AesProcessFilter aesPF = new AesProcessFilter();

     // examples for setting process filters here:
     // aesPF.setProcessState(1);
     // QName processName = new QName("TerminateRunningProcesses",
     //              "TerminateRunningProcesses");
     // aesPF.setProcessName(processName);
     aesPF.setAdvancedQuery("getProcessProperty(\"State\")='3'");
     System.out.println("aesPF: " + aesPF.getAdvancedQuery());

     aesPFT.setFilter(aesPF);
     AesProcessListType response = port.getProcessList(aesPFT);
     System.out.println("Total Rows: " +
              response.getResponse().getTotalRowCount() + "\n");

     java.util.Iterator <AesProcessInstanceDetail> it =
         response.getResponse().getRowDetails().getItem().iterator();
     AesProcessInstanceDetail currentRow;
     while (it.hasNext()) {
        currentRow = it.next();
        System.out.println("Process ID: " + currentRow.getProcessId());
        System.out.println("Process Name: " + currentRow.getName());
        System.out.println("Process State: " + currentRow.getState());
        System.out.println("Date Started: " + currentRow.getStarted());
        System.out.println("Date Ended: " + currentRow.getEnded());
        System.out.println();
     }

   } catch(Exception e) {
      e.printStackTrace();
   }
 }
}
```

CHAPTER 3

Identity Service API

This chapter includes the following topics:

What is the Identity Service API

Process Server provides an identity service to enable processes to look up users and groups in an enterprise directory service. This document describes the identity service API and includes a few examples that show the API being used with SOAP and indirectly using Java code. This second example uses the JAX-WS (Process Server IdentityService4J API.

Package Contents

The Identity Service API using JAX-WS (IdentityService4J) and the samples code project contains the following:

`dist/`	The Process Server `identityservice4j` and `wsht4j` API jars
`docs/`	Documentation
`lib/`	Dependencies and third party libraries; also, see the `/common/lib` root directory
`src/`	JAX-WS generated code
`src-examples/`	Sample Java code using the API
`build.xml`	Ant build script; use the latest supported JDK version to build this project
`.project .classpath`	Eclipse IDE project and classpath files

Identity Service Endpoints

The following table shows the Identity Service-related web service endpoints. You will need to change the `hostname` (for example, `localhos`) to reflect the actual host name where the service is hosted.

WSDL	`http://localhost:8080/active-bpel/services/AeIdentityService?wsdl`
Service SOAP Binding	`http://localhost:8080/active-bpel/services/AeIdentityService`

Setup

The Process Server Identity Service is normally available only to processes executing inside the Process Server using the identity service invoke handler.

In order to access the service operations from outside a process, the internal service operations must be exposed to an endpoint. Do this by deploying the `aeidentitysvc` system process to the server, as follows:

1. From the Process Console, click on the **Deploy** button located on the top right.
2. Browse and select the BPR located at `/orchestration/IdentityService/aeidentitysvc.bpr`. This BPR contains the `aeidentitysvc` process.

Note: If your server was configured for secured access, you must provide credentials for the `abIdentityListConsumer` role by creating a `<security-constraint>` definition in the `active-bpel.war` XML file:

```
<security-constraint>
   <web-resource-collection>
     <web-resource-name>Identity List Consumer</web-resource-name>
     <description>Endpoint that exposes an operation on the identity service.</
description>
     <url-pattern>/services/AeIdentityService</url-pattern>
     <http-method>POST</http-method>
     <http-method>PUT</http-method>
     <http-method>DELETE</http-method>
   </web-resource-collection>
   <!--IDENTITYLISTCONSUMER Uncomment to restrict access to the task client services
   <auth-constraint>
      <role-name>abIdentityListConsumer</role-name>
   </auth-constraint>
   IDENTITYLISTCONSUMER-->
</security-constraint>
```

Interacting with the Identity Service API

The Identity Service exposes the operations shown in the following table. You can use SOAP-UI or similar tool (such as the Eclipse's Web Services Explorer) to interact with the Identity Service.

Operation	Description
`findRoles`	Returns a list of role/group names.
`findRolesByPrincipal`	Returns a list of role/group names for a principal name.
`findIdentitiesByRole`	Returns a list of identities associated with role.
`findIdentities`	Returns a list of identities given an identity search query.

Operation	Description
assertPrincipalInQueryResult	Asserts that the a principal is in the resultant identity query. This faults if the principal is not in the result.
assertPrincipalInQueryResultWithResponse	Asserts that a principal is in the resultant identity query and returns the result. This faults if the principal is not in the result.
countIdentities	Returns the total number of identities found in the result after evaluating one or more identity search queries.

findRoles

This operation returns a list of roles available to the Identity Service. A sample `findRoles` SOAP request is similar to:

```
<soapenv:Envelope
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:iden="http://docs.active-endpoints/wsdl/identity/2007/
               03/identity.wsdl">
   <soapenv:Header/>
      <soapenv:Body>
         <iden:emptyElement/>
      </soapenv:Body>
</soapenv:Envelope>
```

The response contains one or more role names, as is shown in the following sample:

```
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <aeidsvc:roleList
              xmlns:aeidsvc="http://docs.active-endpoints/wsdl/identity/2007/
                           03/identity.wsdl">
         <aeidsvc:role>loanreps</aeidsvc:role>
         <aeidsvc:role>loanmgrs</aeidsvc:role>
         <aeidsvc:role>loancsr</aeidsvc:role>
      </aeidsvc:roleList>
   </soapenv:Body>
</soapenv:Envelope>
```

findRolesByPrincipal

This operation returns a list of roles (groups) that a principal belongs to. Here is an example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
           xmlns:iden="http://docs.active-endpoints/wsdl/identity/
                    2007/03/identity.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
        <iden:principalName>loanrep1</iden:principalName>
   </soapenv:Body>
</soapenv:Envelope>
```

Here is a `findRolesByPrincipal()` response:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
```

```
      <aeidsvc:roleList
           xmlns:aeidsvc="http://docs.active-endpoints/wsdl/identity/
                          2007/03/identity.wsdl">
         <!-- one or more roles -->
         <aeidsvc:role>loanreps</aeidsvc:role>
      </aeidsvc:roleList>
   </soapenv:Body>
</soapenv:Envelope>
```

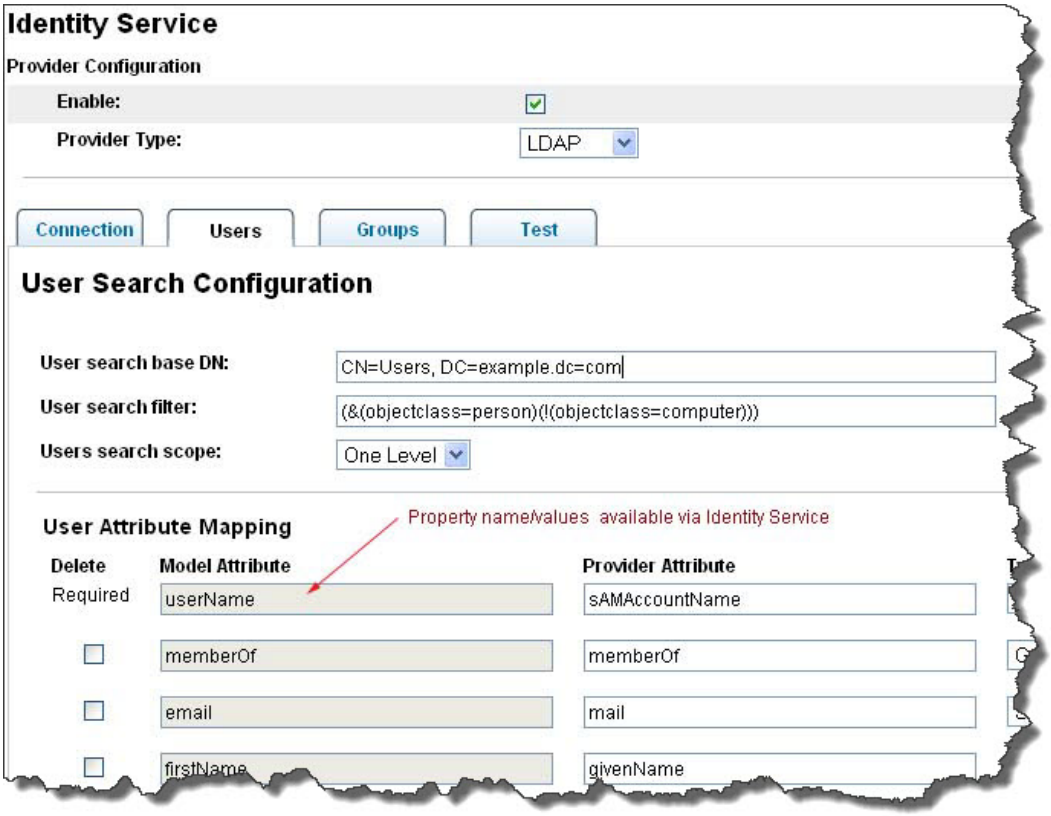
findIdentitiesByRole

The `findIdentitiesByRole()` operation return a list of identities associated with a role. Here is an example:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:iden="http://docs.active-endpoints/wsdl/identity/2007/03/identity.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
      <iden:roleName>loanreps</iden:roleName>
   </soapenv:Body>
</soapenv:Envelope>

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <aeidsvc:identityList
           xmlns:aeidsvc="http://docs.active-endpoints/wsdl/identity/2007/03/
identity.wsdl">
         <aeid:identity
              xmlns:aeid="http://schemas.active-endpoints.com/identity/2007/01/
identity.xsd">
            <aeid:id>CN=John Smith,CN=Users,DC=example,DC=com</aeid:id>
            <aeid:properties>
               <aeid:property
                  name="http://www.activebpel.org/ontologies/higgins/2008/
                        identity-search.owl#firstName">John</aeid:property>
               <aeid:property
                  name="http://www.activebpel.org/ontologies/higgins/2008/
                        identity-search.owl#lastName">Smith</aeid:property>
               <aeid:property
                  name="http://www.activebpel.org/ontologies/higgins/2008/
                        identity-search.owl#email">john.smith@example.com</aeid:property>
               <aeid:property
                  name="http://www.activebpel.org/ontologies/higgins/2008/
                       identity-search.owl#userName">jsmith</
aeid:property>
            </aeid:properties>
            <aeid:roles>
               <aeid:role>loanreps</aeid:role>
               <!-- Additional 'aeid:role' elements -->
            </aeid:roles>
         </aeid:identity>
         <!-- Additional 'aeid:identity' elements -->
      </aeidsvc:identityList >
   </soapenv:Body>
</soapenv:Envelope>
```

Notice the `<aeid:identity>` element in the result set. Within this element, the `<aeid:id>` represents the distinguished name (DN) for an LDAP based identity service providers. The `<aeid:identity>` also has one or more`<aeid:property>`elements, representing the identity service user model attributes.

For example, in the following figure, the LDAP (Active Directory) `givenName` attribute is mapped to `firstName` (in the ActiveVOS domain). This means, the LDAP `givenName` attribute is available using the `http://www.activebpel.org/ontologies/higgins/2008/identity-search.owl#firstName` property name.



To extract the email address from the result (say assigned to variable `$identityList`), use the following expression:

```
$identityList/aeid:identity/aeid:properties/aeid:property
     [@name='http://www.activebpel.org/ontologies/higgins/2008/
         identity-search.owl#email']/text()
```

findIdentities

This operation returns a list of identities based on an identity query. An identity query (`<iden:identityQuery>`) has include and exclude elements. The roles or principals listed within the include element are included in the result set while the ones listed within the exclude element are excluded.

Here is a sample request that constructs a query that fetches all loanreps members except for loanrep2 and also includes user loancsr:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:iden="http://docs.active-endpoints/wsdl/identity/2007/03/identity.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
      <iden:identityQuery>
         <iden:include>
            <iden:group>loanreps</iden:group>
            <iden:user>loancsr</iden:user>
         </iden:include>
         <!-- Additional iden:include elements. -->
         <iden:exclude>
            <iden:user>loanrep2</iden:user>
         </iden:exclude>
         <!-- Additional iden:exclude elements. -->
      </iden:identityQuery>
   </soapenv:Body>
</soapenv:Envelope>
```

The response contains zero or more `<aeid:identity>` elements.:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <aeid:identityResultSet xmlns:aeid="http://schemas.active-endpoints.com/
                identity/2007/01/identity.xsd">
         <aeid:totalRowCount>3</aeid:totalRowCount>
         <aeid:completeRowCount>true</aeid:completeRowCount>
         <aeid:identities>
            <aeid:identity>
               <!-- aeid:identity child elements -->
            </aeid:identity>
            <!-- Additional aeid:identity elements -->
         </aeid:identities>
      </aeid:identityResultSet
   </soapenv:Body>
</soapenv:Envelope>
```

assertPrincipalInQueryResult

The `assertPrincipalInQueryResult` operation checks to see if the a principal exists for an identity query. Restated, it checks if the principal exists in the result set of the evaluating the identity query. This operation faults if the principal is not in the result set.

Here is a sample request to check if user user1 is a member of either the loanreps or loanmgrs groups:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:iden="http://docs.active-endpoints/wsdl/identity/2007/03/identity.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
      <iden:principalQueryAssertion>
         <iden:principalName>user1</iden:principalName>
         <iden:identityQuery>
            <iden:include>
               <iden:group>loanreps</iden:group>
               <iden:group>loanmgrs</iden:group>
            </iden:include>
         </iden:identityQuery>
         <!-- Additional iden:identityQuery elements. -->
      <iden:principalQueryAssertion>
   </soapenv:Body>
</soapenv:Envelope>
```

If `assertPrincipalInQueryResult()` operation faults, the response is similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <Fault xmlns="http://schemas.xmlsoap.org/soap/envelope/">
         <faultcode xmlns:ns1="http://docs.active-endpoints/wsdl/identity/
                   2007/03/identity.wsdl" xmlns="">ns1:searchFault</faultcode>
         <faultstring xmlns=""/>
         <faultactor xmlns=""/>
         <detail xmlns="">
            <aeidsvc:identityFault xmlns:aeidsvc="http://docs.active-endpoints/
                        wsdl/identity/2007/03/identity.wsdl">
               <aeidsvc:code>10</aeidsvc:code>
               <aeidsvc:message>Principal user1 was not found in query.</aeidsvc:message>
            </aeidsvc:identityFault>
         </detail>
      </Fault>
   </soapenv:Body>
</soapenv:Envelope>
```

assertPrincipalInQueryResultWithResponse

This is similar to the previous operation; that is, it asserts that a principal is returned as part of one of the identity queries. All of the identity queries are evaluated and the response indicates which results contained the principal.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:iden="http://docs.active-endpoints/wsdl/identity/2007/03/identity.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
     <iden:principalQueryAssertionWithResponse>
         <iden:principalName>user1</iden:principalName>
         <iden:identityQuery>
            <iden:include>
               <iden:group>loanreps</iden:group>
               <iden:group>loanmgrs</iden:group>
            </iden:include>
         </iden:identityQuery>
         <!-- Additional iden:identityQuery elements. -->
      <iden:principalQueryAssertionWithResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

The response contains a 1-based index to the first identity query that contained the principal. For example, if the request element (`<iden:principalQueryAssertionWithResponse>`) has three identity queries (`<iden:identityQuery>` elements), and the principal was found in the second, the response looks similar to:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
        <aeidsvc:assertionQueryResponse
            xmlns:aeidsvc="http://docs.active-endpoints/wsdl/identity/2007/
                        03/identity.wsdl">2</aeidsvc:assertionQueryResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

countIdentities

This operation returns the total number of identities matched after evaluating all of the identity queries. Here is a sample request and response:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
         xmlns:iden="http://docs.active-endpoints/wsdl/identity/
                     2007/03/identity.wsdl">
   <soapenv:Header/>
      <soapenv:Body>
         <iden:principalQueryAssertionWithResponse>
            <iden:identityQuery>
               <iden:include>
                  <iden:group>loanreps</iden:group>
                  <iden:group>loanmgrs</iden:group>
               </iden:include>
            </iden:identityQuery>
            <!-- Additional iden:identityQuery elements. -->
         <iden:principalQueryAssertionWithResponse>
      </soapenv:Body>
</soapenv:Envelope>

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <aeid:identitiesCount
         xmlns:aeid="http://docs.active-endpoints/wsdl/identity/2007/03/
                     identity.wsdl">6</aeid:identitiesCount>
   </soapenv:Body>
</soapenv:Envelope>
```

Using JAX-WS Based ActiveVOS IdentityService4J API

The Process Server IdentityService4J API is a client based on JAX-WS (version 2.1.5). To use the client API, include the `avos-identityservice4j.jar` along with the dependent libraries for JAX-WS in your classpath. The JAX-WS dependent jars are:

`activation.jar`	Java Activation Framework
`jaxb-api.jar`	JAX Binding
`jaxb-impl.jar`	
`jaxws-api.jar`	JAX-WS
`jaxws-rt.jar`	
`jsr173_api.jar`	Streaming API for XML
`jsr181-api.jar`	Web Services meta data
`resolver.jar`	
`woodstox.jar`	StAX-compliant (JSR-173) Open Source XML-processor

Building IdentityService4J

You can build the API using the provided ant build script. The `avos-identityservice4j.jar` that it creates is copied to the `dist/` directory. Use the latest supported JDK version to build this project.

IdentityService4J Samples

Samples using the IdentityService4J API can be found under the `/src-examples/` source tree.

Listing Role Names Using IdentityService4J API

The following sample snippet shows how to list roles using the identity service using the IdentityService4J API. The `com.activevos.examples.identityservice.FindAllRolesDemo` class file has the complete example.

```
import com.activevos.api.identityservice.AeIdentityService;
import com.activevos.api.identityservice.IdentitySearchPortType;
import com.activevos.api.identityservice.SearchFault;
import com.activevos.api.identityservice.wsdl.EmptyElement;
import com.activevos.api.identityservice.wsdl.TRoleList;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import javax.xml.namespace.QName;

/** Identity Service endpoint WSDL  */
String ID_SERVICE_WSDL = "http://localhost:8080/active-bpel/services/AeIdentityService?
wsdl";

/** Identity Service endpoint QName  */
QName  ID_SERVICE_QNAME = new QName("http://docs.active-endpoints/wsdl/identity/2007/03/
identity.wsdl",
                              "AeIdentityService");
```

```
try
{
   // Create service using WSDL location and service qname.
   AeIdentityService idService = new AeIdentityService(
                           new URL(ID_SERVICE_WSDL), ID_SERVICE_QNAME);
   // Get service port to invoke operations.
   IdentitySearchPortType idServicePort =
                           idService.getAeIdentityServicePort();

   // Invoke findRoles() operation.
   TRoleList roleList = idServicePort.findRoles(new EmptyElement() );
   List roleNames =  roleList.getRole();
   System.out.println("Found role count: " + roleNames.size());
   for (String roleName : roleNames)
   {
      System.out.println("Role name: " + roleName);
   }
}
catch(MalformedURLException mue)
{
   // Invalid service url format
   mue.printStackTrace();
}
catch(SearchFault searchFault)
{
   // identity service related fault.
   System.out.println("Fault Message: " + searchFault.getMessage());
   if (searchFault.getFaultInfo() != null)
   {
      // Fault details.
      System.out.println("FaultInfo Code: " +
                           searchFault.getFaultInfo().getCode());
      System.out.println("FaultInfo Message: " +
                           searchFault.getFaultInfo().getMessage());
   }
   searchFault.printStackTrace();
}
catch(Exception e)
{
   e.printStackTrace();
}
```

Listing Identities Using the IdentityService4J API

This sample snippet illustrates using the `findIdentitiesByRole()` operation. The `com.activevos.examples.identityservice.FindIdentitiesByRoleDemo` class file has the complete example.

```
import com.activevos.api.identityservice.AeIdentityService;
import com.activevos.api.identityservice.IdentitySearchPortType;
import com.activevos.api.identityservice.SearchFault;
import com.activevos.api.identityservice.wsdl.TIdentityList;
import com.activevos.api.identityservice.xsd.TIdentity;
import com.activevos.api.identityservice.xsd.TProperty;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;
import javax.xml.namespace.QName;

/** Identity Service endpoint WSDL  */
String ID_SERVICE_WSDL =
          "http://localhost:8080/active-bpel/services/AeIdentityService?wsdl";

/** Identity Service endpoint QName  */
QName  ID_SERVICE_QNAME = new QName(
       "http://docs.active-endpoints/wsdl/identity/2007/03/identity.wsdl",
       "AeIdentityService");
try
```

```
{
   // Create service using WSDL location and service qname.
   AeIdentityService idService = new AeIdentityService(new URL(ID_SERVICE_WSDL),
        ID_SERVICE_QNAME);
   // Get service port to invoke operations.
   IdentitySearchPortType idServicePort = idService.getAeIdentityServicePort();
   // Invoke findIdentitiesByRole() operation.
   // In this example, get the list of identities for role 'loanreps'.
   TIdentityList idList = idServicePort.findIdentitiesByRole("loanreps");

   // get list of Identities from resultset
   List<TIdentity> identities = idList.getIdentity();
   System.out.println("Found identities count: " + identities.size());
   for (TIdentity identity : identities)
   {
      System.out.println("ID: " + identity.getId()); // id, such as DN of LDAP entry.
      // List of roles this person  belongs to.
      List<String> memberRoles = identity.getRoles().getRole();
      for (String role : memberRoles)
      {
         System.out.println("Member Of : " + role);
      }
      // List all attributes. E.g. firstName, lastName, email etc.
      List<TProperty> properties = identity.getProperties().getProperty();
      for (TProperty  prop : properties)
      {
         System.out.println("property[" + prop.getName() + "] = " +
                  prop.getValue() );
      }
      System.out.println();
   }

}
catch(MalformedURLException mue)
{
   // Invalid service url format
   mue.printStackTrace();
}
catch(SearchFault searchFault)
{
   // identity service related fault.
   System.out.println("Fault Message: " + searchFault.getMessage());
   if (searchFault.getFaultInfo() != null)
   {
      // Fault details.
      System.out.println("FaultInfo Code: " +
                  searchFault.getFaultInfo().getCode());
      System.out.println("FaultInfo Message: " +
                  searchFault.getFaultInfo().getMessage());
   }
   searchFault.printStackTrace();
}
catch(Exception e)
{
   e.printStackTrace();
}
```

Constructing an Identity Query Using the IdentityService4J API

This sample snippet shows how to construct an IdentityQuery, which is used in `findIdentities()`, `assertPrincipalInQueryResult()`, `assertPrincipalInQueryResultWithResponse()`, and `countIdentities()` operations.

The `com.activevos.examples.identityservice.FindIdentitiesDemo` class file has the complete example.

```
// imports similar to previous examples
try
{
   // Create service and get service port 'idServicePort' similar to previous examples.
```

```
   // ...
   // ...

   // Create identity query with includes and excludes:
   // 1) include groups : loanreps,  user: loancsr
   TIdentityQueryValues includeQueryValues = new TIdentityQueryValues();
   includeQueryValues.getGroup().add("loanreps"); // group
   includeQueryValues.getUser().add("loancsr"); // user

   // 2) exclude groups : loanmgrs
   TIdentityQueryValues excludeQueryValues = new TIdentityQueryValues();
   excludeQueryValues.getGroup().add("loanmgrs"); // group

   // Create query with includes and excludes
   TIdentityQuery query = new TIdentityQuery();
   query.setInclude(includeQueryValues);
   query.setExclude(excludeQueryValues);

   // Invoke findIdentites() operation.
   TIdentityResultSet resultSet = idServicePort.findIdentities(query);
   System.out.println("Query match count : " + resultSet.getTotalRowCount());
   // get list of Identities from resultset
   List identities = resultSet.getIdentities().getIdentity();
   System.out.println("Found identities count: " + identities.size());
   for (TIdentity identity : identities)
   {
      // print details, similar to previous example
   }
}
catch(Exception e)
{
   e.printStackTrace();
}
```

CHAPTER 4

Screenflow Programming and SDK

This chapter includes the following topics:

About ActiveVOS Platform SDK

The Platform SDK contains a business user tool for workflow development called the Guide Designer. The guide designer has a simple interface for creating steps such as screens and automated actions and they can use and modify data in a host system. The guides that you create are applications that run from within a web browser connecting to either an internal or external server. The browser can be within a PC or a computer created by Apple.

The Host Provider SDK and associated Service Programming Interface (SPI) allow developers to expose data and operations in their host system to the Designer. At the heart of the system are reusable services that are defined using the powerful Process Developer environment and deployed to the Process Server environment. These services implement the SPI necessary for providing a hosting environment (Host Provider) and are described to the Process Servr using a simple descriptor mechanism

Sample Guide Designer Orchestration Project Templates

Project Developer has three sample Guide Designer orchestration project templates:

- **Guide Designer Demo**: This sample has the source for all demo guides and service all steps used in the demo environment. It also includes additional guides and service call steps. The emphasis here is interacting with databases. You can use the way this orchestration project is organized to help you get started designing your own applications. Also, after deploying the project, you can use the guides it installs to introduce your users to the way guides work without having them interact with your business applications.
- **Guide Designer Customization Starter**: This template can be used as a starting point for creating services and custom themes for use in Guide Designer.
- **Guide Designer Host Provider**: This template can be used as a starting point for customizing the sample host provider.

Each of these projects has its own cheat sheet that will guide you through using them.

Note: Do not deploy more than one of these projects at the same time. If you do, you will see an error dialog with the following message 'Application is not correctly configured' when you are using Process Central'. This occurs because each sample project has its own configuration file and only one can exist. Correct this problem by going to the Process Console, selecting **Catalog | Contribution,** clicking on a contribution's link, then deleting one of the projects.

Guide Designer Host Provider Template

The easiest way to get started is by installing the Guide Designer Host Provider sample project provided with the Process Developer and which runs sin the Process Designer environment. The Process Developer has an orchestration project template sample for creating a Host Provider. Simply select **File > New Project > Orchestration Project,** enter a project name, and then select **Guide Designer Host Provider Template** to set up the project in your workspace.

Sample Project Contents

The sample project contains resources that illustrate how to use an SQL database as the host application on which Process Designer will operate. All screens operate on one or more entities that are contained within a database table. Internally, this sample uses the Data Access system service provided by Process Server to connect to the database with a JDBC connection configured in JNDI.

BPEL Folder

The `bpel` folder contains the following resources:

- `dbHostEnvironmentRuntime.bpel`: A single entry point that Process Designer can use to interact with the entities on the host platform. The interface for this process defines five basic operations: Create, Read, Update, Delete (CRUD) and an advanced Query operation.
- `dbCreateAnyEntityService.bpel`: A simple conversion process that maps service call step requests for creating any entity to the `dbHostEnvironmentRuntime`.
- `dbDeleteAnyEntityService.bpel`: A simple conversion process that maps service call step requests for deleting any entity to the `dbHostEnvironmentRuntime`.

The deployment artifacts for each of these BPEL process are in the `deploy` directory.

Config Folder

The `config` folder contains resources typically found in many projects:

- `aeiHolidayCalendar.xml`: Names holidays. This is used by date data types.
- `app-config.xml`: Configuration file that specifies design time and run time properties for the host provider that are paired with Process Designer. This configuration defines features supported by the host platform as well as possible overrides to Process Designer.
- `custom-services.xml`: Details for custom service call steps that all guide designers can access.>
- `custom-types.avcconfig`: Process Central configuration file for deploying custom Guide Designer custom JavaScript renderers, i18n properties, and the request forms used in the sample that are embedded in the runtime in Process Central.
  You can override Process Designer messages by creating a new `messages.properties` file. After creating this file, make sure that the `avcconfig` file that refers to this `messages.properties` file comes later than built-in `avcconfig` file when they are sorted alphabetically. For example, you could update `custom-types.avcconfig` to include a line such as `<tns:i18n location="messages.properties"/>`.

  You can see what the order is by selecting **Catalog | Resources | Central Configs** from within the Process Console. If one of your files is inadvertently overriding another, you will need to change the file's name to change the execution order.
- `custom-types.xml` : Custom data types and their corresponding JavaScript renderers.

- `MapItRenderer.js`: A simple example of a JavaScript renderer that puts a link on a guide screen that links to a mapping service such as Google Maps.
- `Stage_picklist.xml`: The definition of the values to which a variable can be set. These values are displayed as a picklist.

db_schema Database Artifacts Folder

The `db_schema` folder in the sample project provides the DDL for some common databases that creates the tables required by the guides packaged with the sample. The included Derby database is prepopulated with a schema and sample data.

Forms

This sample has two folders:

- `guide_explorer`: This folder contains the implementation for the guide explorer within the Forms pane of Process Central. This code, if you like, can be the basis for changing the way the explorer displays.
- `request`: This folder contains the implementation for the guide viewer. Similar to the way you can use files in the `guide_explorer` folder, use this file as the basis for changing the way in which the guide viewer displays.

Guides

Sample guides are in the `guides` folder. These are imported and published during when the sample is deployed.

host_env Environment Definition Folder

The `host_env` folder contains the definitions of all entities used by the host application. In this sample, each table in the database schema is mapped to a single entity in this folder. For example, the `contacts` table is represented by the `contacts_detail.xml` file. In addition, the `entityList.xml` file is a master index of all entities that will be deployed.

Other Resources

The `icons` folder has images that guide designers can use and which are used by published guides. Other folders containing data are `deploy`, `sample-data`, and `xquery`.

Sample Customization Starter Template

The Sample Customization Starter template can be used as a starting point for creating service call steps and custom themes for use in Guide Designer Simply select **File > New Project > Orchestration Project**, enter a project name, and then select **Guide Designer Customization Starter** to set up the project in your workspace. The following sections describe the contents of this project.

BPEL Processes

The `bpel` folder contains the `HelloWorld.bpel` file that is based on the `HelloWorldAutomatedStep` system service. The service receives a name and returns a Hello World-type of greeting based on the time of day. The process demonstrates the use of a complex-type response and shows a complex type being rendered for display in HTML and uses JavaScript and related technologies for this purpose. You must always create a custom renderer for your custom types.

Deployment artifacts for these BPEL process are in the `deploy` directory.

Guides

This folder contains the definition for the "Hello World" guide. It also contains the system-generated `guide-catalog.xml` file.

A guide is the application that users build in Guide Designer. (Application developers can also create them within Process Developer and Process Designer.) The guide within this folder has a few step types, including a service call step. This step is essentially the `HelloWorld.bpel` process. All of your application's guides will reside in the `guides` folder.

Forms

The folders in the `forms` folder define two of the components that Process Central displays as part of its user interface: the guide explorer and the guide viewer. Use the code in the following folders to see how each was created. You can also use this code if you wish either to create your own explorer or viewer or modify the ones that exist..

- `guide_explorer`: This folder contains the implementation for the guide explorer within the Forms pane of Process Central. This code, if you like, can be the basis for changing the way the explorer displays.
- `request`: This folder contains the implementation for the guide viewer. Similar to the way you can use files in the `guide_explorer` folder, use this file as the basis for changing the way in which the guide viewer displays.

The `samples-starter.avcconfig.xml` contains information used when configuring form filters.

`avcconfig` files here and in other folders control how Process Designer resources are used in a project.

Note: The Process Server executes your `avcconfig` files in alphabetical order. You can see this order by selecting **Catalog | Resources | Central Configs** from within the Process Console. If one of your files is inadvertently overriding another, you will need to change the file's name to change the execution order.

Reports

The `reports` folder contains the files needed to construct the guides report that displays within Process Central. Use the information in this folder as the basis for creating your own reports. (This report, like other reports used in Process Central, was created using BIRT--Business Intelligence and Reporting Tools.) The `avcconfig` file contains sections that are commented out, and you should uncomment them if you are creating your own versions.

Guide Designer

The `guides` folder contains the resources that the project requires, some of which are deployed in the project deployment contribution and others that you must import directly into Guide Designer. Open the `.xml` files within each of the subfolders to see what was done for this project.

This folder contains four additional folders as well as the `app-config.xml` file.

- `renderings`: The greeting rendering file defines how output data is displayed. It contains layout details and a script that implements the rendering interface. This file is deployed in the project deployment contribution.
- `resource-contribs`: If you wish to provide images to guide designers, you must package them into the contribution being deployed along with a screenflowContribution metadata file.
- `service-contribs`: You must add a `services.xml` file to the deployment package for a service call step. This file contains the configuration details that make the service call step type available and to describe how data is used. This file is deployed in the project deployment contribution.
- `theme-contribs`: When a guide runs, the default background colors, button styles, and other style elements are defined in HTML/CSS format in files deployed to the Process Server and made available to Guide Designer. You can create new themes and use them in a guide. A theme.xml descriptor is deployed as part of the project deployment contribution. This folder also contains other files used by the render. **Note**: If you create more than one Guide Designer Customization Starter project, do not deploy `themes.xml` more than once. Duplicate deployments create a duplicate theme name conflicts.

The folder also has the `app-config.xml` configuration file. This file has design-time and run-time properties for Guide Designer that declared to be used by the host provider This configuration defines features supported by the host platform as well as possible overrides to Guide Designer.

Note: There can be no more than one `app-config.xml` file on a server.

Permissions

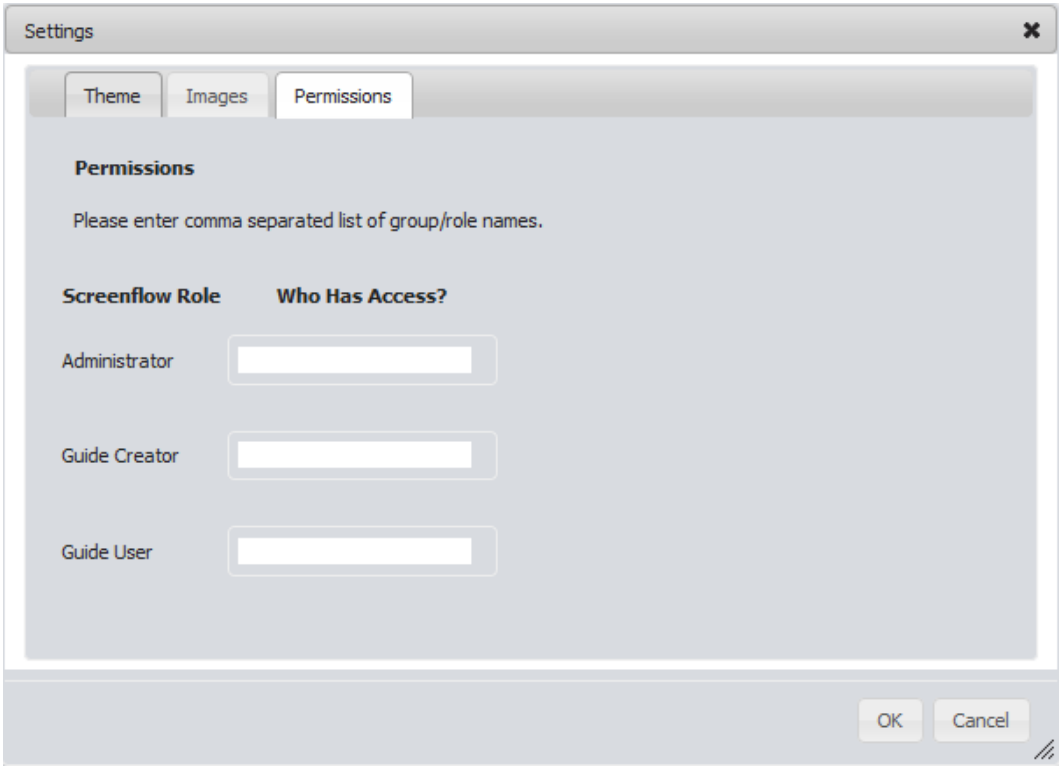
Permissions determine who can see guides when information within the Forms, Designs, and Report panes. (Reports are not available in Process Designer.) Each is set independently; that is, the permissions that you set for one pane do not affect the permissions in the other. An application configuration file sets permissions for the Designs pane and an `.avcconfig` file sets them for the Forms pane. For example, if you set permissions for access in the Designs pane, these settings do not affect what users can see in the Forms pane.

Setting Permissions in an Application Configuration file

When you create an application configuration file with a "Sample Configuration" on page 55 element, setting `globalPermissions` to `true` tells Guide Designer to display a Permissions tab within the guide designer's **Settings** dialog. Here is an example of setting the `globalPermissions` feature:

```
<screenflowConfig ...>
   <global ...>
      <features>
         <feature name="guidePermissions" enabled="true"/>
         ...
      </features>
      ...
```

Now that this is set, users will see the following dialog when displaying the **Settings** dialog from the guide designer's toolbar.



Any user who has access to Process Designer will then be able to make changes here that affect the privileges that a user has for guides.

The permissions allow the following access:

- **Administrator**: Has full access to guides; that is, an administrator can create and run guides, and can see and use this Permissions tab. Only administrators can see and use the **Terminate Running Guides** button that appears when a guide is selected in the upper-right panel displayed when the Designs pane is being displayed.
- **Guide Creator**: Can create and run guides; however, users in this role do not see the Permissions tab.
- **Guide User**: Can only run guides.

The settings made by users in this dialog do not affect permissions for accessing guides in the Forms pane.

Setting Permissions in an .avcconfig File

When you add an `.avcconfig` file that is associated with forms, you can set roles for who can access information. Here's an example that allows users in the three named roles to access forms.

```
<tns:requestCategoryDefs>
  <tns:requestCategoryDef id="requestcategory_guides_launchpad"
                          name="${requestcategory_guides_launchpad}">
     <avccom:requestDef id="hello-world" name="Guide In Form">
        <avccom:allowedRoles>
           <avccom:role>abAdmin</avccom:role>
           <avccom:role>abDeveloper</avccom:role>
           <avccom:role>loanreps</avccom:role>
       </avccom:allowedRoles>
     ...
```

The settings made by users in this dialog do not affect permissions for accessing guides in the Forms pane.

An application can have more than one `.avcconfig` file. For example, you may have a second one that defines permissions for information in the Reports pane.

The order in which the Process Server uses the information in `.avcconfig` files is alphabetical. This means that the elements in one file can overwrite elements in another. You can see this order by selecting **Catalog | Resources | Central Configs** from within the Process Console. If one of your files is inadvertently overriding another, you will need to change the file's name to change the execution order.

Deploying the project

This topic describes how you deploy the sample application.

Setting Up the Database (Guide Designer Host Provider Only)

The sample application uses the embedded Process Server database that ships with Process Developer. If you are using this database, it is set up for you. If you wish to use an external database, you will need to do the following to use the new JNDI data source:

1. Configure your Process Server with an additional JNDI data source.
2. Deploy the DDL provided with the sample.
3. Update the PDD for the `dbHostEnvironmentRuntime`.

Setting Up the Server

The steps for setting up the server are as follows:

1. Start the embedded Process Server. This is described in the Process Developer help.
2. Right click on the `.bprd` file (for example, `ae-db-cloudprovider.bprd`) and choose **Execute** from the context menu.
3. Update the URN mapping. Set `aeHostEnvironmentRuntime` to `dbHostEnvironmentRuntimeService`.

Using the Sample Host Provider

This topic describes running the guides within the sample projects.

Running the Sample Guides

Run the sample guides as follows:

1. Navigate to `http://localhost:8080/activevos-central` in your browser.
2. Log in as `loanrep1/loanrep1`.
3. Choose **Forms** from the left navigation pane.
4. Click on the `Published Guide Designers` folder.
5. Choose one of the sample guides.
6. After the guide loads, follow its prompts.

Mobile Guide Usage

To convert any guide to a mobile guide, begin by changing the theme to "Smartphone" in the **Guide Properties** dialog. The size difference between the screen and a phone could mean that you may have to rearrange and perhaps simplify steps.

Customizing the Sample Host Provider

This section discusses customizations that you can make to the sample project. They demonstrate how you define a new type that provides links to a mapping service such as Google Maps. The first step is defining a JavaScript renderer and defining a custom data type. (*Rendering* is the application of a friendly user interface for the task's interface input and output messages.) Next, modify the project's metadata files that allow these additions to be discovered.

Note: The Process Server executes your project's `.avcconfig` files in alphabetical order. You can see this order by selecting **Catalog | Resources | Central Configs** from within Process Console. If one of your files is inadvertently overriding another, you will need to change the file's name to change the execution order.

Note: When implementing a custom renderer, you must add the `ae-sf-renderer-container` class to the root element (that is, to the `<div>`) of your renderer. If you do not, your interface will open in a new window.

Custom Type Example

Creating a custom data type requires that the new type be defined in a metadata file and a renderer be defined to display it at both design- and runtime.

In the example, `{$sample.project}/config/MapItRenderer.js` renders an HTML anchor with the text "Map It" and the `href` of the anchor is a URL to a mapping service configured in the datatype options.

Code Example: Renderer JavaScript

```
(function($) {
   activevos.util.createAvosPackage("my.custom.namespace");
   function MyCustomRenderer(aDataValueCollection) {
      this.renderField = function(aContainerElement,
            ajQueryContext, aFieldName, aFieldDataValue) {
         // do something
         };
   };
   my.custom.namespace.MyCustomRenderer = MyCustomRenderer;
})(jQuery);
```

This example is a skeleton for a custom renderer and it only requires that the `renderField` function be implemented. The second line defines a unique package name or namespace for the renderer; this is similar to a package name in Java.

The lines shown in bold begin the definition of the custom renderer function. It accepts an argument (shown in the next example) that contains all of the data for the screen. It also defines the `renderField` property, which is also a function and which renders the data. The `renderField` arguments are the HTML elements that contain your custom rendered field, the jQuery context, the field name being rendered, and the field value (see the `AeScreenFlowDataValue` example later in this topic).

Code Example: AeScreenFlowDataValuesCollection Function

```
function AeScreenFlowDataValuesCollection(){
      /**
       * @returns {Array} list of AeScreenFlowDataValue objects.
       */
      this.getDataValues = function() {};
      /**
       * @returns {boolean} true if named data value holder
       * exists.
       */
      this.hasDataValue = function(aName) {};

      /**
       * Returns a type by name or null if not found.
       * @param {String}: aName name of data value
       * @returns {AeScreenFlowDataValue}; the data value or null
       *    if not found.
       */
      this.getDataValue = function(aName) {};
}
```

Code Example: AeScreenFlowDataValue Function

```
function AeScreenFlowDataValue(aType, aName, aValue, aRenderingTypeName) {
   /**
    * Gets the rendering type name. If the rendering type is not
    * set, it returns the type.
    */
   this.getRenderingTypeName = function() {};
   /**
    * Gets the name of the field.
    */
   this.getName = function() {};

   /**
    * Gets the value of the field.
    */
   this.getValue = function() {};

   /**
    * @returns {AeDataOptions} Collection of AeOption objects.
    */
   this.getOptions = function() {};

   /**
    * Returns option value.
    * /
   this.getOptionValue = function(aName) {};

   /**
    * @returns {boolean} true if field is readonly.
    */
   this.isReadOnly = function() {};
}
```

After you define the renderer, you can define the custom type. The `{$sample.project}/config/custom-types.xml` sample project defines a new type named `mapit`. It is bound to the `MapitRenderer.js` custom renderer. To enable this type for the sample application, uncomment the `type` element.

Using JavaScript Libraries

When a custom renderer uses a JavaScript library, it can import it dynamically instead of including it within an `avcconfig` file. For example, you could load a library named `dataTable` as follows:

```
var loadDataTablePlugin = $.dataTable ? null :
            myProject.loadScript(/path/to/datatable/js/file, $);
       $.when(loadDataTablePlugin).done( function() {
           // Use dataTable here}
       );
```

Customizing an Entity

After creating the custom types and renderers, you need to make them available for your host application. The project does this by using the `mapit` type in the custom `contacts` entity to provide a Google Map for the contact. In the `{$sample.project}/host_env/contacts_detail.xml` file, uncomment the field definition for `Full Address.`

Code Example: Full Address Field Definition

```
<host:field label="Full Address" name="FullAddress"
         readonly="true" required="false" type="mapit">
   <host:options>
      <host:option name="mapping-provider"
          hide="true">http://www.bing.com/maps/default.aspx?q=</host:option>
   </host:options>
...
```

The `mapping-provider` attribute's value was changed to use Microsoft Bing as the mapping provider instead of Google Maps. The `mapping-provider` option is hidden so that guide developers cannot edit it. If the `hide` attribute was set to false or removed, guide developers could change the `mapping-provider` in the guide designer.

After altering the entity definition, redeploy the sample project using the packaged BPRD file, as follows:

- Navigate to the guide designer and open the Sales Call Follow up guide.
- Open the Click continue after your verification step and add a read-only field for Full Address.
- Save and publish the guide.

After you launch the updated guide, a new **Map It** link appears that opens a new browser tab when clicked.

Validating Fields

A render class can implement the `validateField` JavaScript function to validate data. This function is called when the user presses an action button.

This function's definition is:

```
validateField(aFieldDataValue, aJQueryContext)
```

where:

- `aFieldDataValue`: An `AeScreenFlowDataValue` instance.
- `aJQueryContext`: A jQuery object to be used as context for validation. Any jQuery plugins imported through `.avcconfig` are available here.

Here's an example:

```
 this.validateField = function(aFieldDataValue, aJQueryContext) {
    var $ = aJQueryContext;
    var salutation = aFieldDataValue.getValue().salutation;
    if ( salutation.name.$t === 'chester'){
         var errorMsg = "Cant equal chester";
         return { result: false, message: errorMsg};
        }
    return { result: true };
}
```

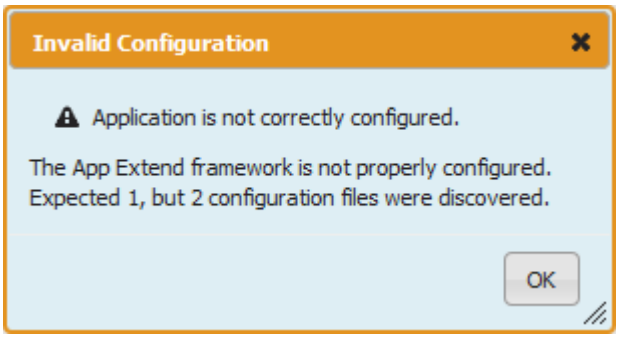
Service Programmer Interface

Guide Designer is configured by XML files that are deployed as part of a contribution to the Process Server catalog. These metadata files describe the application's custom components; for example, objects, service all steps, data types, services, and so on. These components are available to guide designers in the editor environment. The files are discovered by searching the Process Server catalog using the document's XML namespace.

Application Configuration

This Guide Designer configuration metadata is the entry point for the guide editor and runtime components. Configuring Guide Designer for your environment requires that you deploy this metadata file, which will later be discovered when a guide editor is opened or a guide begins execution.

Note: If you deploy a project more than once and each has its own `app-config.xml` file, your users will see the following error message:



If this error occurs, you will need to go to the Process Console (usually running at `localhost:8080/activevos`), select **Catalog | Contributions**, click on a contribution's link, then select the **Offline** button.

Sample Configuration

The following is a sample configuration file:

```
<screenflowConfig xmlns="http://schemas.activevos.com/
         appmodules/screenflow/2012/09/
         avosScreenflowConfiguration.xsd">
   <global hostRuntimeService="runtimeService"
        currentUserEntity="CurrentUserEntity"
        entityIndex="project:/app/entityList.xml"
        applicationName="Custom Application">
      <features>
         <feature name="polymorphism" enabled="false"/>
         <feature name="runAsUser" enabled="false"/>
      </features>
      <helpLink>./help.html</helpLink>
   </global>
   <designer>
      <helpLink>../editor/help.html</helpLink>
   </designer>
</screenflowConfig>
```

In this sample, notice the following:

- The `global` element that has configuration properties common to the editor and run-time environments.
- The `features` element either enables or disables features supported by Guide Designer that may or may not be supported by your host provider.

- The optional `helpLink` attribute is a text node whose value should be a URI that points to a help document. The URI may be relative, an absolute HTTP, or a `project:/` path.

Global Properties

The follow table describes the properties of the global element. These are configured as attributes.

Property	Required	Description
`applicationName`	No	Parameter used for debugging purposes, and whose default value is "Socrates".
`currentUserEntity`	No	If your host supports an entity to provide additional details about the currently logged in user, configure this field to provide the object's name. This name should exist in the entity index.
`entityIndex`	No	Either a `project:/` path to an entity index resource or a service name of a BPEL process that can create an entity index and get entity details for an entity by name. The default value is `avHostEnvironmentService`, which is a Process Designer-provided process to discover entities deployed to the Process Console catalog.
`helpLink`	No	(optional) A text node whose value should be a URI that points to a help document. The URI may be relative, an absolute HTTP, or `project:/` path
`hostRuntimeService`	Yes	The name of the service deployed for the BPEL process that makes, creates, reads, updates, and queries requests to the host application.

Features

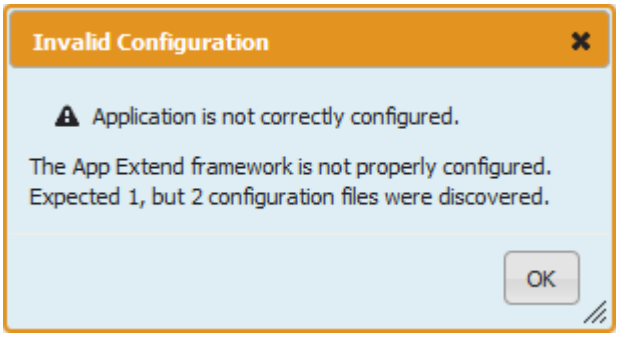
The following features are currently supported by Guide Designer. However, your host run-time may not support them.

Feature	Description
`guidePermissions`	Guides within a project can be restricted by role or group using a tab displayed in the **Settings** dialog invoked from the Guide toolbar on the Home page. When this property is set to true, a Permissions tab appears within this dialog.

`handleRedirectOnDone`	Guides can redirect to another URL when they are done, and the redirect `onDone` feature allows the hosting application to override the default redirect behavior with its own. The default behavior is to navigate the current window to the URL defined by the guide. When this feature is enabled, Guide Designer will no longer do the redirect; instead it will execute an `aesf-done` window event, which your application can listen for and act upon.
	The event data will be a string having the following format:
	`aesf-done | [reference: | url:] data`
	Examples are:
	`aesf-done|url:http://active-endpoints.comaesf-done|url:active-endpoints.com aesf-done|reference:50c7f76e-0f86-49d1-8e1d-ee9478f9b54d`
	The sample project contains an example that shows listening for window events in the `guideExplorer.js` file in the `subscribeToAeSfDoneMessages` function.

Feature	Description
`polymorphism`	If enabled, the user can choose more than one object as a "reference to". If set to true, service call steps and the data service may receive space-delimited object names as a `referenceTo` value. Only set this to true if your host supports polymorphic relationships.
`runAsUser`	If your host supports data access and modification requests at an elevated user, enable this feature. By default, all requests to service call steps and the data service include a `runAsUser` element set to `$current`. When this is enabled, the guide developer sees an option for setting the `runAsUser` value to `$system`, indicating that a guide can create or modify objects in the host that they may not normally have the permissions for.
`redirectToObject`	This feature affects End steps and determines the behaviors guide developers can set when a guide is done. If disabled, the only *on done* behavior is the ability to go to a URL. However, if your host supports URLs for the object in the host and if this feature is enabled, guide developers can choose: - Refresh Current Object - Go to Other Object - URL
`showEmbedLink`	An HTTP link for embedding the guide is displayed.

Note: If you deploy a project more than once and each has its own `app-config.xml` file, your users will see the following error message:



If this occurs, you will need to go to the Process Console (usually running at `localhost:8080/activevos`), select **Catalog** | **Contributions**, click on a contribution's link, then select the **Offline** button.

Permissions

Permissions determine who can see guides when information within the Forms, Designs, and Report panes. (Reports are not available in Process Designer.) Each is set independently; that is, the permissions that you set for one pane do not affect the permissions in the other. An application configuration file sets permissions for the Designs pane and an `.avcconfig` file sets them for the Forms pane. For example, if you set permissions for access in the Designs pane, these settings do not affect what users can see in the Forms pane.

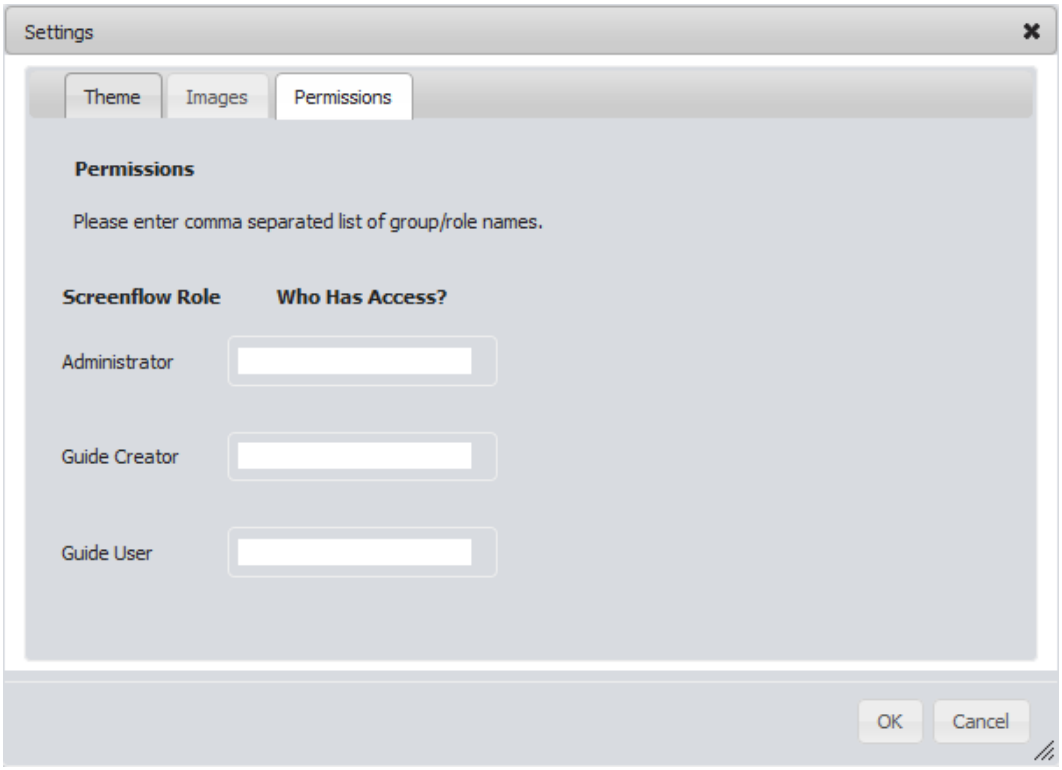
Setting Permissions in an Application Configuration file

When you create an application configuration file with a "Sample Configuration" on page 55 element, setting `globalPermissions` to `true` tells Guide Designer to display a Permissions tab within the guide designer's **Settings** dialog. Here is an example of setting the `globalPermissions` feature:

```
<screenflowConfig ...>
   <global ...>
```

```
<features>
   <feature name="guidePermissions" enabled="true"/>
   ...
</features>
...
```

Now that this is set, users will see the following dialog when displaying the **Settings** dialog from the guide designer's toolbar.



Any user who has access to Process Designer will then be able to make changes here that affect the privileges that a user has for guides.

The permissions allow the following access:

- **Administrator**: Has full access to guides; that is, an administrator can create and run guides, and can see and use this Permissions tab. Only administrators can see and use the **Terminate Running Guides** button that appears when a guide is selected in the upper-right panel displayed when the Designs pane is being displayed.
- **Guide Creator**: Can create and run guides; however, users in this role do not see the Permissions tab.
- **Guide User**: Can only run guides.

The settings made by users in this dialog do not affect permissions for accessing guides in the Forms pane.

Setting Permissions in an .avcconfig File

When you add an `.avcconfig` file that is associated with forms, you can set roles for who can access information. Here's an example that allows users in the three named roles to access forms.

```
<tns:requestCategoryDefs>
  <tns:requestCategoryDef id="requestcategory_guides_launchpad"
                          name="${requestcategory_guides_launchpad}">
     <avccom:requestDef id="hello-world" name="Guide In Form">
        <avccom:allowedRoles>
           <avccom:role>abAdmin</avccom:role>
           <avccom:role>abDeveloper</avccom:role>
```

```
            <avccom:role>loanreps</avccom:role>
          </avccom:allowedRoles>
        ...
```

The settings made by users in this dialog do not affect permissions for accessing guides in the Forms pane.

An application can have more than one `.avcconfig` file. For example, you may have a second one that defines permissions for information in the Reports pane.

The order in which the Process Server uses the information in `.avcconfig` files is alphabetical. This means that the elements in one file can overwrite elements in another. You can see this order by selecting **Catalog | Resources | Central Configs** from within the Process Console. If one of your files is inadvertently overriding another, you will need to change the file's name to change the execution order.

Designer Properties

The designer element allows for configuration of designer-specific properties. Currently, the only child element that you can use is `helpLink`. The text node value of `helpLink` should be a URI that points to a help document. The URI may be relative, an absolute HTTP, or `project:/` path.

Object Descriptions

The objects in your host application must be described to Guide Designer in order for their fields to be rendered. Each object is described in an `entityDetail` document as described by the XML schema provided by the Process Designer SDK. There can be an `entityList` document that acts as an index for each object you describe.

If you do not deploy an `entityList`, and use the default host environment service, an `entityList` document is generated dynamically at runtime by discovering all deployed `entityDetail` documents in the catalog. If you override the `entityIndex` global configuration property with a `project:` value, you must use an `entityList`.

An object detail contains information about:

- The name of the object in the host
- Labels used to present the object to guide creators
- The object's fields
- Any relationships between the object and other objects on the host

The objects and entities named in this paragraph are highlighted in bold in the code example that follows.

The sample account object (which is the next example) defines an object named `accounts` which is displayed to guide designers as `Account` or `Accounts`, and has a primary key field called `AccountId`. In addition, the account object has two additional fields besides the primary key called `Name` and `AccountNumber`. Finally, this object has a relationship with another object named `contacts`.

Code Example: Sample Account Object

```
<host:entityDetail
   xmlns:host=
      "http://schemas.active-endpoints.com/appmodules/
          screenflow/2011/06/avosHostEnvironment.xsd"
      xmlns:comm="http://schemas.active-endpoints.com/
          appmodules/screenflow/2011/07/avosCommon.xsd"
      name="accounts"label="Account"labelPlural="Accounts"
      objectIdField="AccountId">
   <host:detail>
      <host:childRelationship childSObject="contacts"
         field="AccountId" relationshipName="ctct_to_acct" />
      <host:field label="AccountId" name="AccountId"
```

```
            required="true" type="id">
         <host:options>
            <host:option name="length">36</host:option>
         </host:options>
      </host:field>
      <host:field label="Name" name="Name" required="false"
            type="string">
         <host:options>
            <host:option name="length">45</host:option>
         </host:options>
      </host:field>
      <host:field label="AccountNumber" name="AccountNumber"
            required="false" type="string">
         <host:options>
            <host:option name="length">10</host:option>
         </host:options>
      </host:field>
   </host:detail>
</host:entityDetail>
```

Code Example: Sample Index

The following code example shows how an `entityList` document could be created to index all deployed objects. This index provides the minimal amount of metadata about each entity so that all entities are not required to be loaded by the client until they are actually used in a guide.

Each entity element in the index contains all the same attributes as is found in each `entityDetail` element in the entity detail document in this code example. There is an additional attribute: `objectLocation`. It provides a URI to the detail document that defines the entire object. If the `objectLocation` attribute is missing, the entity detail must exist in the same directory as the `entityList` and the filename must be the entity name with an `.xml` extension; for example, `accounts.xml`.

```
<host:entityList
      xmlns:host="http://schemas.active-endpoints.com/
      appmodules/screenflow/2012/09/avosObjectDiscovery.xsd">

   <host:entity name="accounts" label="Account"
      labelPlural="Accounts"
      updateable="true" objectIdField="AccountId"
      objectLocation="./accounts_detail.xml">
   </host:entity>

   <host:entity name="contacts" label="Contact"
      labelPlural="Contacts" updateable="true"
      objectIdField="ContactId"
      objectLocation="./contacts_detail.xml">
   </host:entity>
</host:entityList>
```

Entity Relationships

Defining relationships allows guide designers to read and update data from objects related to the object they are currently operating on. In order for the Guide Designer editor to display these relationships, they must be defined in the entity detail. In the following code example, the `childRelationship` element defines the inverse side of the relationship; in contrast, the owning side of the relationship defines a reference to the relationship by defining a field with a type of `reference` and using its options to define `referenceTo` and `relationshipName`.

Code Example: Sample Contact Object

```
<host:entityDetail
   xmlns:host="http://schemas.active-endpoints.com/
      appmodules/screenflow/2011/06/avosHostEnvironment.xsd"
   xmlns:comm="http://schemas.active-endpoints.com/appmodules/
      screenflow/2011/07/avosCommon.xsd" name="contacts"
      label="Contact" labelPlural="Contacts"
```

```
      objectIdField="ContactId">
    <host:detail>
      <host:field label="ContactId" name="ContactId"
            required="true" type="id">
         <host:options>
            <host:option name="length">36</host:option>
         </host:options>
      </host:field>
      <host:field label="Name" name="Name" required="false"
            type="string">
         <host:options>
            <host:option name="length">45</host:option>
         </host:options>
      </host:field>
      <host:field label="AccountId" name="AccountId"
            required="true" type="reference">
         <host:options>
            <host:option name="referenceTo">Account</host:option>
            <host:option
                 name="relationshipName">ctct_to_acct</host:option>
         </host:options>
      </host:field>
    </host:detail>
  </host:entityDetail>
```

If this accounts object was mapped to an SQL database, the `contacts` table would define the foreign key in which contacts `AccountId` references the `accounts` table's `AccountId` column.

At runtime, requests for data from the data services can contain relationship names as part of the field name using the period character as a delimiter between the relationship and field name. For example, a request for a field for an object might be `relationshipName1`, in which case the host provider will need to resolve `relationshipName1` to an object reference and perform a join in order to file the name value. There may be more then one relationship name in the requested field; for example, `relationship1.relationship2.phoneNumber`.

Data Services

The data services layer is where Process Designer interacts with the objects in your host application. It has five basic operations: create, read, update, delete (CRUD), and query. These must be implemented by a host provider so that the guides can operate on objects during runtime. These data services are only used at guide runtime to read and alter data.

Process Designer only attempts to call read, update, and query operations upon your data service.

Note: We recommend that you implement all five operations as they are implemented in future releases of Process Designer in addition to providing your service call steps and search services a single point of execution for invoking create or delete of objects.

The data service supports batch requests for each operation in order to optimize the number of hits against the host application.

A core concept for objects is that each object instance has an `ObjectId` that uniquely identifies it, and it may have an `ObjectName` that is used to render it on the screen in place of the `ObjectId`. In the sample message shown in the Read and Query operation discussions, you'll notice that the `ObjectName` is always returned regardless of whether the request message specified that the field should be returned.

Create

Create is currently not used by the Process Designer framework. However, it is reserved for future use. We recommend that customers implement this operation for use by their custom create services.

The create operation is called when an entity needs to be created in the host. A create request specifies the type of object, the field name, and a value for each field in the object. Only fields in the entity detail marked as required are required to create an entity. An example input message is provided in the Create Message code example. The response from a create request (the create response message code example) is simply an element with an attribute defining the object ID of the newly created object. Optionally in the response, there can be child elements with rollback details.

Code Example: Create Message

```
<types1:create objectType="account_tasks">
      <types1:fieldValue name="Assigned To ID">
         4de025c6-354e-481e-84b9-fbf315486d26</types1:fieldValue>
      <types1:fieldValue name="Subject">Task assigned by </types1:fieldValue>
      <types1:fieldValue name="Description">sfd</types1:fieldValue>
      <types1:fieldValue name="Due Date">2012-10-29T16:00:00Z
         </types1:fieldValue>
      <types1:fieldValue name="Status">Not Started</types1:fieldValue>
      <types1:fieldValue name="Priority">Normal</types1:fieldValue>
</types1:create>
```

Code Example: Create Response Message

```
<types1:createResponse
   id="24db9f6f-2f66-4a19-915e-c9002faf0a11"
   objectType="account_tasks"/>
```

Read

The read operation reads an entity using its object Id and object type. This operation can only return a result of a single object for the ID supplied. In addition, only the fields requested in the read request are returned in the read response. The following code example provides a simple read request message where the `FirstName`, `LastName`, and referenced `AccountName` fields are requested for a single contact from the `contacts` objectType.

Code Example: Read Message

```
<avosHost:read name="$default" objectType="contacts">
   <avosHost:id>925cdc8b-f8e5-467e-ad1b-9270afb97b8f
    </avosHost:id>
   <avosHost:field>FirstName</avosHost:field>
   <avosHost:field>LastName</avosHost:field>
   <avosHost:field>AccountId2.Name</avosHost:field>
</avosHost:read>
```

In the case of the sample project, this request is to the following SQL statement:

```
SELECT c.FirstName, c.BusinessPhone, a.Name
      FROM contacts c
      INNER JOIN accounts a on c.AccountId = a.AccountId
      WHERE c.ContactId = '925cdc8b-f8e5-467e-ad1b-9270afb97b8f'
```

Code Example: Read Response Message

The response message shown in the next code example returns the requested fields along with the `objectId` of the object that was read. Each `fieldValue` specifies the name of the field along with the field value as a text node. `fieldValue` elements can supply a `displayName` attribute with a Boolean value specifying if the field can be used in place of the `ObjectId` when rendering the object on screen. If the metadata description for the object being read defines a `fieldForDisplay` attribute, the read response should return a `fieldValue` with the `displayName` field's attribute

```
<types1:readResponse id="925cdc8b-f8e5-467e-ad1b-9270afb97b8f"
             name="$default" objectType="contacts">
    <types1:fieldValue
             name="FirstName">Mark</types1:fieldValue>
    <types1:fieldValue
             name="BusinessPhone">111-222-5555</types1:fieldValue>
    <types1:fieldValue displayName="true"
```

```
            name="LastName">Ford</types1:fieldValue>
    <types1:fieldValue
            name="AccountId2.Name">Acme Inc.</types1:fieldValue>
</types1:readResponse>
```

Update

Update requests are issued by guides during runtime for each updateable field on a guide screen. The request contains the ID of the object being updated in addition to the fields and values being updated.

Code Example: Update Message

```
<host:update objectType="contacts"
        xmlns:host="http://schemas.active-endpoints.com/
        appmodules/screenflow/2011/06/avosHostEnvironment.xsd">
   <host:id >925cdc8b-f8e5-467e-ad1b-9270afb97b8f</host:id>
   <host:fieldValue name="FirstName">Mark</host:fieldValue>
   <host:fieldValue name="LastName">Lords</host:fieldValue>
   <host:fieldValue name="Email">mark@myemail.com</host:fieldValue>
</host:update>
```

The response for an update message is simply an empty `updateResponse` element. Optionally, it can have rollback instructions.

Query

A query operation is essentially a read operation on the host provider; however, it isn't limited to reading a single object. Query requests can provide complex WHERE clause statements that filter objects by any of the object's field. Other query options, besides a WHERE clause, can query by the `ObjectName` or a collection of object Ids.

Code Example: Query Response

The following example queries for the first and last name of the three objects defined in the id element. The response for this query request will result in three items returned.

```
<ns:query limit="100" name="Object Query" objectType="contacts"
      xmlns:ns="http://schemas.active-endpoints.com/appmodules/
         screenflow/2011/06/avosHostEnvironment.xsd">
   <ns:field>FirstName</ns:field>
   <ns:field>LastName</ns:field>
   <ns:id>925cdc8b-f8e5-467e-ad1b-9270afb97b8f</ns:id>
   <ns:id>b6f0f0b2-9f38-4771-8365-58eb4f7b7d41</ns:id>
   <ns:id>d75e2a5e-6ae2-44e2-b3df-ea10b6b17d82</ns:id>
</ns:query>
```

The following query request looks for contacts objects that contain a "D" in their display name.

```
<ns:query limit="100" name="Object Query" objectType="contacts"
      xmlns:ns="http://schemas.active-endpoints.com/appmodules/
         screenflow/2011/06/avosHostEnvironment.xsd">
   <ns:field>FirstName</ns:field>
   <ns:field>LastName</ns:field>
   <ns:name exactMatch='false'>D</ns:name>
</ns:query>
```

The following query looks for contacts with a first name property containing "B" and which is associated with `AccountId 13`.

```
<ns:query limit="100" name="Object Query" objectType="contacts"
      xmlns:ns="http://schemas.active-endpoints.com/appmodules/
         screenflow/2011/06/avosHostEnvironment.xsd">
   <ns:field>FirstName</ns:field>
   <ns:field>LastName</ns:field>
   <ns:where>FirstName like '%B%' and AccountId = 13</ns:where>
</ns:query>
```

Code Example: Query Response Return

The response for the queries in the three previous code examples return response messages that looks something like what is shown in the next code example. Each entry in the query returns a `fieldValue` for each field requested and additionally marks the `fieldValue` that serves as the `ObjectName` with a `displayName` attribute of true.

```
<types1:queryResponse name="Object Query" objectType="contacts">
   <types1:entry id="925cdc8b-f8e5-467e-ad1b-9270afb97b8f">
      <types1:fieldValue
            name="FirstName">Mark</types1:fieldValue>
      <types1:fieldValue displayName="true"
            name="LastName">Ford</types1:fieldValue>
   </types1:entry>
   <types1:entry id="b6f0f0b2-9f38-4771-8365-58eb4f7b7d41">
      <types1:fieldValue
            name="FirstName">John</types1:fieldValue>
      <types1:fieldValue displayName="true"
            name="LastName">Acme</types1:fieldValue>
   </types1:entry>
   <types1:entry id="d75e2a5e-6ae2-44e2-b3df-ea10b6b17d82">
      <types1:fieldValue
         name="FirstName">Joe</types1:fieldValue>
      <types1:fieldValue displayName="true"
         name="LastName">Biden</types1:fieldValue>
   </types1:entry>
</types1:queryResponse>
```

Rollbacks

A rollback undoes the changes made to objects when a guide user navigates back to the step that made the change. For example, if the first step changes the name field of an object, the object is updated as soon as the user navigates to the second step. If the user goes back to the first screen, a rollback reverses the name change.

The operations that support rollbacks are those that mutate objects: create, update, and delete. The rollback segment of rollback response messages allows a custom user-defined service to be named to undo work already done.

The Process Designer framework does not automatically handle rollback segment creation. This means that it does not call a delete on your data server to undo a create operation.

Code Example: Rollback

Items in bold text are discussed in the paragraph following this example.

```
<aetgt:createResponseWrapper
      xmlns:aetgt="http://schemas.active-endpoints.com/
         appmodules/screenflow/2011/06/avosHostEnvironment.xsd"
      xmlns:types1="http://schemas.active-endpoints.com/
         appmodules/screenflow/2011/06/avosHostEnvironment.xsd"
      xmlns:xsd1="http://schemas.active-endpoints.com/
         appmodules/screenflow/2010/10/avosScreenflow.xsd">
   <types1:createResponse
         id="24db9f6f-2f66-4a19-915e-c9002faf0a11"
         objectType="account_tasks">
      <xsd1:onRollback deleteCount="1">
         <xsd1:serviceName>dbDeleteAnyEntityService
               </xsd1:serviceName>
         <xsd1:avosServiceName>dbDeleteAnyEntityService
               </xsd1:avosServiceName>
         <xsd1:runAsUser/>
         <aesf:hostContext
            xmlns:aesf="http://schemas.active-endpoints.com/
               appmodules/screenflow/2010/10/avosScreenflow.xsd"
            xmlns:aetgt="http://schemas.active-endpoints.com/
               appmodules/screenflow/2010/10/avosScreenflow.xsd"
```

```
            xmlns:sf="http://schemas.active-endpoints.com/
               appmodules/screenflow/2010/10/avosScreenflow.xsd">
            <aesf:arg name="ObjectType">contacts</aesf:arg>
            <aesf:arg name="ObjectId">
               925cdc8b-f8e5-467e-ad1b-9270afb97b8f</aesf:arg>
         </aesf:hostContext>
         <xsd1:parameters>
            <xsd1:parameter
               name="ObjectType">account_tasks</xsd1:parameter>
         <xsd1:parameter
            name="ObjectId">24db9f6f-2f66-4a19-915e-c9002faf0a11
           </xsd1:parameter>
         </xsd1:parameters>
      </xsd1:onRollback>
   </types1:createResponse>
</aetgt:createResponseWrapper>
```

This example declares that the Process Designer framework will execute the `dbDeleteAnyEntityService` service provided in the sample, and delete the `account_tasks` ObjectType where the `account_task` object has an `ObjectId` of "24db9f6f-2f66-4a19-915e-c9002faf0a11". The host context arguments and the `runAsUser` element is additional metadata that can be used by your service when executing the delete.

Automated Steps

Service Call steps can interact with systems using data provided by guide during guide runtime. These steps have no visual component to them during runtime.

See the Service Call documentation elsewhere in this help.

Search Services

Search services can dynamically populate data (for users to choose from) on the screen, at runtime. They only apply to updateable fields on the screen. Most often, search services fill picklists and tables with values that the user will choose from. They can also be used for fields that accept a single value or those that accept multiple values.

Search services are especially useful if the list of values to choose depends on other user choices on the same screen. For example, say you want two fields on the screen: one to pick a state and one to pick a city. In this case, when a user selects a state, the list of cities within the state can be dynamically populated by a search service.

Creating a Search Service

A few search services are provided with Process Designer. A search service can be implemented using a BPEL process.

A search service is defined in an XML file (typically `services.xml`) indicating the service name, input and output parameters, and a few other details.

```
<searchService name="service:/avosObjectQuerySearchService">
   <displayName>Object Query</displayName>
   <description>Querying of objects is done by specifying zero or more
   <expressions as a WHERE clause. The results of the query are displayed
   <as a table or as a dropdown, depending on the number of display
   <columns used.</description>
   <mode name="modes" />
   <input>
      <parameter
         name="Where Clause"
         type="whereclause"
         required="false"
         description="A description for the search service." />
```

```
      <parameter name="Domain" required="false" type="string"
                 description="QC Domain"/>
      </parameter>
   </input>
   <searchOutput type="reference">
      <options>
         <option name="referenceTo"></option>
      </options>
   </searchOutput>
</searchService>
```

Here are definitions of elements you can use:

- **mode element:** Indicates when the search service should be invoked. If it is not specified, the search service is only used or invoked at runtime. However, if the search service can also be used at design time for populating input parameters in the editor or used when simulating or previewing the guide, add this element. Supported modes are `designtime`. `simulation`, `preview`, and `runtime`.
- **input section:** Describes the parameters expected by the search service.
- **searchOutput section:** Describes the data type for which the search service is applicable. This allows for showing a filtered list of search services at design time.
- **reference type:** The search service should support the `display-options` parameter. It could contain a list of columns for which search values are returned.

Implementing a Search Service Using a BPEL Process

To implement a search service using a BPEL process, create a process using the `Guide Designer List` system service which has the `List Operations` WSDL interface and a `list` operation. As easy way to get started is to look at the sample search service in the starter project.

The name of the search service should be the Guide Designer process service name having a `service:/` prefix.

Implementing a Search Service Using a REST Service

TBD.

Built-in Search Services

The search services included with Process Server are:

- **Object Query**: Used to query for objects using a WHERE clause. This search service can be used for fields of type `reference`.
- **Advanced Query**: Used to query for any arbitrary list of values. This search service can be used for fields of type `reference`.
- **List Child Objects**: If your data model supports parent-child relationships, this search service can retrieve list of children of a specified type. This search service can be used for fields of type `reference`.

Data Types and the Repository API

This section describes the following:

- Built-In Data Types
- Custom Data Types
- Process Designer Repository API

Built-In Data Types

Process Designer ships with a variety of supported data types along with a default renderer for each. Data types can be customized with a variety of options that affect how a field is rendered and validated.

Type: boolean

The **boolean** data type supports rendering of true/false data in either a check box control or a pair of radio buttons with the labels *Yes/No* or *No/Yes*. Options are as follows:

Option Name	Description
boolean_show_as	Choose how the boolean is rendered. Choices are: - Checkbox - Yes/No - No/Yes

Type: currency

The **currency** data type extends the base type double and inherits all the options supported by **double**. It also has the following options:

Option Name	Description
scale	The maximum number of digits to the right of the decimal point. If you enter 0, there cannot be digits to the right of the decimal point. The default is 2

Type: date

The **date** data type renders a date picker control to choose a date that does not have a time component. There are no options to this data type.

Type: datetime

The **datetime** data type renders a date/time picker control to choose a date and time. There are no options to this data type.

Type: double

The **double** data type can contain fractional portions. Options are as follows:

Option Name	Description
precision	Total number of digits, including those to the left and the right of the decimal place
scale	Maximum number of digits to the right of the decimal place
hover_text	Tool tip value displayed when a user hovers over the control
min	The minimum value allowed
max	The maximum value allowed

Type: email

An **email** data type contains an email address.

Type: formatted string

The **formattedString** data type has a base type of string so it inherits all of its options. It also has the following options:

Option Name	Description
format	A custom format; see http://digitalbush.coojects/masked-input-plugin/ for supported formatting syntax

Type: id

The **id** type is used to for `ObjectId` fields. This type extends the base **string** type, so it inherits all of its options. It also has the following options:

Option Name	Description
referenceTo	The "name" of the object defining the ID field.
display_as_name	Boolean value indicating if the `ObjectId` or object name is rendered on screen.

Type: int

The **int** data type contains numbers with no fractional portion. Options are as follows:

Option Name	Description
digits	The maximum number of digits that an integer can have
hover_text	Tool tip value displayed when a user hovers over the control
min	The minimum value allowed
max	The maximum value allowed

Type: multipicklist

The **multipicklist** data type is a picklist from which one or more values can be selected and includes a set of enumerated values. This type extends **picklist**. Options are as follows:

Option Name	Description
multiSelect	When set to true, multiselection within picklists can be used

Type: objectlist

An **objectlist** data type represents one or more object reference. It extends the base type **picklist**. It also has the following options:

Option Name	Description
referenceTo	The name of the object the `ObjectList` will reference

display-options	A complex JSON structure that contains columns to display and any pagination and filtering options when references are stored in a data table.
multiSelect	When set to true (which is the default), multiple values can be selected from the picklist

Type: percent

The **percent** data type extends the base type **double** and inherits all the options supported by **double**. It also has the following options:

Option Name	Description
precision	The maximum number of digits in the number. The default: 5.
scale	The maximum number of digits to the right of the decimal point. If you enter 0, there are no digits to the right of the decimal point. The default: 2.
min	The minimum percent value. The default: 0
max	The maximum percent value. The default: 100

Type: phone

The **phone** data type contains phone numbers, which can include alphabetic characters. A guide's designer is responsible for phone number formatting.

Type: picklist

The **picklist** data type is a list or choice that is rendered as an HTML option list. The picklist type extends the **string** base type. It also has the following options:

Option Name	Description
values	Picklist values in the form of the picklist element type. This is a comma-separated list of the items in the list.

Type: reference

The **reference** data type is used when a field refers to the `ObjectId` of another object, which makes the `referenceTo` option required and the value will be the name of the object being referred to. The base type of reference is **id**, so it inherits all of the **id** and **string** options plus those shown in the following table:

Option Name	Description
display-options	A complex JSON structure that contains columns to display and any pagination and filtering options when references are stored in a data table

Type: richtextarea

A **richtextarea** data type extends the base **textarea** type and supports rich formatting of text. It also has the following options:

Option Name	Description
text_width	The width of the text area relative to the guide's width.
text_height	The number of rows within the area the guide displays into which the user can type text.
length	The maximum number of characters that can be entered in the text area.
text_size	The size of the displayed text.

Type: string

Character strings are of data type **string** and contain text. Some have length restrictions, depending upon the data being stored. Options are as follows:

Option Name	Description
text_size	Font size of the text on the screen
hover_text	Tool tip value displayed when a user hovers over the control
length	The number of characters in the string

Type: textarea

A **textarea** type extends the base string type, so it inherits all of the string option. It also has the options shown in the following table:

Option Name	Description
text_width	The width of the text area relative to the guide's width.
text_height	The number of rows within the area the guide displays into which the user can type text.
length	The maximum number of characters that can be entered in the text area.
text_size	Font size of the text on the screen

Type: time

A **time** data type handles time values.

Type: url

A **url** data type contains a URL. Options are as follows:

Option Name	Description
urlDisplayReadonlyAs	Your choices are Link, Button, or IFrame
urlDisplayReadOnlyLabel	Text that identifies the contents of the field.

frame_height	The height of the area in which the URL's target is displayed.
frame_width	The width of the area in which the URL's target is displayed.

Custom Data Types

Custom data types allow for custom rendering of fields in your entity. They also allow Process Designer to do type matching; this provides the guide developer more appropriate choices when mapping data to fields. For example, if the input to a service call step requires an `email` type, the Guide Designer editor will only show you options for the `email` type.

The sample project has an example of creating a custom type with a custom renderer. If you want to use an existing renderer with a new data type, override a base type and do not add a renderer. This means that your new data type inherits the renderer from its base type.

The following is an example of specifying an `accountNumber` type that has a base type of `string`. Even though this new type will render as a `string` during design time, guide developers will have more relevant data mappings for it.

Code Example: Sample Custom Data Type

```
<tns:type name="accountNumber" base="string"
      label="Account Number">
   <tns:description>Account Number</tns:description>
</tns:type>
```

ScreenFlow Repository API

The sample project has a sample that uses the repository JavaScript API to query Process Designer for guides, and which are then displayed to your end users so they can choose one to execute. This repository API is useful if your application needs to dynamically present users with a list of available guides in menu form.

The following sample function queries for all guides using the `getEntries()` `activevos.socrates.screenflow.central.repository.AeScreenFlowRepository` function. `getEntries()` takes two arguments:

- `filter`, which is currently not implemented.
- A callback function that is called when the Ajax request completes. It receives an array of `Item` objects whose properties are shown in the Sample Item Object code sample.

Code Example: Query for Guides

```
function() {
   var repo = new activevos.socrates.screenflow.central.
                        repository.AeScreenFlowRepository();
   var filter = null; // currently not used
   repo.getEntries(filter,
         function(aSuccess, aItemList) {
            if (aSuccess) {
               for (var i = 0; i < aItemList.length; i++) {
                  var item = aItemList[i];
               }
         }
         });
};
```

Code Example: Sample Item Object

```
{
   "Item" : {
```

```
      "EntryId"                 : { "$t" : "" },
      "Name"                    : { "$t" : "" },
      "MimeType"                : { "$t" : "" },
      "Description"             : {           },
      "AppliesTo"               : { "$t" : "" },
      "Tags"                    : {           },
      "VersionLabel"            : { "$t" : "" },
      "State"                   : { "$t" : "" },
      "ProcessGroup"            : {           },
      "CreatedBy"               : { "$t" : "" },
      "CreationDate"            : { "$t" : "" },
      "ModifiedBy"              : { "$t" : "" },
      "ModificationDate"        : { "$t" : "" },
      "PublishedBy"             : { "$t" : "" },
      "PublicationDate"         : { "$t" : "" },
      "PublishedContributionId" : { "$t" : "" },
      "AutosaveExists"          : { "$t" : "" }
   }
}
```

Image Resources

If you wish to provide images to guide designers, you must package them into the contribution being deployed along with a `screenflowContribution` metadata file. In the following code example, the metadata file is providing a single image with a display name of "Image 1" located in the same directory as the metadata file. The image resource `content-type` is also specified with the `type` attribute.

Code Example: Image Metadata

```
<tns:screenflowContribution
      xmlns:tns="http://schemas.active-endpoints.com/
      appmodules/screenflow/2011/07/avosResourceDiscovery.xsd">
   <tns:resource name="image-1.jpg" type="image/jpg">
      <tns:displayName>Image 1</tns:displayName>
      <tns:resourceURI>image-1.jpg</tns:resourceURI>
   </tns:resource>
</tns:screenflowContribution>
```

Guide Embedding and the Forms Viewer

Embedding a guide into your application involves adding an HTML IFrame with the source attribute set to the URL for your guide. The guide URL will include query string arguments that allow for the hosting application to provide additional parameters about the current guide user, the object the guide will execute upon, and even the hosting application itself.

The URL for the guide runtime is:

```
http://localhost:8080/activevos-central/avc/formviewer/
   project:/com.activevos.socrates.central/web/viewer.html
```

You should replace replace `localhost` and port 8080 with those used by your servert

You can add query string parameters to this URL. For example the following adds the `avsf_sflow_uri` parameter that names the location of a published guide whose location is `project:/sf.Custom_Guide/Custom_Guide.xml`.

The full URL will look like:

```
http://localhost/activevos-central/avc/formviewer/project:
    /com.activevos.socrates.central/web/
    viewer.html?avsf_sflow_uri=project:
    /sf.Custom_Guide/Custom_Guide.xml
```

You can add additional query string arguments to ensure that users can only have one running instance of a guide for an object and to provide context about the hosting application.

Here is a description of all query string parameters that the viewer application uses:

- `avsf_sflow_uri`: The URL-encoded value of guide's published contribution URI, as was shown in the example.
- `correlationId`: An optional correlation ID for the guide. The ID is a value the application developer provides. It can have any unique string value. If you do not specify one, the server will create and use one. For example, Cloud Extend (an implementation that runs within Salesforce) uses the following formula: `correlationId = Salesforce org ID + object id` (for example, an account) `+ guide ID` (or name)

  After a guide is launched, you can reuse this correlation ID (since it can be recalculated) to select the same guide instance when the user reloads or revisits the web page.
- `host-ObjectType`: The host entity type such as an account or contact.
- `host-ObjectId`: The object (for example, an account or a contact) ID or a primary key.
- `host-UserId`: The currently logged in username. In Process Server, you can use the JavaScript `AE_ACTIVEVOS_PRINCIPAL_NAME` global.

Guide Correlation

You may want to prevent a user from starting more than one guide that operates upon on a single object at a time. Use correlation to ensure that a user will rejoin any existing executing guides for the object. By setting the `correlationId` query string argument to a value, you can ensure that a user will always re-enter an existing executing guide. A good correlation is one that can reliably be reconstructed even if the user logs out and back into the host system.

Host Context Locale and Time Zone

The host context is data provided by the host application that is accessible to the Guide Designer framework and is passed to all Service Call steps, as well as data provider and search service processes.

Setting Up the Host Context

This host context data is essentially a key/value pair of strings that contain additional data about the host application, guide user, embedding application, and so on. These parameters are provided to a guide during runtime by adding a **host-** prefix (notice the hyphen used as a suffix) to the parameter's name and passing them as query string arguments on the guide URL. For example, if your host provider implementation requires an argument for embedding an application ID named `AppId`, you would add the following to the guide URL: `host-AppId=someValue`.

Three host context parameters names are reserved by Guide Designer and may be required when embedding or executing your guide.

Name	Description
`ObjectType`	This is required if your guide applies to an object. It is not required if your guide applies to any object (that is, it is not directly associated with an object). The value is the name of the object.
`ObjectId`	This is required if your guide applies to an object. It is not required if your guide applies to any object. The value is the ID of the object the guide will execute on.
`UserId`	This is optional, but may be used to set the `Current User` for the current guide. This may be useful if the ID for the embedding application needs to be translated to something else for guide execution.
`ExtraInfo`	Additional metadata made available to reports for filtering or sorting.

So to set the `ObjectType` to `contacts`, your embedded URL would contain a query string argument like `host-ObjectType=contacts`. At runtime, the `hostContext` element would contain an `arg` element with a name attribute of `ObjectType` and a text node value of `contacts`.

Setting the User's Locale

You can set user locale information to your application by using the `host-UserLocale` URL parameter; for example:

```
host-UserLocale=en_US
```

The arguments to `host-UserLocale` are those defined in the ISO 639 standard.

Setting the User's Time Zone

You can set the time zone for your application by using the `host-UserTimeZone` URL parameter; for example:

```
host-UserTimeZone=America/Los_Angeles
```

The arguments to `host-UserTimeZone` are those defined within the `tz_database`; see `http://en.wikipedia.org/wiki/List_of_tz_database_time_zones` for more information.

Setting Business Days in a Calendar

When setting a dates, you may want your users to set date intervals using business days rather than normal calendar days. While Guide Designer does understand that a business week differs from a normal week in that it only has five days, it doesn't know what holidays your company offers. Set your company's holidays using a `HolidayCalendar` element. Here's an example:

```
<HolidayCalendar xmlns="http://schemas.active-endpoints.com/appmodules/screenflow/
2012/12/avosHostCalendar.xsd">
   <WeekendDay name="Saturday" dayOfWeek="Saturday" endDate=""/>
   <WeekendDay name="Sunday" dayOfWeek="Sunday"/>
   <AnnualMonthDayHoliday name="New Year's Day" month="January" day="1"
weekendMoveToNext="Monday"/>
   <AnnualDayOfWeekInMonthHoliday name="President's Day" month="February"
dayOfWeek="Monday" occurance="Third"/>
   <AnnualDayOfWeekInMonthHoliday name="Memorial Day" month="May" dayOfWeek="Monday"
occurance="Last"/>
   <AnnualMonthDayHoliday name="Independence Day" month="July" day="4"
weekendMoveToNext="Monday"/>
   <SpecificDateHoliday name="Day After Independence Day" startDate="2013-07-05"/>
   <AnnualDayOfWeekInMonthHoliday name="Labor Day" month="September" dayOfWeek="Monday"
occurance="First"/>
   <AnnualDayOfWeekInMonthHoliday name="Columbus Day" month="October" dayOfWeek="Monday"
occurance="Second"/>
   <AnnualDayOfWeekInMonthHoliday name="Thanksgiving" month="November"
dayOfWeek="Thursday" occurance="Fourth"/>
   <AnnualDayOfWeekInMonthHoliday name="Day After Thanksgiving" month="November"
dayOfWeek="Friday" occurance="Fourth"/>
   <AnnualMonthDayHoliday name="Christmas Day" month="December" day="25"/>
</HolidayCalendar>
```

Notice the `avosHostCalendar.xsd` namespace. If this isn't declared, Guide Designer will not know that this is a calendar file.

After you deploy your XML file, Guide Designer will understand your company's holidays.

Object Linking to Host Application

If your host application has landing pages for the objects in the application, you may want to provide HTML links in the guide to the objects page in the host application. For example, an object whose `ObjectId` is 123 can be accessed in the hosting application using the following URL: `http://hostapp.net/objects/123`. This link would then appear in the guide.

You will need to create a JavaScript function that can build a URLs from the Object Id. The following code examples shows uses the Object Id to build and return the URL. The `getLinkToObject` function is also passed the host context in case additional information about the host is needed to build the URL.

Code Example: Link Builder Sample

You must assign your builder to `activevos.socrates.screenflow.data.renderers.provider` so that it can be discovered at runtime. This is shown in the second to last line in the following example.

```
(function($){
    activevos.util.createAvosPackage("hostapp.link.generator");
    function MyLinkBuilderUtil() {
        this.getLinkToObject = function(aHostContext, aId) {
            return "http://hostapp.net/objects/" + aId;
        };
    }
    activevos.socrates.screenflow.data.renderers.provider =
            new MyLinkBuilderUtil();
})(jQuery);
```

Code Example: Configuration to Deploy Link Builder

The following is an example of the Process Central configuration file required to deploy your custom JavaScript function into a guide.

```
<tns:avosCentralConfiguration
                xmlns:tns="http://schemas.active-endpoints.com/avc/
                2009/07/avoscentral-config.xsd" >
    <tns:centralIncludes>
            <script xmlns="http://www.w3.org/1999/xhtml"
                src="link-builder.js"></script>
    </tns:centralIncludes>
</tns:avosCentralConfiguration>
```

Embedding Guide Designer

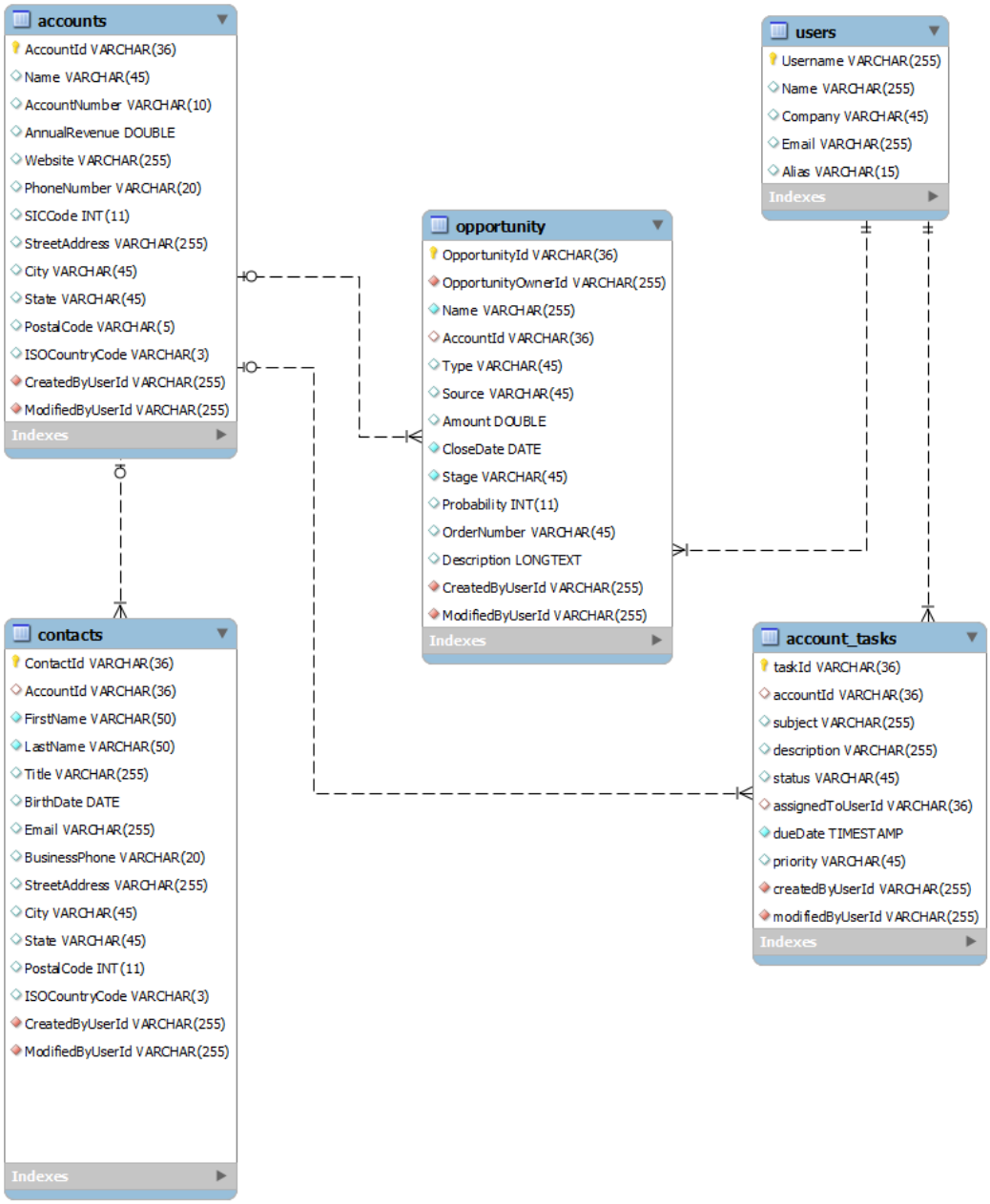
An HTML IFrame can also be used to embed the Guide Designer into your application using the following URL:

```
https://localhost:8080/activevos-central/avc/avc.jsp
```

By default, this URL embeds all of Process Central with all of Central's additional UI components. Use the `aeembed=true` query string parameter to remove the Central header. If you wish to remove Human Task, Forms, and Reports from Central set the `aemgrs=socrates` query string parameter.

Sample Project Guides

The sample project models a CRM that is similar to Salesforce.com that uses an SQL database as the host application in which Process Designer operates. All guides operate on one or more entities that are contained within database tables. Here is the schema for these entities:



Desktop Set

- **Sales Call with Contact Prompt**: Prompts the user for a contact and then runs the *Sales Call Follow-up (Mobile)* embedded guide.
- **Sales Call Follow-up**: After a meeting ends, users can enter new action items, notes, contacts, and opportunities. They can also set follow-up action items for themselves and others.

Logging into Guide Designer

You can access Process Designer from Process Central. Start the Process Server and then open Process Designer by typing a URL into a browser that supports HTML 5--most modern browsers except for Internet Explorer 7 and 8 do.

```
http://[host]:[port]/activevos-central
```

The default is `http://localhost:8080/activevos-central`.

Log in with a user name and password defined in your identity service. For example in the server embedded in Process Developer, you can log in with *loanrep1/loanrep1*.

Process Designer editors are accessed using the Designs pane.

Note: Every guide developer must be assigned to the `abTaskClient` security role. If the Process Server is secured, users must also be assigned to the `abServiceConsumer` security role. If your server is secured and you do not have this role, Process Server displays a 403 transport error and you will see network errors from `avMetaInfoAccessService` and `avThemeDiscovery`. Also, an administrator can change the `guidePermissions` feature element within `app-config.xml` so to set permissions. When set to true, the **Settings** dialog box within the Design Home toolbar has a Permissions tab.

What is a Guide Designer Automated Step

A Guide Designer guide is built with steps. There are only a few step types, such as a *screen step* in which there is a prompt and answers that automatically create new steps.

A Service Call step is another step type. This is described in the Service Call discussion elsewhere in this help. This topic looks at creating services using Process Developer.

The steps for creating an a service call step are:

1. Create a new BPEL process.
2. In the Participants view, create a new Process Service Consumer based on the Process Designer system service.
3. Build the process to do something required by a guide. A process's requirements include:
   - Defining the input for the service. This can be required and user-defined data, and it can either be simple or complex. Process Designer editors can add their own data by defining fields in the **Properties** dialog.
   - Setting response messages from the service.
   - Naming the resources the guide will use, such as images.
   - Creating the HTML renderings for complex data types.
4. Create the `services.xml` file for the service all step service descriptor.

Note: For additional getting started resources, create a new orchestration project and select the Guide Designer Customization Starter template.

Using the Automated Step Request Response and Fault Messages

Your first step is to create a Process Service Consumer with the service call system service, as follows:

1. In the Participants view, create a new Process Service Consumer.
2. Select System Services from the Interfaces tree.
3. Select Guide Designer Automated Step.
4. Drag the automatedStep operation to the canvas to create a receive-reply pair of activities.

Request Message

Note: In earlier releases, service call steps were called automated steps.

The following sample data shows an example of the request message:

```
<xsd1:automatedStepRequest
    xmlns:xsd1="http://schemas.active-endpoints.com/appmodules/
        screenflow/2010/10/avosScreenflow.xsd">
   <xsd1:userId>string</xsd1:userId>
   <xsd1:correlationId>string</xsd1:correlationId>
   <xsd1:serviceName>string</xsd1:serviceName>
         <!--Optional:-->
   <xsd1:avosServiceName>string</xsd1:avosServiceName>
         <!--Optional:-->
   <xsd1:avosStepTitle>string</xsd1:avosStepTitle>
         <!--Optional:-->
   <xsd1:hostContext>
         <!--Optional:-->
      <xsd1:arg name="string"/>
   </xsd1:hostContext>
        <!--Optional:-->
   <xsd1:parameters>
        <!--Optional:-->
      <xsd1:parameter name="string"/>
   </xsd1:parameters>
       <!--Optional:-->
   <xsd1:guideData>
       <!--Optional:-->
      <xsd1:parameter name="string"/>
   </xsd1:guideData>
</xsd1:automatedStepRequest>
```

The elements of the message are as follows:

automatedStepRequest input element	**Description**
`avosServiceName`	The actual Process Server service name that is used to call the service. Except for advanced cases, this name is usually the same as `serviceName`. This element is usually not needed by service call steps.
`avosStepTitle`	The title given by the user when creating a step that executes this service call. This is useful for formatting error messages.
`correlationId`	Generated ID for the running guide instance. You can also create parameters that can be added for the URL for a running guide.
`guideData`	Contains the complete set of input and output fields in the running guide. It is only passed if the service call describes itself as wanting it by setting the attribute `passAllData="true"` in the service element for the service call in the `services.xml` file. Additionally when using `passAllData`, the service can modify the data and return changes in the `guideData` element of the response.
`hostContext`	Provides information used when the Process Designer system is operating in a host data mode. This is normally not used by service call steps.
`parameter`	Contains the name/value pairs for the input parameters that describe this service call in `services.xml`.
`parameters`	This is the most important part of the code. It has name/values pairs that were described as input to this service call in `services.xml`.

automatedStepRequest input element	Description
serviceName	The service name used to describe this service in the services.xml file. This name can be different than the Process Server service name. This lets you reuse the same service, presenting different service call step information to the user. This is an advanced capability not used by most service call steps.
userId	Process Central login name of the user running the guide calling this step.

Reply Message

Note: In earlier releases, a service call step was called a Service Call step.

The following sample data shows an example of the reply message:

```
<xsd1:automatedStepResponse
           xmlns:xsd1="http://schemas.active-endpoints.com/appmodules/
           screenflow/2010/10/avosScreenflow.xsd">
   <!--Optional:-->
   <xsd1:fields>
      <!--Optional:-->
      <xsd1:field name="string"/>
   </xsd1:fields>
      <!--Optional:-->
   <xsd1:guideData>
      <!--Optional:-->
      <xsd1:parameter name="string"/>
   </xsd1:guideData>
</xsd1:automatedStepResponse>
```

The elements of the message are described in the following table:

automatedStepResponse output element	Description
fields	Contains the data described as output fields for the service call step. They are name/value pairs where the *name* is in the attribute name and the *value* is the data under the element. For both input and output, the field can be defined as complex (attribute complex="true") in the services.xml file on the parameter or field). In that case, the value can be arbitrary XML that can be used in a renderer. See the Starter project example for a simple example used in Hello World.
guideData	Contains the new data to be set for the guide context data if the service has passAllData="true" in the services.xml file

Example Using Complex-typed Data

You can use complex-typed data in a service call step. In the Starter project, refer to the renderings and service-contribs files as one example.

The HTML Rendering File for Complex Data

In order to display the above data in a guide, you must provide an HTML file that renders XML message data into HTML.

Note: Be sure to see the Project Orchestration template for the Guide Designer Customization Starter project, which provides a starting point for creating rendering and theme files. Also, note that the rendering file is deployed as part of the service call step's deployment contribution.

Data can also be rendered with a rendering hint based on a JSON object literal.

A guide's developer should not have to think about fault handling when building service call steps. Your process can handle two fault types by using the *automatedStep* operation's fault messages in callback responses. You do not need to add process or scope fault handlers to your process. Use a *Reply with Fault* for fault handling. An internal Process Designer service catches the fault and presents an error to the user.

The two fault messages are:

- **automatedStepValidationFault** with a fault name of `invalidData`
  You can add programming logic to your BPEL process to check for invalid data items in the request message. You can also create a callback response containing an error message that indicates missing or invalid data. Doing this lets the user correct the request data and resubmit it. Essentially, an internal service catches the fault and presents an error screen to the user. See the example below of using the reply with fault handler. A sample fault message looks like this:

  ```
  <xsd1:invalidData
          xmlns:xsd1="http://schemas.active-endpoints.com/
              appmodules/screenflow/2010/10/avosScreenflow.xsd">
     <xsd1:field name="Customer Id"
          reason="Incorrect number of digits" required="true"/>
  </xsd1:invalidData>
  ```

  where

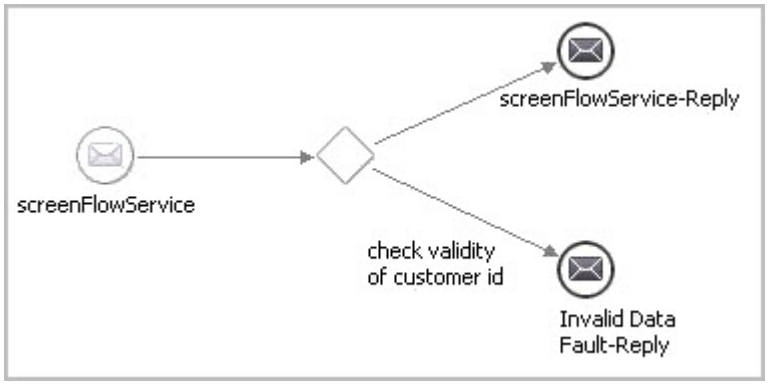
  - **field name**: The data field
  - **reason**: The string describing the problem
  - **required**: A Boolean indicating true for required and false for optional. If the field is required, the user is not allowed to advance to the next screen without valid data entered.

- **automatedStepFault** with a fault name of `fault`.
  This fault is similar to the invalid data fault, but allows you to create an error message that tells the user to make corrections; for example, the user cannot schedule a Saturday delivery to the address entered. This fault can also handle cases where a new version of the BPEL process is deployed with new required fields after a guide is published.

  A sample fault message looks like the following:

  ```
  <xsd1:screenFlowFaultResponse
          xmlns:xsd1="http://schemas.active-endpoints.com/
               appmodules/screenflow/2010/10/avosScreenflow.xsd">
      <xsd1:reason>string</xsd1:reason>
  </xsd1:screenFlowFaultResponse>
  ```

  The following illustration shows an example of creating a reply with fault activity to check for invalid data. The link to the callback reply contains a transition condition that checks for invalid data.



Providing Rendering Hints for Guide Data

The *Using the Automated Step Request, Response, and Fault Messages* topic described how you can provide a rendering file and rendering file parameters for data items.

The formatting types of rendering hints are based on JavaScript Object Notation (JSON). JSON is an extensible format that allows you to configure rendering parameters. Using JSON lets you reuse the same data type different ways. For an introduction to JSON, see *Customizing Task and Form Scripts: An Introduction* elsewhere in this help.

Basic JSON format

A rendering file is described as a JSON object literal with a property named `form`. This property holds the rendering location of an HTML page deployed to the Process Server. The page can contain JavaScript code to describe how to render XML message data into HTML format, such as a video, interactive map, or product order. You provide a rendering file as part of the contribution.

The JSON format basic configuration is shown in the following example:

```
<tns:dataDef name="Customer Address Formatted" type="complex">
   <tns:description>Customer address formatted with rendering
   </tns:description>
   <tns:renderingHint>  {
          "form" : "project:/address-rendering.html"
     }
   </tns:renderingHint></tns:dataDef>
```

Note: The curly braces and the form parameter are required.

In addition to this basic configuration, you can use additional Guide Designer parameters and you can provide your own, which is described in sections following this one.

Multi-line edit (textarea) for strings

You can enable `textarea` HTML controls for strings by adding `rows` and `cols` JSON rendering hints that are built-into Process Server. For example:

```
<tns:dataDef name="Email Message" type="string">
   <tns:description>Large text area for email messages</tns:description>
   <tns:renderingHint>{ "rows" : 3, "cols" : 40}
   </tns:renderingHint>
</tns:dataDef>
```

Additional Format Options for Strings

You can add a JSON format as a rendering hint for strings. For example, a U.S. telephone number can have a rendering hint such as:

```
<tns:renderingHint>
    { format : "(999) 999-9999" }
</tns:renderingHint>
```

The format consists of the following special characters:

- a: Represents an alphabetic character (A-Z,a-z)
- 9: Represents a numeric character (0-9)
- *: Represents an alphanumeric character (A-Z,a-z,0-9)

Some example formats are:

- "(999) 999-9999" (U.S. phone number)
- "999aaa" (an Id with three digits and three letters)

Adding Customizations

As part of a rendering hint, you can add your own custom parameters. The parameters are those that you have created in your rendering file.

In the following example, a `myBoolean` parameter in `email-rendering.html` can have two different values, such as `myCheckbox` or `myRadioButtons`. For the rendering hint, the `myCheckbox` value is selected:

```
<tns:dataDef name="Email Short Form" type="complex">
  <tns:description>Comments area</tns:description>
   <tns:renderingHint>
      { "form" : "project:/email-rendering.html",
         myBoolean : myCheckbox
      }
   </tns:renderingHint>
</tns:dataDef>
```

This rendering description is only used for displaying data in a guide. No formatting changes are made to the underlying data.

Creating the Automated Step Service Descriptor

Note: In earlier releases, a service call step was called an automated step.

You must add a services.xml file to the deployment package for a service call step. This file contains the configuration details added to Process Designer that make the service call available and describe how data is used.

The file is based on the `avosServiceDiscovery.xsd` schema located in the `com.activevos.socrates.repository.services\schema` plugin.

The elements you need to define are shown in the following table:

Schema element	Description
`description`	Description of what the service call does. It also provides instructions about required input. You can write an extensive description, if desired, to help a guide or process designer understand what the service call step does.
`displayName`	Name that appears in the list of service calls steps in the guide and process designers.
`iconLocation`	Project path to an icon file that appears in the service call step in the guide. The size should be 32 x 32 pixels.
input parameters	The data used by this service.
`largeIconLocation`	Project path to an icon file that appears in the service call step properties dialog, if available. Otherwise, the smaller icon or default is used.
output fields and options	Output fields are name/value pairs where name is in the attribute name and value is the data under the element.
service name	The name of the service, which can be the real service name of a deployed process. If it is different, define an `avosServiceName` value.
`showForObject`	For future implementation.

Here is an example of a services.xml file

```
<tns:screenflowContribution
       xmlns:tns="http://schemas.active-endpoints.com/
          appmodules/screenflow/2010/10/avosServiceDiscovery.xsd">
   <tns:service name="HelloWorldAutomatedStep">
      <tns:displayName>Hello World</tns:displayName>
      <tns:description>Service Call step that accepts a user name
        and replies with a complex type value to greet the user.
```

```
        </tns:description>
        <tns:input>
           <tns:parameter name="Hello Name" type="string" required="true"/>
        </tns:input>
        <tns:output>
            <tns:field name="Greeting" type="TimeOfDayGreeting" />
        </tns:output>
     </tns:service>
  </tns:screenflowContribution>
```

Creating a Theme

When a guide runs, the default background colors, button styles, and other style elements are defined in HTML/CSS format in files deployed to the Process Server and made available to Guide Designer. You can modify the default theme or create new themes. In Guide Designer, the contributed themes are listed in the **Settings** dialog.

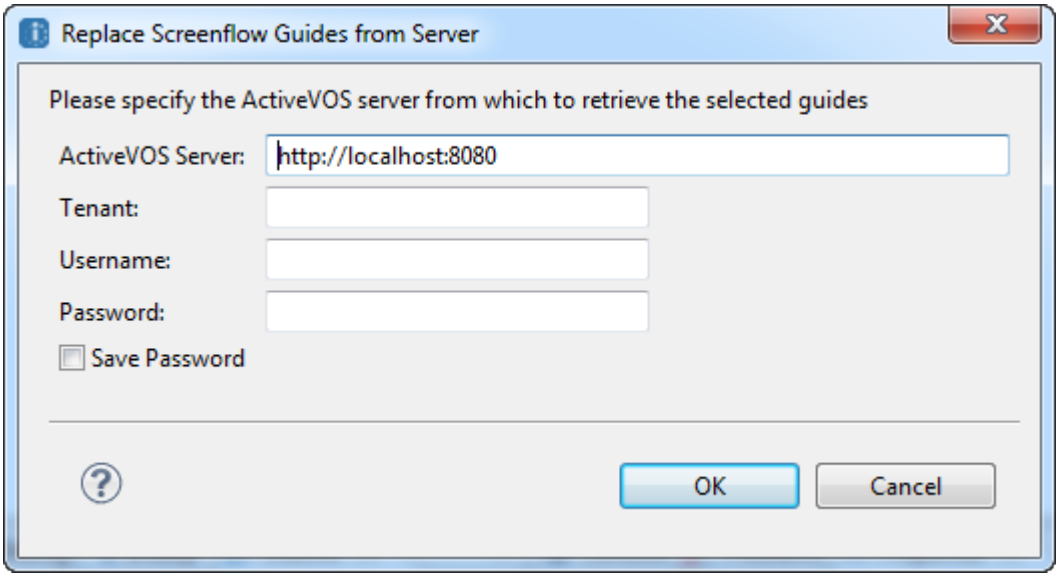
To create and deploy your own theme:

1. View a running or simulated guide to see the default look and feel.
2. In the guide designer, open the **Settings** dialog to see the default runtime and optional preview themes. Note that when you are trying out different themes at the guide level, you do not have to save the guide to see the new theme in action. Just start a simulation by clicking the **Simulate** button.
3. For ideas on what to add to a theme, view the Guide Designer Customization Starter orchestration project, which contains themes files.
4. Create your theme HTML/CSS files.
5. Create a themes.xml deployment descriptor. The default descriptor is in the Process Server catalog. The themes.xml schema is `avosThemeDiscovery.xsd`., which is located in the `com.activevos.socrates.repository.services\schema` Process Developer plugin. Refer to the documentation within the schema for hints on using images and descriptions.
6. Deploy the theme files and descriptor to the server as a deployment contribution. Refer to *Creating and Deploying a Business Process Archive Contribution* elsewhere in this help.

Replacing Guides From a Server Location

Guides within a project can be edited either on the server or within Process Developer.

When you are ready to replace or update a guide, select guides in the Project Explorer, right-click on it, then click **Replace Guides from Server**. Project Developer displays the following dialog:



After selecting this command, the existing guide is overwritten.

Project Files

Project Files

Create a new orchestration project, and on the *Create Project from Template* page, select **Guide Designer Demo**. The folders and files in this project are described in the following table.

Project Files	Description
bpel	
`CalendarReminder.bpel`	Creates a human task notification as the Set Reminder service call step in the Sell New Services guide. The input is the date entered in the preceding screen step. This illustrates the use of a People activity in conjunction with guides.
`CrunchbaseAPIUntyped.bpel`	Accepts input parameters such as `firstname`, `lastname`, and `company` and uses them to make a request to the CrunchBase API. It returns an XML response. If the search succeeded, it returns matched companies or persons. If it doesn't, the result is empty.
`CrunchbaseAutomatedStep.bpel`	Checks input parameters and if they are correct, it makes a request using `CrunchbaseAPIUntyped` and it returns the results.
`EmailSender.bpel`	Uses the Process Server `Email Service`.
`GoogleMap.bpel`	Displays a Google map as the Check area for service outage service call step in the Customer Connection Problems guide. The map is rendered using the power outages parameter, which is a complex type using the `googlemap-rendering.html` rendering discussed below.
`InstallAppointment.bpel`	Creates a human task as the Install Appointment service call step in the Sell New Services guide. This illustrates the use of a People activity in conjunction with guides.

Project Files	Description
LookupCustomerAddress.bpel	Creates the Look up customer address service call step in the Customer Connection Problems guide. This illustrates how to use input data in an service call step.
LookUpCustomerRecord.bpel	Creates the Look up customer record process used by the Customer Connection Problems guide. This illustrates how to use input data in an service call step.
LookUpServiceProvider.bpel	Creates the Look up provider by phone number service call step in the Sell New Services guide. Input is a phone number provided in the step's instructions. This illustrates how to use input data in an service call step.
PingModem.bpel	Creates the Ping modem service call step in the Customer Connection Problems guide. Input is a phone number provided in the step's instructions. This illustrates how to use input data in an service call step.
guides	
* Start Here	A tutorial guide describing the basics of creating a guide.
Crunchbase service test	A small guide that extracts information from the Crunchbase database. Use it to validate that Guide Designer is connecting to it.
Customer Connection Problems	A guide that diagnoses network connection problems for a customer by checking for service outages in the customer's area, pinging their modem, and doing other tests. This illustrates using several service call steps as well as other step types, including screen, jump to, and end steps.
Double check	A simple application used as an embedded guide in the Start Here tutorial.
Email and URL Links Demonstration	A guide that demonstrates how to incorporate email and to how to use an external service (this example lets you search Google from within a guide).
Installation Checklist	Creates an installation checklist that runs as a guide inside a human task. This shows an example of how to use the Embed Code for a published guide.
Sell New Services	Creates a sales discussion with a customer and uses the Create Appointment guide as an embedded guide. This illustrates the use of an embedded guide.
Guide Designer\renderings	
googlemap rendering files	The files in this area display a Google map in the Customer Connection Problems guide that is based on an address. These files contain options for displaying the map.
Guide Designer\service-contribs	
services.xml	Contains the service descriptor details for service call steps. This is a required file for service call step deployments.

Importing the Demo Guides into Guide Designer

Note: In previous releases, a service call step was called an automated step.

The guide designer uses the JIT spacetree technology to allow any user (not necessarily a developer) to create a web application called a "guide". Typically, a user is a domain expert who can design a tree-based

application for such applications as technical support calls, sales calls, or organizational business processes. A guide is an application with a single starting point, branching to steps that have different outcomes.

In the Guide Designer Demo project (see the Guide Designer Automated Step Demo Sample Project), there are several guides as well as their support files that you can import to examine how to implement a guide.

To import all demo files into Guide Designer:

1. Create the Service Call Step Service template, described in the Guide Designer Automated Step Demo Sample Project.
2. Start the server and deploy the BPR.
3. Log in to Process Central.
4. On the Guide Designer Home page, select Import and from the file system, import all guides from the `guides` folder in the sample project. You can import multiple guides in an archive file.
   Note: Ensure that you import all guides. Some guides have a dependency on others.

For additional details, select Help in the Process Designer.

CHAPTER 5

ActiveVOS WSHT API

Process Server implements the operations described by the OASIS WS-HumanTask (WSHT) API task client specifications to manage human tasks as well as interact with specific tasks. In addition to WSHT API, the Process Server also provides an extension API to the OASIS WSHT API with enhanced functionality. This document introduces the reader to the Informatica implementation of the WSHT and Extension APIs, and includes a few examples describing the use of the API via SOAP and indirectly via Java code using JAX-WS. You should read the `WS-HumanTask Architecture PDF document` to get an appreciation of WSHT. This document will help you understand concepts necessary for using the API.

Package Contents

The WSHT API using JAX-WS (WSHT4J) and samples code project contains the following artifacts:

Directory	Description
`dist/`	Contains the `wsht4j` API jar
`docs/`	Documentation
`lib/`	Dependencies and third party libraries. See also root directory `/common/lib`.
`orchestration/`	BPEL processes uses in the examples.
`src/`	JAX-WS generated code for the WSHT4J.
`src-examples/`	Sample code using the WSHT4J.
`build.xml`	Ant build script; use the latest supported JDK version to build this project
`.project .classpath`	Eclipse IDE project and classpath files.

HumanTask Related Web Service Endpoints

The follow table shows the Human Task related web service endpoints. Note that these endpoints require that the client application provide login credentials of the principal.

Name	Service Endpoint	Description
WSHT API	`http://host:port/active-bpel/services/AeB4PTaskClient-taskOperations` **Name**	WSHT API as described by `ws-humantask-api.wsdl`.
Process Server Extensions to WSHT API	`http://host:port/active-bpel/services/AeB4PTaskClient-aeTaskOperations`	Additional operations for interacting with human tasks. Note the service name ends with "aeTaskOperations" - the "ae" prefix indicating an Informatica-specific service.
WSHT Task Feeds	`http://host:port/active-bpel/services/REST/AeB4PTaskFeed`	The WSHT API `getMyTasks()` operation represented as a RSS or ATOM syndication feed.

Background and Setup

Before using the Web Services Human Task API, you should be familiar with the following concepts:

- Authentication and Using the Process Server Identity Service
- Development Setup
- Creating Human Tasks for Use with Samples
- Using SOAP-UI to Create Human Task Processes
- Using Web Forms to Create Human Task Processes

Authentication and Use of the Process Server Identity Service For Authorization

There are 4 basic components involved when interacting with tasks on the Process Server via service located at API endpoint `http://.../active-bpel/services/AeB4PTaskClient-taskOperations`.

```
 [Client: accesses /services/AeB4PTaskClient-taskOperations with username and password]
-->
  ----> [Web Application Container: checks if user has access to resources] -->
  ----> [ActiveVOS engine: execution of tasks and API] -->
  ----> [Identity Service: checks if user has access to tasks]
```

A Task Client is any application that wants to access the API via a web service call. The Inbox application is one such implementation. All clients access the WSHT API at the service endpoint located at `http://.../active-bpel/services/AeB4PTaskClient-taskOperations`.

The API service endpoint (`/services/AeB4PTaskClient-taskOperations`) is normally secured using standard Java servlet security. By default, this service is secured to provide access to a user with the role of `abTaskClient`. This security constraint is defined in the `active-bpel.warweb.xml` file. Anyone wishing to access this endpoint must provide authentication credentials and have the `abTaskClient` role.

It is the responsibility of the web application container (such as Tomcat) to make sure that the client accessing the service endpoint has proper credentials. As mentioned previously, any principal with role of

`abTaskClient` (as defined in the `web.xml`) should have access to the resources (service). If the credentials are not provided or incorrect, the web container normally responds with HTTP 401 (unauthorized).

Once access to service endpoint is granted (by the web container), the request along with the principal's username is passed into the Process Server engine's task client API implementation. At this point, the engine consults a ACL and the Identity Service to see which tasks the principal has access to. The principal must have a role associated with the tasks (generic human role such as a potential owner, business administrator) in order to view or manipulate tasks.

Note: The user credentials (such as via HTTP BASIC authentication and session cookies thereafter) are required for every web service request. Otherwise you will get an empty response (same as an anonymous user would obtain). In most cases, this is the reason for getting empty responses (as evidenced by a lack of task in the Process Central's Inbox task list). Another reason could be that the task has been escalated and assigned to another user/admin and therefore not accessible to this user.

In summary, the web container is responsible for authentication and authorizing access only to the service endpoint. The Identity Service is then used to determine what tasks (if any) the user is authorized to view and or modify.

Development Setup

In order to interact with the WSHT API using web services, make sure that the following are configured on your Process Server:

1. Process Sever Identity Service is enabled and configured. For example, LDAP/Active Directory.

   If you are using the evaluation version, you can use the default settings, which is to use an XML file (similar `tomcat-users.xml`) that contains list of roles/groups and users:

Username	Password	Roles
loanrep1	loanrep1	loanreps, abTaskClient
loanrep2	loanrep2	loanreps, abTaskClient
loanrep3	loanrep3	loanreps, abTaskClient
loanmgr1	loanmgr1	loanmgrs, abTaskClient
loanmgr2	loanmgr2	loanmgrs, abTaskClient
loancsr1	loancsr1	loancsrs, abTaskClient

   Note: the task users (for example, loanrep1) also have the `abTaskClient` role in order access the secured service endpoint.

2. The Web service endpoint to access the task operations is secured (for example, using HTTP Basic authentication). This can be done by un-commenting the appropriate section in the Process Server's `web.xml` file (located in `active-bpel.war)`. Here's an example of what the `active-bpel.war web.xml` file looks like:

```
<security-constraint>
 <web-resource-collection>
  <web-resource-name>WS-HT BPEL for People Client</web-resource-name>
  <description>Endpoint that contains all of the operations for an WS-HT BPEL for
People client
    application. These applications will provide an interface for people to interact
with tasks that
```

```
        were created by people activities within a BPEL process.</description>
      <url-pattern>/services/AeB4PTaskClient-taskOperations</url-pattern>
      <url-pattern>/services/AeB4PTaskClient-aeTaskOperations</url-pattern>
      <url-pattern>/taskxsl/*</url-pattern>
      </web-resource-collection>
       <!-- TASKCLIENT Uncomment to restrict access to the task client services
          <auth-constraint>
             <role-name>abTaskClient</role-name>
          </auth-constraint>
          TASKCLIENT
        -->
    </security-constraint>
```

Note: The Web container's authentication provider (for example, `tomcat-users.xml` in Tomcat file based provider) must use the same data (that is, list of usernames and roles) as does the Identity Service.

Creating Human Tasks for Use with Samples

In order to use the samples described in this document, you will need to create some tasks on the Process Server. The samples uses a bare-bones version of the Loan Approval process that is shipped with the Process Developer. The BPEL process that implements this simplified Loan Approval process is located in `/orchestration/HumanTaskDemo/bpel/humantaskdemo.bpel`.

This process must be deployed to the server by using the BPR file located at `/orchestration/HumanTaskDemo/deploy/humantaskdemo.bpr`.

This process simply takes a `<loan:loanProcessRequest/>` (using the `loanRequest.xsd` schema) element as the create message and passes it onto a human task. The SOAP message looks like:

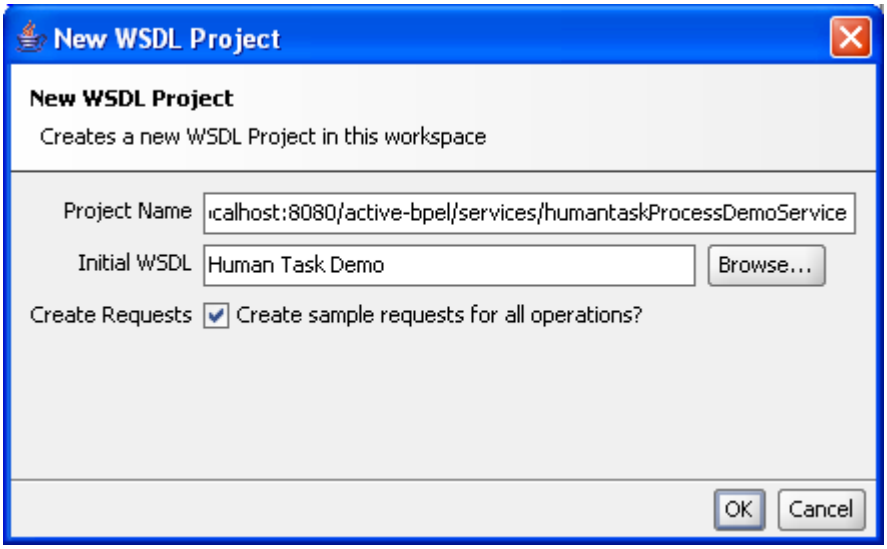
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:loan="http://schemas.active-endpoints.com/sample/LoanRequest/2008/02/
loanRequest.xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <loan:loanProcessRequest>
         <loan:loanType>Automobile</loan:loanType>
         <loan:firstName>John</loan:firstName>
         <loan:lastName>Smith</loan:lastName>
         <loan:dayPhone>2039299400</loan:dayPhone>
         <loan:nightPhone>2035551212</loan:nightPhone>
         <loan:socialSecurityNumber>123-45-6789</loan:socialSecurityNumber>
         <loan:amountRequested>15000</loan:amountRequested>
         <loan:loanDescription>Application to finance the purchase of a Toyota Prius</
loan:loanDescription>
         <loan:otherInfo>Down payment is US$7500</loan:otherInfo>
         <loan:responseEmail>john.smith@example.com</loan:responseEmail>
      </loan:loanProcessRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Before running any of the examples, you should create one or more tasks by sending the create messages to the humantaskdemo's service endpoint located at `http://localhost:8080/active-bpel/services/humantaskProcessDemoService`. You can send these messages using two methods: SOAP-UI or a REST based web form.

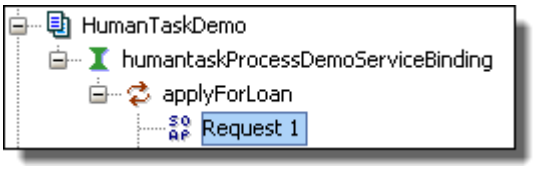
Using SOAP-UI to Create Human Task Processes

Download and install SOAP-UI. SOAP-UI can be downloaded from `http://www.soapui.org/`. After installing SOAP-UI:

1. Choose **File > New WSDL Project**.
   a. Enter a name for the project; for example: HumanTaskDemo.
   b. For the initial WSDL, enter `http://localhost:8080/active-bpel/services/humantaskProcessDemoService`. Make sure that Create Request check box is checked.
   c. Press **OK.**



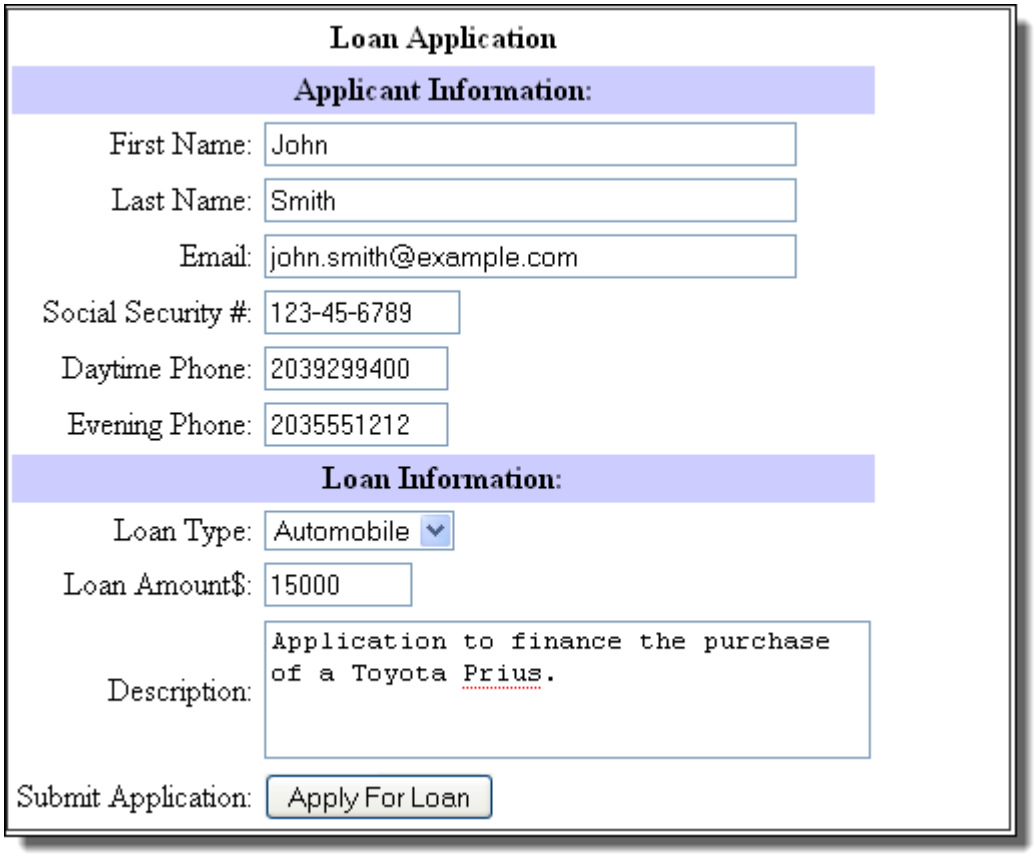
2. Save the SOAP-UI project.
3. Select project from the Project Explore; expand the port type associated with the `humantaskProcessDemoServiceBinding`. Then double click on the Request to see the sample create message.



4. Populate the request and send the message to the server.

Using Web Form to Create Human Task Processes

You can use the user friendly HTML form to send create messages to the `humantaskdemo` process using a second REST based process.



1. Open the HTML document located in the `/docs/InvokeLoanApproval-REST-form.html` using a web browser.
2. Fill out the Loan Application web form and press the **Apply for Loan** button.
3. Repeat to send more create messages.

You can verify that tasks were created by logging into the Inbox using the credentials for loanrep1.

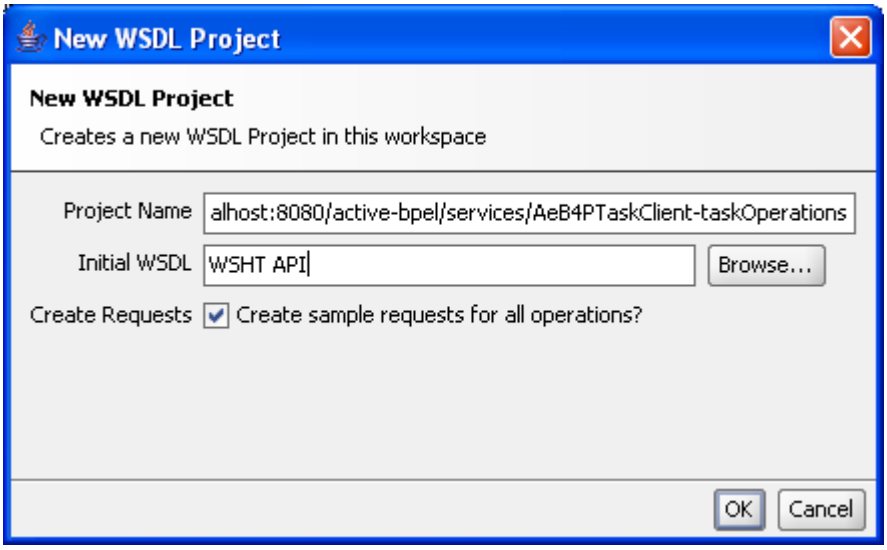
Interacting with WSHT API

The topics below discuss how you interact with the Web Services Human Task (WSHT) API.

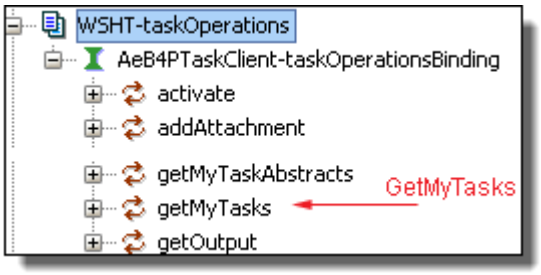
Accessing List of Tasks Using SOAP UI

To quickly see the WSHT API in action, you can use SOAP-UI or similar tool (such as the Eclipse's Web Services Explorer tool).

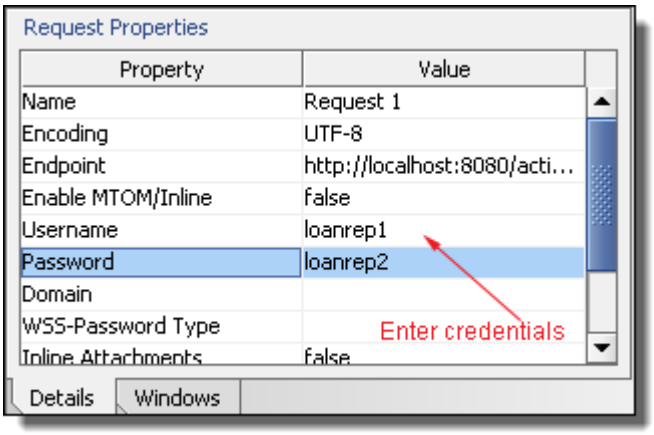
To use SOAP-UI, create a new WSDL Project in SOAP-UI using the endpoint address of the WSHT API service for the initial WSDL: `http://localhost:8080/active-bpel/services/AeB4PTaskClient-taskOperations`. Save the project after SOAP-UI has created the sample requests.



From SOAP-UI's Project Explorer, locate the project for the WSHT API and expand the `getMyTasks` operation and double click on the request to reveal its sample request.



Enter the username and password (for example, `loanrep1/loanrep1`) for this request using the Request Properties window shown in the bottom left of SOAP-UI application:



Send the following SOAP request:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:xsd="http://www.example.org/WS-HT/api/xsd">
   <soapenv:Header/>
   <soapenv:Body>
     <htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
       <htdt:taskType>TASKS</htdt:taskType>
```

```
         <htdt:genericHumanRole>POTENTIAL_OWNERS</htdt:genericHumanRole>
         <htdt:status>READY</htdt:status>
         <htdt:maxTasks>5</htdt:maxTasks>
      </htdt:getMyTasks>
   </soapenv:Body>
</soapenv:Envelope>
```

This request fetches all unclaimed (status = READY) tasks that the current user can access. The response should contain one or more `<htdt:taskAbstract>` elements representing each task.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <htdt:getMyTasksResponse
            xmlns:htdt="http://www.example.org/WS-HT/api/xsd"
            xmlns:htda="http://www.example.org/WS-HT/api">
         <htdt:taskAbstract>
            <htda:id>urn:b4p:5</htda:id>
            <htda:taskType>TASK</htda:taskType>
            <htda:name>ApproveLoan</htda:name>
            <htda:status>READY</htda:status>
            <htda:priority>2</htda:priority>
            <htda:taskInitiator>anonymous</htda:taskInitiator>
            <htda:potentialOwners>
               <htd:users xmlns:htd="http://www.example.org/WS-HT">
                  <htd:user>loanrep1</htd:user>
                  <htd:user>loanrep2</htd:user>
               </htd:users>
            </htda:potentialOwners>
            <htda:businessAdministrators>
               <htd:groups xmlns:htd="http://www.example.org/WS-HT">
                  <htd:group>loanmgr1</htd:group>
               </htd:groups>
            </htda:businessAdministrators>
            <htda:createdOn>2009-04-23T19:12:21.615Z</htda:createdOn>
            <htda:createdBy>anonymous</htda:createdBy>
            <htda:activationTime>2009-04-23T19:12:21.615Z</htda:activationTime>
            <htda:isSkipable>true</htda:isSkipable>
            <htda:hasPotentialOwner>true</htda:hasPotentialOwners>
            <htda:startByExists>false</htda:startByExists>
            <htda:completeByExists>false</htda:completeByExists>
            <htda:presentationName>Loan Approval Demo</htda:presentationName>
            <htda:presentationSubject>Loan request for US$ 15000 from John Smith
                        [taskid: urn:b4p:5]</htda:presentationSubject>
            <htda:renderingMethodExists>false</htda:renderingMethodExists>
            <htda:hasOutput>false</htda:hasOutput>
            <htda:hasFault>false</htda:hasFault>
            <htda:hasAttachments>false</htda:hasAttachments>
            <htda:hasComments>false</htda:hasComments>
            <htda:escalated>false</htda:escalated>
            <htda:primarySearchBy>123-45-6789</htda:primarySearchBy>
         </htdt:taskAbstract>
      </htdt:getMyTasksResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

You will need the `resultset` value for other task operations as it has the task ID, which is within the `<htda:id>task identifier</htda:id>` element. For example, to claim this task, the SOAP message is:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsd="http://www.example.org/WS-HT/api/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <htdt:claim xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
         <htdt:identifier>urn:b4p:62</htdt:identifier>
      </htdt:claim>
   </soapenv:Body>
</soapenv:Envelope>
```

Refer to the WSHT API WSDL, schemas, and documentation for information on other operations.

Parameters Used In GetMyTasks Request

The previous example showed how to get a list of tasks using the `getMyTasks` operation. The basic request message is:

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
   <htdt:taskType/>
   <htdt:genericHumanRole/>
   <htdt:workQueue/>
   <htdt:status/>
   <htdt:whereClause/>
   <htdt:createdOnClause/>
   <htdt:maxTasks/>
</htdt:getMyTasks>
```

The key elements in this request are:

Element	Possible Values	Notes
`<htdt:taskType/>`	- TASKS - NOTIFICATIONS - ALL	Indicates whether to return tasks, notifications, or both
`<htdt:genericHumanRole/>`	- POTENTIAL_OWNERS - OWNER - ADMINISTRATORS - NOTIFICATION_RECIPIENTS - INITIATOR - STAKEHOLDERS	Optional element. If this element is not provided, the server uses POTENTIAL_OWNER (and implicitly the OWNER)
`<htdt:workQueue/>`	group/role name	Optional element. Name of group/role. The user must be a member of the group.
`<htdt:status/>`	- READY (unclaimed - ready to be claimed) - RESERVED (claimed) - IN_PROGRESS (started) - SUSPENDED - EXITED - FAILED - ERROR - COMPLETED - OBSOLETE	This is a repeating element. Include more than one to simulate "OR" across two or more values. For example, including the following two elements filters the result set to tasks that are claimed (RESERVED) or started (IN_PROGRESS) by current user: `<htdt:status>IN_PROGRESS</htdt:status>` `<htdt:status>RESERVED</htdt:status>`
`<htdt:whereClause/>`	Where clause string.	Optional element. Filter by using a simple where clause. For example, - By taskId: "Task.ID = urn:b4p:8765309" // should only return 1 task. - By owner: "Task.Owner = jenny" // All tasks owned by jenny (reserved, in_progress, completed, failed) - By priority: "Task.Priority = 3" - By search by: "Task.PrimarySearchBy = customerid-8765309"

Element	Possible Values	Notes
<htdt:createdOnClause/>		Optional Element. Filter by the date task was created. For example, Task.CreatedOn = 2009-10-05T11:14:00Z The date should be a schema dateTime (xsd:dateTime) formatted value.
<htdt:maxTasks/>	Maximum number of tasks to be returned. Integer value (> 0).	Optional Element.

Sample GetMyTasks Requests

The following sections describe some of the sample request elements for getMyTasks operation:

- List all unclaimed tasks (that can be claimed/started by a potential owner)
- List tasks that have been claimed by current principal(owner)
- List all my (owned by principal) tasks that are in progress (started)
- List all closed tasks owned by current principal
- List all open tasks (ready, reserved, in progress and suspended) that are accessible by the task initiator
- List tasks that are claimed (reserved) where the principal is a business administrator of the task
- List all notifications

List all unclaimed tasks (that can be claimed/started by a potential owner)

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
    <htdt:taskType>TASKS</htdt:taskType>
    <htdt:genericHumanRole>POTENTIAL_OWNERS</htdt:genericHumanRole>
    <htdt:status>READY</htdt:status>
    <htdt:maxTasks>20</htdt:maxTasks>
</htdt:getMyTasks>
```

List tasks that have been claimed by current principal (owner)

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
    <htdt:taskType>TASKS</htdt:taskType>
    <htdt:genericHumanRole>OWNER</htdt:genericHumanRole>
    <htdt:status>RESERVED</htdt:status>
    <htdt:maxTasks>20</htdt:maxTasks>
</htdt:getMyTasks>
```

List all my (owned by principal) tasks that are in progress (started)

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
    <htdt:taskType>TASKS</htdt:taskType>
    <htdt:genericHumanRole>OWNER</htdt:genericHumanRole>
    <htdt:status>IN_PROGRESS</htdt:status>
    <htdt:maxTasks>20</htdt:maxTasks>
</htdt:getMyTasks>
```

List all closed tasks owned by current principal

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
    <htdt:taskType>TASKS</htdt:taskType>
    <htdt:genericHumanRole>OWNER</htdt:genericHumanRole>
```

```
    <htdt:status>EXITED</htdt:status>
    <htdt:status>FAILED</htdt:status>
    <htdt:status>ERROR</htdt:status>
    <htdt:status>COMPLETED</htdt:status>
    <htdt:status>OBSOLETE</htdt:status>
    <htdt:maxTasks>20</htdt:maxTasks>
</htdt:getMyTasks>
```

List all open tasks (ready, reserved, in progress and suspended) that are accessible by the task initiator (<htdt:genericHumanRole/> is 'INITIATOR')

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
    <htdt:taskType>TASKS</htdt:taskType>
    <htdt:genericHumanRole>INITIATOR</htdt:genericHumanRole>
    <htdt:status>READY</htdt:status>
    <htdt:status>RESERVED</htdt:status>
    <htdt:status>IN_PROGRESS</htdt:status>
    <htdt:status>SUSPENDED</htdt:status>
    <htdt:maxTasks>20</htdt:maxTasks>
</htdt:getMyTasks>
```

List tasks that are claimed (reserved) where the principal is a business administrator of the task

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
    <htdt:taskType>TASKS</htdt:taskType>
    <htdt:genericHumanRole>ADMINISTRATORS</htdt:genericHumanRole>
    <htdt:status>RESERVED</htdt:status>
    <htdt:maxTasks>20</htdt:maxTasks>
</htdt:getMyTasks>
```

List all notifications

```
<htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
    <htdt:taskType>NOTIFICATIONS</htdt:taskType>
    <htdt:genericHumanRole>NOTIFICATION_RECIPIENTS</htdt:genericHumanRole>
    <htdt:maxTasks>20</htdt:maxTasks>
</htdt:getMyTasks>
```

Getting Task Input and Output Data

Most of task specific operations such as claim, start, stop etc. are simple and they require only the task id element. You may refer to the WSHT documentation and `ws-humantask-api.wsdl` for further details.

The `getInput`, `getOutput`, and `setOutput` operations require the task ID as well as the input (output) part name. For example, the `humantaskdemo.wsdl approveLoanWshtPT` port type `approveLoan` operation is described as follows:

```
<portType name="approveLoanWshtPT">
   <operation name="approveLoan">
   <input message="tns:WshtLoanInput" />
   <output message="tns:WshtLoanOutput" />
   </operation>
</portType>
```

Note: The above shows the port type for the Human Task used by the People Activity (which is not the same as the port type used by the `humantaskdemo` BPEL process).

The input message is defined as:

```
<message name="WshtLoanInput">
   <part name="request" element="loan:loanProcessRequest" />
</message>
```

Note the message part name is `request`. The message part name is included in the WSHT API `getInput` message:

```
<htdt:getInput xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
   <htdt:identifier>urn:b4p:5</htdt:identifier>
   <htdt:part>request</htdt:part>
</htdt:getInput>
```

Similarly, the output message part is named `response`:

```
<message name="WshtLoanOutput">
   <part name="response" element="loan:loanApprovalResponse" />
</message>
```

The `getOuput` is:

```
<htdt:getOutput xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
   <htdt:identifier>urn:b4p:5</htdt:identifier>
   <htdt:part>response</htdt:part>
</htdt:getOutInput>
```

The `setOutput` has an additional element `<htdt:taskData>` which contains the output message part element. For example:

```
<htdt:setOutput xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
   <htdt:identifier>urn:b4p:5</htdt:identifier>
   <htdt:part>response</htdt:part>
   <htdt:taskData>
     <!-- loanApprovalResponse element (output) -->
     <loan:loanApprovalResponse xmlns:loan=
           "http://schemas.active-endpoints.com/sample/LoanRequest/2008/02/
loanRequest.xsd">
       <loan:responseToLoanRequest>approved</loan:responseToLoanRequest>
       <loan:responseDescription>Your loan has been approved.</loan:responseDescription>
     </loan:loanApprovalResponse>
   </htdt:taskData>
</htdt:setOutInput>
```

Configuring a GetMyTasks Filter with a whereClause

For an overview of this topic, see *Configuring Task Role Filters* elsewhere in this help.

In the `Task Role Filters` section of the `.avcconfig` file, you can add a filter to apply to the tasks in one of the Show drop-down entries. For example, you can create a filter for Unclaimed Tasks to show only Priority zero (highest priority).

To filter the task list for the Show filter:

Modify the default configuration in the `<avccom:filter/>` section of the `<avccom:taskFilterDef/>` section.

The following code snippet shows the default configuration:

```
<avccom:filter>
  <tsst:getTasks
       xmlns:tsst="http://schemas.active-endpoints.com/b4p/wshumantask/
             2007/10/aeb4p-task-state-wsdl.xsd">
   <htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
     <htdt:taskType>TASKS</htdt:taskType>
     <htdt:genericHumanRole>POTENTIAL_OWNERS</htdt:genericHumanRole>
     <htdt:status>READY</htdt:status>
     <htdt:whereClause>Task.Name = 'ApproveLoan'</htdt:whereClause>
     <htdt:maxTasks>20</htdt:maxTasks>
   </htdt:getMyTasks>
   <tsst:taskIndexOffset>0</tsst:taskIndexOffset>
```

```
    </tsst:getTasks>
  </avccom:filter>
```

Filtering a task list by parameter via WS-HT `whereClause`

You can use the following operators in the `whereClause`:

- >
- >=
- <
- <=
- =
- like
- and
- or
- ()

For a complete list of task column names, task types, generic human roles and other WS-HT specifications, you can work with the SDK. To get started, see *What is the Process Server WS-HumanTask API?* elsewhere in this help.

The following examples show various `whereClause` syntax. Note that the Booleans `and` and `or` are case-sensitive.

Example 1

The following example shows a sample `whereClause` using a standard WS-HT task property name. For a list of standard WS-HT names, see *WS-HT Task Property List* elsewhere in this help.

```
<htdt:whereClause>Task.Name = 'ApproveLoan'</htdt:whereClause>
```

Example 2

This example combines filter properties.

```
Task.Priority &lt; 3 and Task.Owner = 'loanrep1'
```

Example 3

This example also combines filter properties.

```
Task.PaProcessId = 10 and(Task.Escalated = true or Task.Priority = 0)
```

Example 4

The following example shows using presentation parameters (that is, custom task properties).

```
Task.Name = 'LoanApproval' and loanAmount &gt; 10000 and zipCode = '06484'
```

For details, see `Creating Custom Task Properties` elsewhere in this help.

Using a getMyTasks orderBy Element

You can sort the results returned by `getTasks` by setting the `orderBy` element. The fields name you can use are:

- Created
- Expiration
- Modified
- Name

- Owner
- PresentationName
- PresName
- PresSubject
- Priority
- State
- Summary

The following example sorts the results with the most recent first (that is, in descending order), followed by the priority:

```
orderBy orderBy = new OrderBy();
orderBy.getFieldId().add("-Created");
orderBy.getFieldId().add("Priority");
```

Notice the minus sign ("-") being used to set the order to descending.

In XML, this same example is written as follows:

```
<ns:orderBy>
     <ns:fieldId>-Created</ns:fieldId>
     <ns:fieldIPdriority</ns:fieldId>
</ns:orderBy>
```

Process Server Extensions to WSHT API

Task data has the following information:

- Identifier (task ID)
- Name (name of task).
- Type (TASK or NOTIFICATION).
- Presentation information such as presentation name, subject, description, and so on
- Context data such as potential owners, business administrators, actual owner, creation dates, and other meta data
- Operational data such as the input for example, loan approval name, email, loan amount), output, comments, attachments, and others

The current version of the WSHT API (at endpoint `/services/AeB4PTaskClient-taskOperations`) allows you to access all of these data by making multiple web service calls. For example, if you have an application that needs to populate the user interface with the task presentation name, input data, output data, and comments, you will have to make discrete web service calls to operations using the following functions:

- `getTaskInfo()`
- `getTaskDescription()`
- `getInput()`
- `getOutput()`
- `getComments()`

Alternate method to retrieve all of the task data using a single web service call is the Process Server `getInstance` operation available from the `aeTaskOperations` port type at endpoint `http://host:port/active-bpel/services/AeB4PTaskClient-aeTaskOperations`. This operation returns a single element

<trt:taskInstance> described in the aeb4p-task-rt.xsd schema. It contains all of the task data, including input data, output data, comments, attachments, presentation information, and the like. (The aeTaskOperations port type is described in aeb4p-task-state.wsdl). The <trt:taskInstance> element also contains some additional useful meta data such as the list of operations the current principal can invoke given the task's current status and the principal's role.

See the XML files at /docs/ae-getTaskInstance-request.xml and /docs/ae-getTaskInstance-response.xml for a detailed request and response example within your Process Developer installation's com.activevos.com plugin directory

The complete list of extension operations provided by the /services/AeB4PTaskClient-aeTaskOperations is shown in the following table:

Process Server Extension Operation	Description
authorize	Verifies that the principal can access the service endpoint.
deleteAttachmentById	Deletes an attachment using its attachment ID. Process Server assigns an ID for each attachment that is added to a task..The WSHT API does not have an operation for deleting attachments.
getAttachmentById	Retrieves attachment by ID. (The WSHT API identifies attachments by their name instead of by unique IDs.)
deleteComment	Deletes a comment using its comment ID. The Process Server assigns an ID to each comment that is added to the task. The WSHT API does not have an operation to delete comments.
updateComment	Updates a comment. This operation is not provided in the standard WSHT API.
getInstance	Returns full details of a task given a task ID. This operation is useful when you need the complete information about a task with out having to make separate WSHT API calls to get the information.
getTasks	Provides index offset (for pagination), column ordering, and the like. This is an extension to the WSHT getMyTasks task listing operation.

Using JAX-WS Based ActiveVOS WSHT4J API

The Process Server wsht4j API is a WSHT API client based on JAX-WS (version 2.1.5). To use the client API, include the avos-wsht4j.jar along with the dependent libraries (for JAX-WS) in your classpath. The JAX-WS dependent jars are:

Jar File	Comment
activation.jar	Java Activation Framework
jaxb-api.jar	JAX Binding
jaxb-impl.jar	

Jar File	Comment
jaxws-api.jar	JAX-WS
jaxws-rt.jar	
jsr173_api.jar	Streaming API for XML
jsr181-api.jar	Web Services meta data
mimepull.jar	
resolver.jar	
saaj-api.jar	SAAJ
woodstox.jar	StAX-compliant (JSR-173) Open Source XML-processor

Building WSHT4J

You can build the API using the provided ant build script. The built `avos-wsht4j.jar` is copied to the `dist/` directory. Use the latest supported JDK version to build this project.

WSHT4J Samples

Samples using the wsht4j API can be found under the `/src-examples/` source tree.

Listing Tasks Using WSHT4J API

The following sample snippet shows how to list unclaimed tasks using the wsht4j API. See `com.activevos.examples.wsht.GetMyTasksDemo1` (and `GetMyTasksDemo2`) class for the complete examples.

```
import com.activevos.api.humantask.AeB4PTaskClientTaskOperations;
import com.activevos.api.humantask.TaskOperations;
import com.activevos.api.humantask.wsht.htda.TGenericHumanRoleType;
import com.activevos.api.humantask.wsht.htda.TStatus;
import com.activevos.api.humantask.wsht.htda.TTask;
import java.util.ArrayList;
import java.util.List;
import javax.xml.ws.BindingProvider;
// Define endpoint, username and password
String serviceUrl =
    "http://localhost:8080/active-bpel/services/AeB4PTaskClient-taskOperations";
String username = "loanrep1";
String password = "loanrep1";
// Create service and get service port.
AeB4PTaskClientTaskOperations wshtService =
    new AeB4PTaskClientTaskOperations();
TaskOperations wshtServicePort = wshtService.getAeB4PTaskClientTaskOperationsPort();
// Set request properties such as binding endpoint and BASIC credentials.
((BindingProvider)wshtServicePort).getRequestContext().
    put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, serviceUrl);
((BindingProvider)wshtServicePort).getRequestContext().
    put(BindingProvider.USERNAME_PROPERTY, username);
((BindingProvider)wshtServicePort).getRequestContext().
    put(BindingProvider.PASSWORD_PROPERTY, password);

// Task type. In this case we are looking for the tasks (and not notifications).
// In general, the taskType is 'TASKS' for all cases unless you want notifications
// in the response. For notifications, use the value 'NOTIFICATIONS'. If you want
// tasks and notifications, use 'ALL'.
```

```
String taskType = "TASKS";

// Set role to be potential owners. That is, tasks that are available
// to potential owners.
TGenericHumanRoleType role = TGenericHumanRoleType.POTENTIAL_OWNERS;
    // workQueue - not used.
String workQueue = null;

// Task status - this is one of the main filters we use to narrow down the
// resultset. Unclaimed tasks have a status of 'READY'.
// Some other common status enumerations are claimed (RESERVED),
// started (IN_PROGRESS), failed (FAILED), and completed (COMPLETED).
List<TStatus> statuses = new ArrayList<TStatus>();
statuses.add(TStatus.READY); // READY = 'unclaimed'.
    // whereClause - not used.
String where = null;
    // createdOn - not used.
String createdOnClause = null;

// maximum number of tasks to be returned in the resultset.
int maxtasks = 5;
// Call getMyTasks
List<TTask> tasks = wshtServicePort.getMyTasks(taskType, role, workQueue, statuses,
where,
      createdOnClause, maxtasks);

// Note: TTask is of complex type tTask defined in ws-humantask-api-wsdl.xsd schema.
int count = tasks.size();
System.out.println("Number of tasks: " + count);
if ( count == 0 )
{
   // No tasks. Normally, this happens if:
   // - The username (defined in PRINCIPAL_USERNAME) does not have access to any tasks
   // - There are no unclaimed tasks (that is, tasks with status = READY).
   return;
}
//
// Get 1st task in the list and print some information.
//
TTask task = tasks.get(0);

// The taskId is unique and normally takes for form "urn:b4p:NNN" where NNN is some
number
// (usually representing the process id that implements the task on the ActiveVOS
server).
// The taskId is required for all task related operations.
String taskId = task.getId();
System.out.println("TaskID: " + taskId);

// print some additional information:
System.out.println("Name: " + task.getName());// task QName
System.out.println("PresentationName: " + task.getPresentationName());
System.out.println("PresentationSubject: " + task.getPresentationSubject());

// Expected value is READY (unclaimed).
System.out.println("Status: " + task.getStatus().toString());
```

Claiming a Task Using WSHT4J API

The following sample snippet shows how to claim task using the wsht4j API. Look at the `com.activevos.examples.wsht.BasicOperations` class for the complete example showing how to claim, start, set the output, and complete a task .

```
// Service port
TaskOperations wshtServicePort = null;

//
// Code to create wshtServicePort similar to previous example
```

```
//
// Id of task we want to claim
String taskId = "urn:b4p:5";
try
{
   // claim
   wshtServicePort.claim(taskId);
}
catch(IllegalArgumentFault illegalArgumentFault)
{
   //Fault due to invalid task id.
   System.out.println("IllegalArgumentFault: " + illegalArgumentFault.getFaultInfo());
}
catch(IllegalStateFault illegalStateFault)
{
   // Illegal state. For example, tried to claim a task that was not in a
   // a READY state (unclaimed).
   IllegalState illegalState = illegalStateFault.getFaultInfo();
   // Sample:
   // illegalState.getStatus():  'RESERVED'  (already claimed)
   // illegalState.getMessage(): 'Task must be in the READY state to be claimed.'
   System.out.println("IllegalStateFault: "
            + "\n taskStatus=" + illegalState.getStatus()
            + "\n " + illegalState.getMessage());
}
```

Additional Examples Using WSHT4J

Operation	Example Class
getMyTasks (various examples)	com.activevos.examples.wsht.GetMyTasksDemo1 com.activevos.examples.wsht.GetMyTasksDemo2
Quick overview of claim, start, setOuput and complete.	com.activevos.examples.wsht.BasicOperationsDemo
getTaskInfo	com.activevos.examples.wsht.operations.GetTaskInfoAndDescriptionDemo
getTaskDescription	com.activevos.examples.wsht.operations.GetTaskInfoAndDescriptionDemo
claim	com.activevos.examples.wsht.operations.ClaimStartStopReleaseDemo
start	com.activevos.examples.wsht.operations.ClaimStartStopReleaseDemo
stop	com.activevos.examples.wsht.operations.ClaimStartStopReleaseDemo
release	com.activevos.examples.wsht.operations.ClaimStartStopReleaseDemo
setPriority	com.activevos.examples.wsht.operations.SetPriorityDemo
getInput	com.activevos.examples.wsht.operations.GetInputDemo
getOutput	com.activevos.examples.wsht.operations.GetSetOutputDemo
setOutput	com.activevos.examples.wsht.operations.GetSetOutputDemo
deleteOutput	com.activevos.examples.wsht.operations.GetSetOutputDemo
addComments	com.activevos.examples.wsht.operations.CommentsDemo

Operation	Example Class
getComments	com.activevos.examples.wsht.operations.CommentsDemo
addAttachment	com.activevos.examples.wsht.operations.AttachmentsDemo
deleteAttachments	com.activevos.examples.wsht.operations.AttachmentsDemo
getAttachmentInfos	com.activevos.examples.wsht.operations.AttachmentsDemo
getAttachments	com.activevos.examples.wsht.operations.AttachmentsDemo
delegate (assign)	com.activevos.examples.wsht.operations.DelegateDemo
forward	com.activevos.examples.wsht.operations.ForwardDemo
suspend	com.activevos.examples.wsht.operations.SuspendResumeDemo
resume	com.activevos.examples.wsht.operations.SuspendResumeDemo
complete	com.activevos.examples.wsht.BasicOperationsDemo
deleteFault	
setFault	
activate	
fail	
getRenderingTypes	com.activevos.examples.wsht.operations.RenderingsDemo
getRendering	com.activevos.examples.wsht.operations.RenderingsDemo
nominate	
remove	
setGenericHumanRole	
skip	

Getting Task Lists As an RSS or ATOM Feed

The list of tasks can be accessed as a RSS or ATOM feed using the REST based service located at `http://host:port/active-bpel/services/REST/AeB4PTaskFeed`. This endpoint also requires the principal username and password.

The `AeB4PTaskFeed` service implements the WSHT `getMyTasks()` operation. The request parameters should passed as part of the HTTP GET request query string:

Parameter Name	Value(s)	Description
format	rss or atom	Optional. The response feed format. Default value is atom.
role	- user - administrator - initiator - stakeholder	Optional. List tasks on behalf of a task role. The default value is user (which is equivalent to potential owners and actual owner).
filter	- open - unclaimed - reserved - reserved_started - started - suspended - closed - completed - failed - exited - error - obsolete	Optional. The default value is open, which mean list all open tasks (that is, tasks with status = READY, RESERVED, IN_PROGRESS or SUSPENDED).
maxTasks	integer	Optional. Number of tasks to return in the feed. The default value is 20.
taskIndexOffset	non negative integer	Optional. Index offset used for pagination. The default value is 0.
searchBy	percent encoded string	Optional search by string value.

Example 1

Lists all open tasks that are accessible to the user. Returns the ATOM response

```
http://localhost:8080/active-bpel/services/REST/AeB4PTaskFeed
```

Example 2

As above, but returns the RSS response. Note the `format=rss` query string value in the request.

```
http://localhost:8080/active-bpel/services/REST/AeB4PTaskFeed?format=rss
```

Example 3

Lists tasks that the user is currently working on (IN_PROGRESS). Returns its response as a RSS feed. Note the `filter=started` value in the query string.

```
http://localhost:8080/active-bpel/services/REST/AeB4PTaskFeed?filter=started&format=rss
```

Recommended External Tools

If you are writing your own client code (Java, .NET, and so on), you could use an excellent debugging tool called Charles (free download at `http://www.charlesproxy.com/`). Charles is a web/reverse proxy that lets you monitor XML traffic between the client (your code) and the server (Informatica WSHT).

CHAPTER 6

Embedding Request Forms in Standalone Web Pages

Process Server exposes all its processes and services using XML and JavaScript Object Notation (JSON) bindings in addition to SOAP, REST and JMS. The XML (and JSON) binding makes it easy for application developers already familiar with XML/JSON-based REST application development to invoke processes and obtain responses from them.

Using XML or JSON bindings frees developers from having to obtain and learn SOAP libraries to build applications that leverage Process Server service-based processes. This approach allows JavaScript developers to use various libraries such as jQuery to build process-enabled applications.

Process Server uses the JSON binding for Process Central request and task forms. This implementation provides functions that developers can use for process-enabled application projects. Examples of the use of these functions are provided in this section of the help.

Service Endpoints

The following table shows the endpoints for services exposed by Process Server using various bindings. The ServiceName indicates the name of the service - LoanApprovalService for example.

Binding	Service Endpoint	Description
SOAP 1.1	`http://host:port/active-bpel/services/ServiceName`	Default SOAP endpoint for all services except for REST style services Note: The Process Server engine Administration API service (ActiveBpelAdmin) is available only at the SOAP 1.1 endpoint.
SOAP 1.2	`http://host:port/active-bpel/services/soap12/ServiceName`	SOAP 1.2 endpoint. All services are exposed through this endpoint except the REST and ActiveBpelAdmin services.
REST	`http://host:port/active-bpel/services/REST/ServiceName`	Endpoint for REST binding. It excludes the ActiveBpelAdmin service.

Binding	Service Endpoint	Description
XML	`http://host:port/active-bpel/services/XML/ServiceName`	Endpoint for simple XML binding. It excludes REST and ActiveBpelAdmin services.
JSON	`http://host:port/active-bpel/services/JSON/ServiceName`	Endpoint for JSON binding. it excludes REST and ActiveBpelAdmin services.

Note that the Process Server service API (ActiveBpelAdmin) is available only at the SOAP 1.1 address. This service is not available in SOAP 1.2, XML or JSON bindings.

XML Binding

Processes deployed to the Process Server are available as simple XML bindings in addition to the standard SOAP binding providing that the process and deployment is WS-I compliant:

- WSDL portType operation is a document-literal
- Operation input message has only one part and that part is of element type
- Operation output message has only one part and that part is of element type

Invoking a Process

To invoke a process using a XML binding endpoint, you need to:

1. Set the POST header content-type header to text/xml
2. Set the POST payload (body content) to be the request xml element
3. Include authorization headers if needed
4. Send an HTTP POST message to the service XML endpoint

The response from the POST will be an HTTP 200/OK response with a content-type of `text/xml`. The response body will contain the service's response XML. The following snippet shows the HTTP request used to invoke the Loan Approval process using the humantaskProcessDemoService service.

```
POST /active-bpel/services/XML/humantaskProcessDemoService HTTP/1.1
Content-Length: 710
Content-Type: text/xml; charset=UTF-8
Authorization: Basic YWVhZG1pbjphZWFkbWlu
Host: localhost:8080
<loan:loanProcessRequest xmlns:loan=
    "http://schemas.active-endpoints.com/sample/LoanRequest/2008/02/loanRequest.xsd">
 <loan:loanType>Automobile</loan:loanType>
 <loan:firstName>John</loan:firstNam>
 <loan:lastName>Smith</loan:lastName>
 <loan:dayPhone>2039299400</loan:dayPhone>
 <loan:nightPhone>2035551212</loan:nightPhone>
 <loan:socialSecurityNumber>123-45-6789</loan:socialSecurityNumber>
 <loan:amountRequested>15000</loan:amountRequested>
 <loan:loanDescription>Application to finance the purchase of a Toyota Prius</
loan:loanDescription>
 <loan:otherInfo>Down payment is US$7500</loan:otherInfo>
```

```
 <loan:responseEmail>john.smith@example.com</loan:responseEmail>
</loan:loanProcessRequest>
```

The response looks like:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Transfer-Encoding: chunked
Date: Wed, 10 Mar 2010 22:53:40 GMT
<loan:status xmlns:loan="http://www.active-endpoints.com/wsdl/humantaskdemo">
  Thank you for applying for the Automobile loan for the amount of US$15000.
  Your loan is currently pending approval. You will receive an email once a decision has
  been made.
</loan:status>
```

Fault Response

A fault is indicated with an HTTP response code of 500 and whose content-type is `text/xml` (instead of `text/html` or `text/plain` indicating a generic "internal server error"). An example of a fault response is shown here:

```
HTTP/1.1 500 Internal Server Error
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Transfer-Encoding: chunked
Date: Thu, 11 Mar 2010 19:09:51 GMT
Connection: close
<aex:Fault xmlns:aex="http://www.active-endpoints.com/2004/06/bpel/extensions/">
   <faultcode name="systemError"
          namespace="http://www.active-endpoints.com/2004/06/bpel/extensions/"/>
   <faultstring>Could not find match for Operation from given parameters</faultstring>
</aex:Fault>
```

Attachments

When sending attachments the payload of the HTTP body must be `multipart/related` content, with the first part being the message XML payload (`text/xml`), followed by additional parts representing the attachments. Attachments sent with the payload are bound to the process variable associated with the message `Receive` activity.

```
POST /active-bpel/services/XML/humantaskProcessDemoService HTTP/1.1
Content-Type: multipart/related; type="text/xml"; start="<part1_id>";
boundary="the_boundry"
Content-Length: 1410
MIME-Version: 1.0
Host: localhost:8080
--the_boundry
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <part1_id>
<loan:loanProcessRequest xmlns:loan=
    "http://schemas.active-endpoints.com/sample/LoanRequest/2008/02/loanRequest.xsd">
 <loan:loanType>Automobile</loan:loanType>
 <loan:firstName>John</loan:firstName>
 <loan:lastName>Smith</loan:lastName>
 <loan:dayPhone>2039299400</loan:dayPhone>
 <loan:nightPhone>2035551212</loan:nightPhone>
 <loan:socialSecurityNumber>123-45-6789</loan:socialSecurityNumber>
 <loan:amountRequested>15000</loan:amountRequested>
 <loan:loanDescription>Application to finance the purchase of a Toyota Prius</
loan:loanDescription>
 <loan:otherInfo>Down payment is US$7500</loan:otherInfo>
 <loan:responseEmail>john.smith@example.com</loan:responseEmail>
</loan:loanProcessRequest>
--the_boundry
Content-Type: text/plain; charset=us-ascii
```

```
Content-Transfer-Encoding: 7bit
Content-ID: <part2_id>
[...Text Attachment Content...]
--the_boundry
Content-Type: image/jpeg
Content-ID: <part3_id>
Content-Transfer-Encoding: BASE64
Content-Description: Picture A
[...Image Content...]
--the_boundry--
```

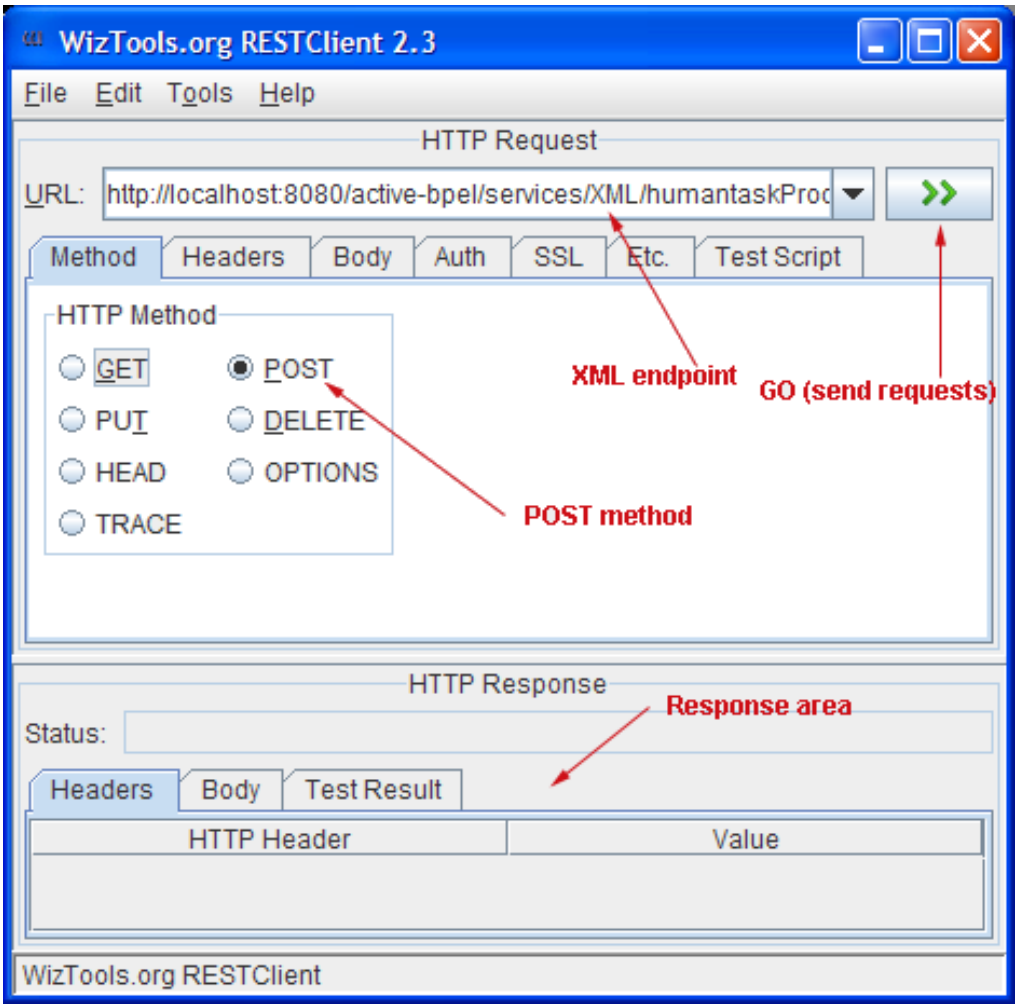
For testing, you can use any HTTP client testing tool available to you. For example, SOAP-UI, cURL (a popular command line tool: `http://curl.haxx.se/`) and RESTClient (see `http://code.google.com/p/rest-client/`), a Java based application to test RESTful web services.

Using the RESTClient Tool

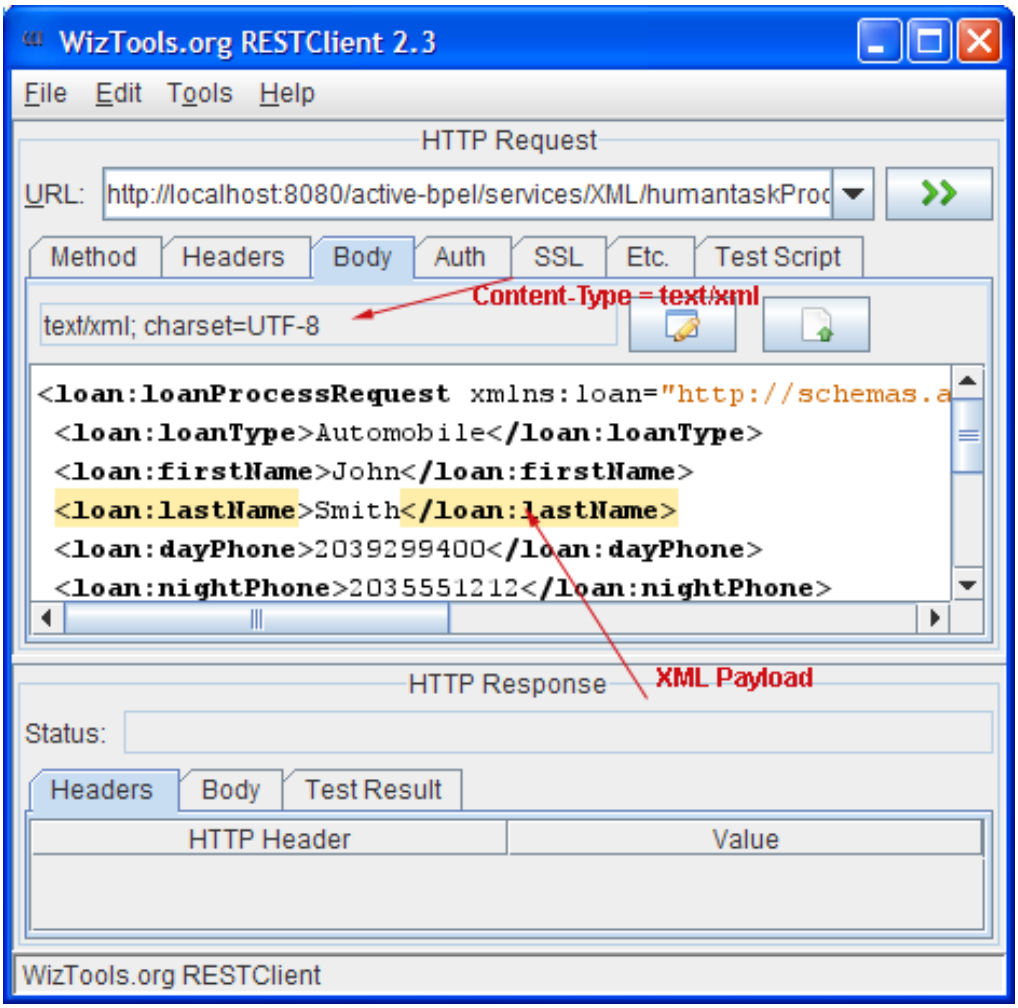
RESTClient is a Java application used to test RESTful services. It can be used to test a variety of HTTP communications including invoking Process Server processes using the XML binding endpoint. To use the GUI version of this tool, download the jar restclient-ui-2.3-jar-with-dependencies.jar library from `http://code.google.com/p/rest-client/downloads/list`.

To launch the application, run the command `java -jar restclient-ui-2.3-jar-with-dependencies.jar`. This will bring up the GUI application similar to the screenshot shown below.

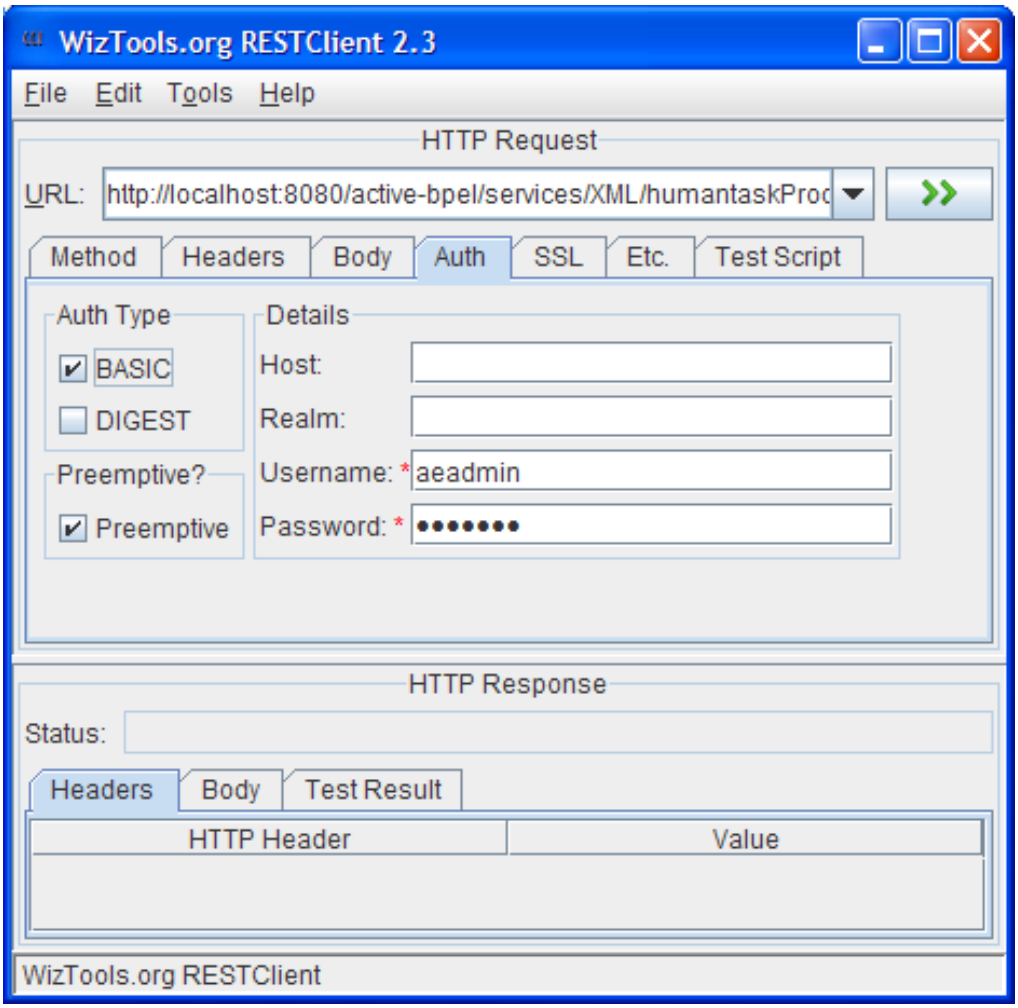
1. To invoke the sample process in provided with this SDK, ensure that your server is running and that you have deployed the humantaskProcessDemo bpr provided with this SDK to it.
2. To send a XML POST message to invoke a service, set the service URL for the XML binding. For example, `http://localhost:8080/active-bpel/services/XML/humantaskProcessDemoService`.



3. In the Method tab, set the HTTP Method to POST.
4. In the Body tab, set content-type to text/xml.
5. In the Body tab, provide the xml request element. You can copy the `<loan:loanProcessRequest>` sample shown in the previous section.



6. Set authorization information if needed via Auth tab.



7. Press the **Go** button to send the request
8. The response from the service is then shown in the Body tab in the HTTP Response area.

Java Example

The following snippet shows one approach to sending an HTTP POST to a service endpoint. The complete code is available in the `com.activevos.examples.xmlbinding.SimpleXmlServiceRequest` class provided to you in the examples folder.

```
public class SimpleXmlServiceRequest {
   /**
    * Class to hold service response data.
    */
   static class ServiceResponse {
      static final int SUCCESS = 0;
      static final int FAULTED = 1;
      static final int ERROR = 2;
      /** Code indicating if the service invoke was a success, fault or other error.*/
      int responseCode;
      /** Response xml data */
      String responseData;
   }

   /**
    * Invokes XML service using POST method returns service response.
```

```
    */
   public static ServiceResponse invokeService(URL aXmlServiceUrl, String aXmlPayload,
         String aUsername, String aPassword) throws IOException {
      HttpURLConnection httpConnection = null;
      BufferedReader reader = null;
      OutputStreamWriter writer = null;
      try
      {
         // create connection
         URLConnection c = aXmlServiceUrl.openConnection();
         httpConnection = (HttpURLConnection)c;
         httpConnection.setRequestProperty("Content-Type", "text/xml");
         httpConnection.setRequestProperty("Content-Length",
Integer.toString(aXmlPayload.length()));
         // Set credentials (if secured using BASIC auth).
         if (aUsername != null && aPassword != null) {
            // code to set the authorization header (e.g. BASIC)
         }
         httpConnection.setDoOutput(true);
         httpConnection.setInstanceFollowRedirects(true);
         // send the payload
         writer = new OutputStreamWriter(httpConnection.getOutputStream());
         writer.write(aXmlPayload);
         writer.flush();
         // read response
         if ( httpConnection.getResponseCode() == HttpURLConnection.HTTP_OK
               || httpConnection.getResponseCode() == HttpURLConnection.HTTP_ACCEPTED ) {
            reader = new BufferedReader(new
InputStreamReader(httpConnection.getInputStream()));
         } else {
            reader = new BufferedReader(new
InputStreamReader(httpConnection.getErrorStream()));
         }
         // read response
         StringBuilder sb = new StringBuilder();
         String line = null;
         while ((line = reader.readLine()) != null) {
            sb.append(line);
         }

         ServiceResponse response = new ServiceResponse();
         response.responseData = sb.toString();
         if ( httpConnection.getResponseCode() == HttpURLConnection.HTTP_OK
               || httpConnection.getResponseCode() == HttpURLConnection.HTTP_ACCEPTED ) {
            // Success!
            response.responseCode = ServiceResponse.SUCCESS;
         } else if (httpConnection.getResponseCode() ==
HttpURLConnection.HTTP_INTERNAL_ERROR
               && httpConnection.getContentType().toLowerCase().startsWith("text/xml") )
{
            // Faulted! (response code is 500 and content-type is text/xml
            response.responseCode = ServiceResponse.FAULTED;
         } else {
            // http/transport or other error
            response.responseCode = ServiceResponse.ERROR;
         }
         return response;
      }
      finally {
          //
         // clean up code goes here. E.g.: close writer, reader and disconnect http
connection.
      }
   }
   public static void main(String[] args) {
      // Create sample request. In this example, the sample request is created using
      // a string. Ideally, the XML element should be built using DOM i.e. with
DocumentBuilderFactory,
      // and DocumentBuilder.
      String xmlRequest = "<loan:loanProcessRequest
               xmlns:loan=\"http://schemas.active-endpoints.com/sample/LoanRequest/
```

```
2008/02/loanRequest.xsd\">\n"
            + " <loan:loanType>Automobile</loan:loanType>\n"
            + " <loan:firstName>John</loan:firstName>\n"
            + " <loan:lastName>Smith</loan:lastName>\n"
            + " <loan:dayPhone>2039299400</loan:dayPhone>\n"
            + " <loan:nightPhone>2035551212</loan:nightPhone>\n"
            + " <loan:socialSecurityNumber>123-45-6789</loan:socialSecurityNumber>\n"
            + " <loan:amountRequested>15000</loan:amountRequested>\n"
            + " <loan:loanDescription>Application to finance the purchase of a Toyota
Prius</loan:loanDescription>\n"
            + " <loan:otherInfo>Down payment is US$7500</loan:otherInfo>\n"
            + " <loan:responseEmail>john.smith@example.com</loan:responseEmail>\n"
            + "</loan:loanProcessRequest>";

      try {
         URL loanRequestUrl = new URL(
               "http://localhost:8080/active-bpel/services/XML/
humantaskProcessDemoService");
         System.out.println("Invoking service...");
         ServiceResponse response = invokeService(loanRequestUrl, xmlRequest,
"username", "password");

         if ( response.responseCode == ServiceResponse.SUCCESS ) {
            System.out.println("Success:");
         } else if ( response.responseCode == ServiceResponse.FAULTED ) {
            System.out.println("Faulted:");
         } else {
            System.out.println("Error:");
         }
         System.out.println(response.responseData);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

JSON Binding

Processes deployed to Process Server are available as JSON (JavaScript Object Notation) bindings. Similar to services with XML binding, the services must be WS-I compliant:

- WSDL `portType` operation is document-literal.
- Operation input message has only one part and that part is of element type.
- Operation output message has only one part and that part is of element type.

JSON Representation of XML Content

In order to invoke a service using JSON, the service request message, which is normally defined in XML, must be represented in JSON. The Process Server represents XML elements in JSON notation based on the methodology described by Google GData convention at `http://code.google.com/apis/gdata/json.html`.

XML element Represented as a JSON Object

```
<contactInfo />
```

JSON Equivalent

```
{
  "contactInfo" : {}
}
```

Attributes

Element attributes are represented as string properties.

```
<contactInfo type="home" default="true" />
```

JSON Equivalent

```
{
  "contactInfo" : {
                   "type"   : "home",
                   "default": "true"
                  }
}
```

In JavaScript, accessing attributes can be done as follows (assuming the JSON data is assigned to variable named `doc`):

```
var doc = { "contactInfo" : {
                     "type": "home",
                     "default": "true"
                  }
            };
     // type attribute is accessed via doc.contactInfo["type"]
     // or doc.contactInfo.type.
 alert("contact information type: " + doc.contactInfo.type);
```

Child Elements

Child elements are converted to Object type properties.

```
<contactInfo type="home" default="true" >
   <phone />
</contactInfo>
```

JSON equivalent (see 'phone' element object in line 5):

```
{
  "contactInfo" : {
                   "type"   : "home",
                   "default": "true",
                   "phone" : {}
                  }
}
```

Element text

Text values of elements are converted to string property named `$t`. The following shows how the `<phone/>` element text value "(203-555-1212)" is represented.

```
<contactInfo type="home" default="true" >
   <phone type="voice">203-555-1212</phone>
</contactInfo>
```

JSON:

```
{
  "contactInfo" : {
                     "type"   : "home",
                     "default": "true",
                     "phone"  : {
                                  "type" : "voice",
                                  "$t"   : "203-555-1212"
                                 }
                   }
}
```

Example JavaScript:

```
     // JavaScript:
var doc = { .... }; // define json
    // the phone number is doc.contactInfo.phone.$t
alert("phone number: " + doc.contactInfo.phone.$t);
```

Repeating elements

Elements that may appear more than once are converted to an array of objects:

```
<contactInfo type="home" default="true" >
   <phone type="voice">203-555-1212</phone>
   <phone type="fax">203-555-1213</phone>
</contactInfo>
```

JSON:

```
{
  "contactInfo" : {
                    "type"   : "home",
                    "default": "true",
                    "phone"  : [
                                  {
                                   "type" : "voice",
                                   "$t"   : "203-555-1212"
                                  },
                                  {
                                   "type" : "fax",
                                   "$t"   : "203-555-1213"
                                  }
                                ]

                  }
}
```

Example:

```
    // JavaScript:
    // Access phone via array index: doc.contactInfo.phone[0].$t
var i;
for (i=0; i < doc.contactInfo.phone.length; i++) {
  alert( "phone " + i + " = " + doc.contactInfo.phone[i].$t);
}
```

Namespace Prefix

If an element has a namespace prefix, the prefix and element name are concatenated using "`$`". Attribute prefixes are presented in the same way. For example, `<c:contactInfo>` is represented as `{ c$contactInfo: {} }`. Attribute `xmlns:c="urn:ns:contactinfo"` is represented as `xmlns$c:"urn:ns:contactinfo"`. The following example XML uses prefixed elements `xmlns:p="urn:ns:person"`, `xmlns:c="urn:ns:contactinfo"`, and also two elements that are unqualified (`email` and `photo`).

```
<p:person xmlns:p="urn:ns:person">
   <p:firstName>John</p:firstName>
   <p:lastName>Smith</p:lastName>
   <c:contactInfo xmlns:c="urn:ns:contactinfo" type="home" default="true" >
      <c:phone type="voice">203-555-1212</c:phone>
      <c:phone type="fax">203-555-1213</c:phone>
      <email>jsmith@example.com</email>Happy
   </c:contactInfo>
   <photo>http://example.com/jsmith/profile.png</photo>
</p:person>
```

JSON

```
{
  "p$person" : {
    "xmlns$p" : "urn:ns:person",
```

```
    "p$firstName" : {
      "$t" : "John"
    },
    "p$lastName" : {
      "$t" : "Smith"
    },
    "c$contactInfo" : {
      "xmlns$c" : "urn:ns:contactinfo",
      "default" : "true",
      "type" : "home",
      "c$phone" : [
        {
          "type" : "voice",
          "$t" : "203-555-1212"
        },
        {
          "type" : "fax",
          "$t" : "203-555-1213"
        }
      ],
      "email" : {
        "$t" : "jsmith@example.com"
      }
    },
    "photo" : {
      "$t" : "http://example.com/jsmith/profile.png"
    }
  }
}
```

Example:

```
    // JavaScript:
var doc = { ... }; // JSON representation of person.
    // first Name: /p:person/p:firstName/text()
alert("firstName =" + doc.p$person.p$firstName.$t);
    // photo /p:person/photo/text()
alert("photo =" + doc.p$person.photo.$t);
    // email /p:person/c:contactInfo/email/text()
alert("email =" + doc.p$person.c$contactInfo.email.$t);
var i;
for (i=0; i < doc.p$person.c$contactInfo.c$phone.length; i++) {
   alert( "phone " + i + "=" + doc.p$person.c$contactInfo.c$phone[i].$t);
}
```

Default namespace

Instead of using elements with prefixes, you can declare the namespace on each element. For example instead of `<c:contactInfo xmlns:c="urn:ns:contactinfo">`, you can use `<contactInfo xmlns="urn:ns:contactinfo">`. Now, the JSON object properties do not have `"$"` in the middle of their property names, which is easier to use; for example, `contactInfo` instead of `c$contactInfo`. Process Server always uses namespaces when it serializes (converts) XML to JSON.

The prefixed element `<p:person xmlns:p="urn:ns:person">` used prviously can be represented using the default namespace of each element (when the namespace changes):

```
 <person xmlns="urn:ns:person">
   <firstName>John</firstName>
   <lastName>Smith</lastName>
   <contactInfo xmlns="urn:ns:contactinfo" type="home" default="true" >
      <phone type="voice">203-555-1212</phone>
      <phone type="fax">203-555-1213</phone>
      <email xmlns="">jsmith@example.com</email>
   </contactInfo>
   <photo xmlns="">http://example.com/jsmith/profile.png</photo>
</person>
```

The following shows the JSON representation and how it is used in JavaScript. Notice that it is easier to read and use compared to JSON properties notation that concatenates a prefix/element with a `$`. Both formats work; however Process Server always uses the non-prefix method when it converts XML to JSON.

```
{
  "person" : {
    "xmlns" : "urn:ns:person",
    "firstName" : {
      "$t" : "John"
    },
    "lastName" : {
      "$t" : "Smith"
    },
    "contactInfo" : {
      "xmlns" : "urn:ns:contactinfo",
      "default" : "true",
      "type" : "home",
      "phone" : [
        {
          "type" : "voice",
          "$t" : "203-555-1212"
        },
        {
          "type" : "fax",
          "$t" : "203-555-1213"
        }
      ],
      "email" : {
        "xmlns" : "",
        "$t" : "jsmith@example.com"
      }
    },
    "photo" : {
      "xmlns" : "",
      "$t" : "http://example.com/jsmith/profile.png"
    }
  }
}
```

Example:

```
     // JavaScript:
var doc = { ... }; // JSON representation of person using default namespace (and no
prefixes).
    // first Name:
alert("firstName =" + doc.person.firstName.$t);
    // (compare above to prefixed version doc.p$person.p$firstName.$t)
    // photo
alert("photo =" + doc.person.photo.$t);
    // (compare to prefixed version doc.p$person.photo.$t)
    // email /p:person/c:contactInfo/email/text()
alert("email =" + doc.person.contactInfo.email.$t);
    // (compare to doc.p$person.c$contactInfo.email.$t)
var i;
for (i=0; i < doc.person.contactInfo.phone.length; i++) {
 alert( "phone " + i + "=" + doc.person.contactInfo.phone[i].$t);
}
```

One advantage of using the non-prefixed method when dealing with JSON representations is that you do not have to determine the actual prefix bound to the element at runtime to access element properties. So, instead of `doc.ns1$person.ns5$contactInfo.email.type`, you can use `doc.person.contactInfo.email.type`.

Restrictions on JSON Representation of XML Content

The following restrictions exist for JSON representation of XML content:

- Mixed types are not supported.

- Element text nodes that contain only whitespace characters (including tabs, line feeds, and the like) are not serialized into JSON, since in general, these artifacts are due to pretty printing XML.
- For the XML document that uses prefixes, you must assign a unique prefix for each namespace rather than overloading a prefix to multiple namespaces.

```
<!-- Prefix 'p' is initially used for ns urn:ns:person. -->
<p:person xmlns:p="urn:ns:person">
   <!-- Prefix 'p' is re-used in contactInfo in a different ns.
        Avoid this case. -->
   <p:contactInfo xmlns:p="urn:ns:contactinfo" type="home" default="true" >
   </p:contactInfo>
</p:person>
```

- Elements or attributes names should follow standard rules for naming JavaScript variables. For example, do not use any punctuation marks of any kind in element (or attribute) names other than the underscore. For example, `<first-name>` is not allowed because it has a hyphen(-) in it name).
- Avoid attributes or elements names that are reserved words in JavaScript. See `https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Reserved_Words` for more information.
- Avoid elements or attributes named `length`.
- Avoid having an element whose attributes and child elements have the same name. For example:

```
<contactInfo xmlns="urn:ns:contactinfo" type="home" default="true" >
   <!-- Note 'type' is an attribute (of contactInfo) as well as a
        child element. Avoid this case. -->
   <type>foo</type>
   <phone type="voice">203-555-1212</phone>
   <phone type="fax">203-555-1213</phone>
   <email xmlns="">jsmith@example.com</email>
</contactInfo>
```

Using AE_JSON_NODE_UTIL JavaScript library

The `AE_JSON_NODE_UTIL` JavaScript object provided by the `ae-avc-util.js` script file included with this SDK has helper functions that you can use when working with JSON representation of XML data.

Function	Description

`AE_JSON_NODE_UTIL.getElements( aJsonObject)`	Returns JSON objects in an array. Use this when the JSON object represents a repeating element.
	
	`var phones = AE_JSON_NODE_UTIL.getElements( doc.person.contactInfo.phone) ;`
	`  // phones is an array.`

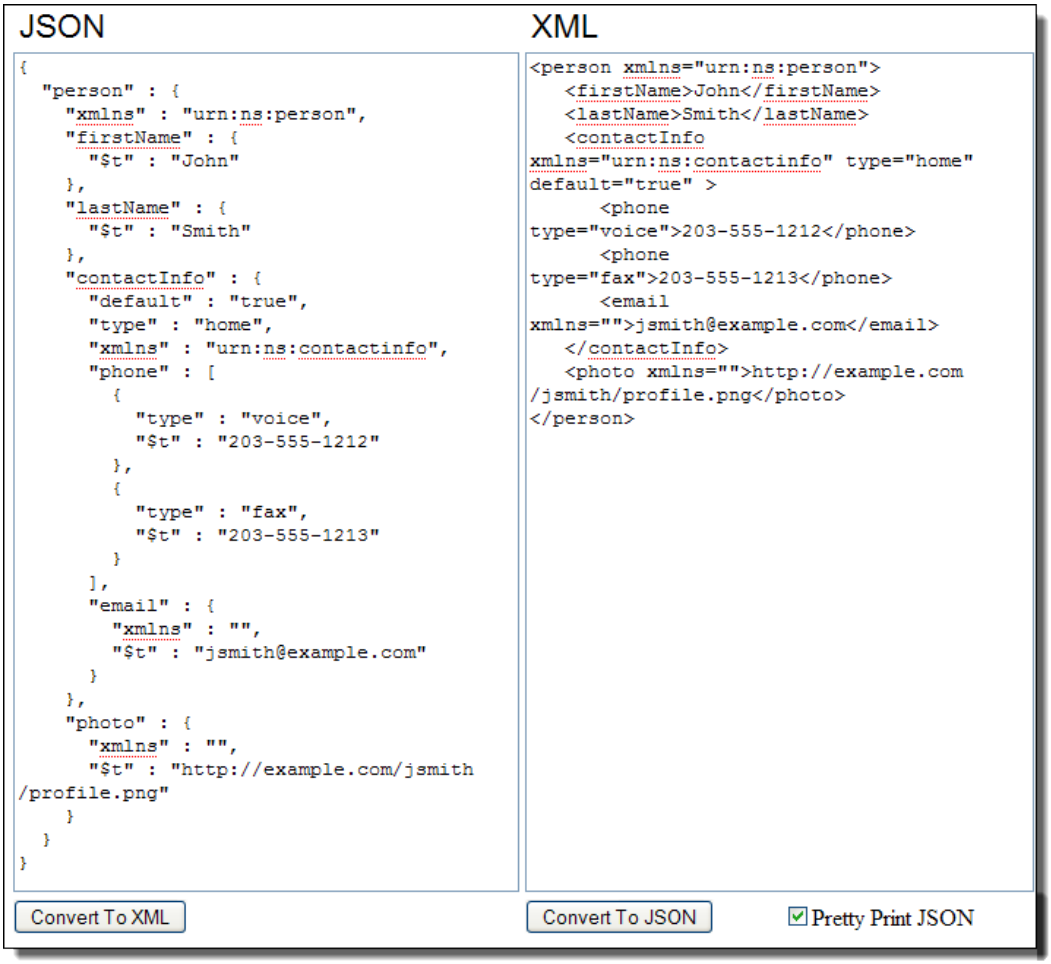
`AE_JSON_NODE_UTIL.getText( aJsonObject, aDefaultText)`	Returns the text value (the value of `$t` property) given the JSON object. If the text node does not exist, the default string value is returned.
	
	`    // get first number or return "Not Available" if number does not exist.`
	`var phones = AE_JSON_NODE_UTIL.getElements(doc.person.contactInfo.phone);`
	`   // phone_number is 203-555-1212.`
	`var phone_number = AE_JSON_NODE_UTIL.getText(phones[0], "Not Available");`

Function	Description
`AE_JSON_NODE_UTIL.isXsiNil( aJsonObject)`	Returns true if the `aJsonObject` element exists and it is XSI nulled.

`AE_JSON_NODE_UTIL.getAttribute( aJsonObj, aAttributeName, aDefault)`	Returns attribute value given its name. If the attribute does not exist, the default value is returned. Normally you do not have to use this function as you can directly access JSON properties (attributes). This method is useful if your XML has a case where an element child and attribute have the same name (see restrictions above).
	
	`// get list of phones`
	`var phones = AE_JSON_NODE_UTIL.getElements(doc.person.contactInfo.phone);`
	`// get phone type (voice, fax etc.)`
	`var phone_type = AE_JSON_NODE_UTIL.getAttribute(phones[0], "type", "Not Available");`
	`// phone_type is 'voice'.`

JSON XML Conversion Tool

During development you can use the JSON/XML online conversion tool provided by the ActiveVOS at `http://host:port/active-bpel/jsonConverter.html` where `host` and `port` is the host and port where Process Server is installed (for example, localhost:8080). This tool allows you to enter well-formed XML content and convert it to JSON, or enter JSON content and convert it to XML.



Invoking a Process Using JSON

Invoking a process using JSON is similar to the method described for the XML binding endpoint:

1. Prepare to send an HTTP POST message to the service JSON endpoint at `http://host:port/active-bpel/services/JSON/serviceName`.
2. Set the POST header content-type to application/json.
3. Set POST payload (body content) to be the string version of the JSON request.
4. Include authorization headers if needed.
5. Send an HTTP POST message to the service JSON endpoint.

The response from the POST is a HTTP 200/OK response with content-type of `application/json`. The response body will contain a JSON service response (as a string). The following snippet shows the HTTP request used to invoke the Loan Approval process using the `humantaskProcessDemoService` service. .

```
POST /active-bpel/services/JSON/humantaskProcessDemoService HTTP/1.1
Content-Length: 581
Content-Type: application/json; charset=UTF-8
Authorization: Basic YWVhZG1pbjphZWFkbWlu
```

```
Host: localhost:8080
{"loanProcessRequest":
  {"xmlns":"http:\/\/schemas.active-endpoints.com\/sample\/LoanRequest\/2008\/02\/
loanRequest.xsd",
   "loanType":{"$t":"Automobile"},
   "firstName":{"$t":"John"},
   "lastName":{"$t":"Smith"},
   "dayPhone":{"$t":"2039299400"},
   "nightPhone":{"$t":"2035551212"},
   "socialSecurityNumber":{"$t":"123-45-6789"},
   "amountRequested":{"$t":"15000"},
   "loanDescription":{"$t":"Application to finance the purchase of a Toyota Prius"},
   "otherInfo":{"$t":"Down payment is US$7500"},
   "responseEmail":{"$t":"john.smith@example.com"}
  }
 }
```

The response to the above request looks like:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
{"status":
  {"xmlns":"http:\/\/www.active-endpoints.com\/wsdl\/humantaskdemo",
      "$t":"Thank you for applying for the Automobile loan for the amount of US$15000.
Your loan is currently pending approval."
  }
}
```

Faults

A fault response is returned with HTTP error code 500 and content-type `application/json`:

```
HTTP/1.1 500 Internal Server Error
Content-Type: application/json;charset=utf-8
{"Fault":
   {"xmlns":"http:\/\/www.active-endpoints.com\/2004\/06\/bpel\/extensions\/",
    "faultcode":
           {"name":"invalidVariables",
            "namespace": "http:\/\/docs.oasis-open.org\/wsbpel\/2.0\/process\/
executable",
            "xmlns":""
            },
     "faultstring":
           {"xmlns":"",
            "$t":" fault error message"
           }
  }
}
```

Attachments

Similar to XML process invokes, the payload of the HTTP body must be `multipart/related` content, with the first part being the JSON payload with content-type `application/json`, followed by additional parts representing the attachments.

```
POST /active-bpel/services/JSON/humantaskProcessDemoService HTTP/1.1
Content-Type: multipart/related; type="application/json"; start="<part1_id>";
boundary="the_boundary"
Content-Length: 1234
MIME-Version: 1.0
Host: localhost:8080
--the_boundary
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <part1_id>
```

```
{"loanProcessRequest":
  {"xmlns":"http:\/\/schemas.active-endpoints.com\/sample\/LoanRequest\/2008\/02\/
loanRequest.xsd",
   "loanType":{"$t":"Automobile"},
   "firstName":{"$t":"John"},
   "lastName":{"$t":"Smith"},
   "dayPhone":{"$t":"2039299400"},
   "nightPhone":{"$t":"2035551212"},
   "socialSecurityNumber":{"$t":"123-45-6789"},
   "amountRequested":{"$t":"15000"},
   "loanDescription":{"$t":"Application to finance the purchase of a Toyota Prius"},
   "otherInfo":{"$t":"Down payment is US$7500"},
   "responseEmail":{"$t":"john.smith@example.com"}
  }
 }
--the_boundary
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
Content-ID: <part2_id>
[...Text Attachment Content...]
--the_boundary
Content-Type: image/jpeg
Content-ID: <part3_id>
Content-Transfer-Encoding: BASE64
Content-Description: Picture A
[...Image Content...]
--the_boundary--
```

Attachments Using Multipart Form-Data

Processes can be invoked using a `multipart/form-data` type payload (normally used by HTML forms). When using this content-type, Process Server requires the form-data to contain one field with name `_json`. The value of this field must contain the JSON request.

```
POST /active-bpel/services/JSON/humantaskProcessDemoService HTTP/1.1
Content-Type: multipart/form-data; boundary="the_boundary"
Content-Length: 3449
MIME-Version: 1.0
Host: localhost:8080
--the_boundary
Content-Disposition: form-data; name="_json"
{"loanProcessRequest":{"xmlns":"http:\/\/schemas.active-endpoints.com\/sample\/
LoanRequest\/2008\/02\/loanRequest.xsd","loanType":{"$t":"Automobile"},"firstName":
{"$t":"John"},"lastName":{"$t":"Smith"},"dayPhone":{"$t":"2039299400"},"nightPhone":
{"$t":"2035551212"},"socialSecurityNumber":{"$t":"123-45-6789"},"amountRequested":
{"$t":"15000"},"loanDescription":{"$t":"Application to finance the purchase of a Toyota
Prius"},"otherInfo":{"$t":"Down payment is US$7500"},"responseEmail":
{"$t":"john.smith@example.com"}}}
--the_boundary
Content-Disposition: form-data; name="file_1"; filename="picture.gif"
Content-Type: image/gif
[image content]
--the_boundary--
```

The above request was generated by the HTML form:

```
<form method='POST'
   enctype='multipart/form-data' action='http://localhost:8080/active-bpel/services/JSON/
humantaskProcessDemoService'>
   <!-- JSON message payload in a hidden field named '_json'  -->
   <input type="hidden" name="_json" value='{"loanProcessRequest":{"xmlns":"http:\/\/
schemas.active-endpoints.com\/sample\/LoanRequest\/2008\/02\/loanRequest.xsd","loanType":
{"$t":"Automobile"},"firstName":{"$t":"FileUploadJohn"},"lastName":
{"$t":"Smith"},"dayPhone":{"$t":"2039299400"},"nightPhone":
{"$t":"2035551212"},"socialSecurityNumber":{"$t":"123-45-6789"},"amountRequested":
{"$t":"15000"},"loanDescription":{"$t":"Application to finance the purchase of a Toyota
Prius"},"otherInfo":{"$t":"Down payment is US$7500"},"responseEmail":
```

```
{"$t":"john.smith@example.com"}}}' />
   File1: <input type="file" name="file_1" /> <br/>
   <input type="submit" value="Upload File"/>
</form>
```

Normally the `multipart/form-data` request generation and handling of the response is done using JavaScript (for AJAX-style browser applications). Note that the response to a process invoke (with or without attachments) using `multipart/form-data`, is an `application/json` response (as shown in previous examples--use the table of contents to see them). Since a response from a process can have attachments in addition to the response message (JSON), you can limit the response to the message (JSON) only by specifying `messageOnly=true` in the query string. Similarly, to restrict the response to contain only the attachments (if the process sends back attachments), use `attachmentOnly=true` in the query string.

ActiveVOS XML and JSON Bindings for Services

Some AJAX-based form submit scripts that support file upload expect the return `content-type` to be `text/html` (instead of `application/json`) because of the way scripts target iFrames. In these cases, you specify parameter `responseContentType=text/html`. If the `responseContentType` IS present, Process Server uses the value defined by the parameter as the response content-type instead of the default `application/json`.

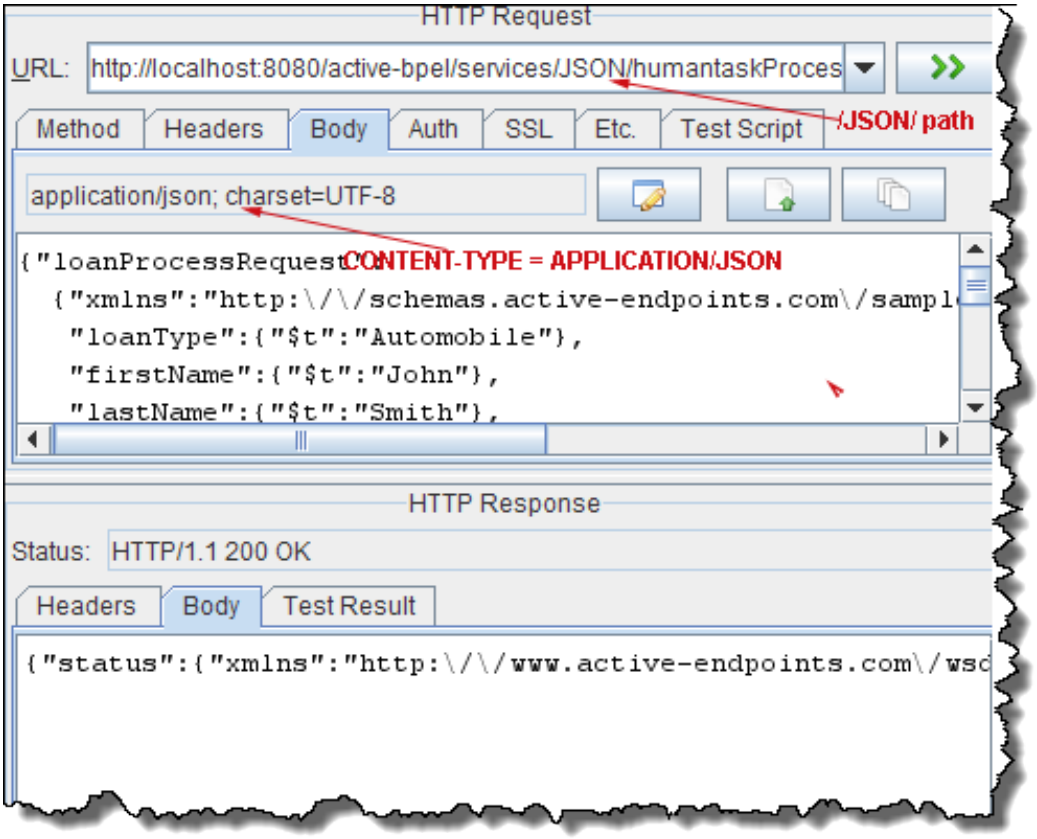
A few AJAX file upload scripts such as the `jQuery Forms Plugin` (see `http://plugins.jquery.com/project/form/`) also require responses to `multipart/form-data` forms POSTs to be wrapped in a HTML element such as the `textarea` tag. Do this by using the `responseWrap` parameter. The value of this parameter should be the HTML element tag name.

The following shows HTML file upload form used by the jQuery Forms plugins, which requires the response to be returned wrapped in a HTML textarea tag with content-type head of `text/html`:

```
<form method='POST'
       enctype='multipart/form-data'
       action='http://localhost:8080/active-bpel/services/JSON/
humantaskProcessDemoService'>
   <!-- JSON message payload in a hidden field named '_json'  -->
   <input type="hidden" name="_json"
        value='{
         "loanProcessRequest":
         {"xmlns"               :"http:\/\/schemas.active-endpoints.com\/sample\/
LoanRequest\/2008\/02\/loanRequest.xsd",
          "loanType"            :{"$t":"Automobile"},
          "firstName"           :{"$t":"FileUploadJohn"},
          "lastName"            :{"$t":"Smith"},
          "dayPhone"            :{"$t":"2039299400"},
          "nightPhone"          :{"$t":"2035551212"},
          "socialSecurityNumber":{"$t":"123-45-6789"},
          "amountRequested"     :{"$t":"15000"},
          "loanDescription"     :{"$t":"Application to finance the purchase of a Toyota
Prius"},
          "otherInfo"           :{"$t":"Down payment is US$7500"},
          "responseEmail"       :{"$t":"john.smith@example.com"}}}' />
   <!-- Force response content-type to text/html -->
   <input type="hidden" name="responseContentType" value="text/html" />
   <!-- Return response wrapped in a textarea element -->
   <input type="hidden" name="responseWrap" value="textarea" />
   File1: <input type="file" name="file_1" /> <br/>
   <input type="submit" value="Upload File"/>
</form>
```

RESTClient

You can use the RESTClient test application to send JSON data to the Process Server. The content-type of the payload should be set to `application/json`.



Invoking a Process with jQuery

The following example shows how to invoke a process using AJAX with jQuery. The scripts used for the request require jQuery 1.3.2+ (`http://www.jquery.com`) and `json2.js` (`http://www.JSON.org/json2.js`). The `json2.js` script is used to convert a JSON object to a string.

```
var loanRequestJSON = {"loanProcessRequest":
    {"xmlns":"http:\/\/schemas.active-endpoints.com\/sample\/LoanRequest\/2008\/02\/
loanRequest.xsd",
        "loanType":{"$t":"Automobile"},
        "firstName":{"$t":"John"},
        "lastName":{"$t":"Smith"},
        "dayPhone":{"$t":"2039299400"},
        "nightPhone":{"$t":"2035551212"},
        "socialSecurityNumber":{"$t":"123-45-6789"},
        "amountRequested":{"$t":"15000"},
        "loanDescription":{"$t":"Application to finance the purchase of a Toyota Prius"},
        "otherInfo":{"$t":"Down payment is US$7500"},
        "responseEmail":{"$t":"john.smith@example.com"}
     }
};
    // convert JSON to string using function in json2.js script
var jsonStr = JSON.stringify(loanRequestJSON);
    // Send jQuery AJAX POST to http://localhost:8080/active-bpel/services/JSON/
humantaskProcessDemoService
$.ajax( {
   url : "http://localhost:8080/active-bpel/services/JSON/humantaskProcessDemoService",
   type : "POST",
   contentType : "application/json; charset=utf-8",
   data : jsonStr, // JSON payload
   dataType : "json", // expected return data type
   cache : false,
```

```
   success : function(aJsonResponse, aTextStatus) {
      // callback function on success
      alert("status=" + aJsonResponse.status.$t);
   },

   error : function(aXmlHttpReq, aTextStatus, aErrorThrown) {
     // callback function when an error occurs - including faults.
     // code that checks for response status code and data
     // to determine the type of error and fault.
     // var resp_contentType = aXmlHttpReq.getResponseHeader("Content-Type");
     // var http_statusCode = aXmlHttpReq.status;
     // var http_statusMsg = aTextStatus;
   }
});
```

The `AE_AJAX_SERVICE_UTIL` Javascript object in the `ae-avc-util.js` script included with the SDK provides has a helper function `postJSON(url, jsonRequest, successCallbackFn, faultCallbackFn, errorCallbackFn)` to POST JSON data.

```
var loanRequestJSON = {"loanProcessRequest":
    {"xmlns":"http:\/\/schemas.active-endpoints.com\/sample\/LoanRequest\/2008\/02\/
loanRequest.xsd",
     "loanType":{"$t":"Automobile"},
     "firstName":{"$t":"John"},
     "lastName":{"$t":"Smith"},
     "dayPhone":{"$t":"2039299400"},
     "nightPhone":{"$t":"2035551212"},
     "socialSecurityNumber":{"$t":"123-45-6789"},
     "amountRequested":{"$t":"15000"},
     "loanDescription":{"$t":"Application to finance the purchase of a Toyota Prius"},
     "otherInfo":{"$t":"Down payment is US$7500"},
     "responseEmail":{"$t":"john.smith@example.com"}
    }
};
AE_AJAX_SERVICE_UTIL.postJSON(
   // service url
 "http://localhost:8080/active-bpel/services/JSON/humantaskProcessDemoService",

   // JSON request object (not string)
 loanRequestJSON,

   // success callback of JSON data
 function(aJsonResponse) {
   alert("status=" + aJsonResponse.status.$t);
 },

   // fault callback of JSON data
 function(aJsonFault) {
   // handle fault
 },

   // http error callback
 function(aStatusCode, aStatusMessage) {

   // handle transport error

 }
);
```

Here is another example that uses the WS-HumanTask (WSHT) API `getMyTasks` operation via JSON. This example fetches tasks using the API and populates a HTML table with the results.

```
 <! --  html snippet -->
<table cellpadding="2" border="1">
   <thead>
      <tr><th colspan="5">Get My Tasks Response (from JSON):</th></tr>
      <tr><th>Task ID</th> <th>Status</th><th>Owner</th>
```

```
      <th>PresentationName</th><th>PresentationSubject</th></tr>
   </thead>
   <! --  the table body will be populated using JavaScript -->
   <tbody id="taskList">
   </tbody>
</table>


// getMyTasks reqests
var getMyTasksRequest = {
     "getMyTasks" : {
             "xmlns" : "http://www.example.org/WS-HT/api/xsd",
             "taskType" : { "$t" : "TASKS" },
             "genericHumanRole" : { "$t" : "POTENTIAL_OWNERS" },
             "status" : { "$t" : "READY" },
             "maxTasks" : { "$t" : "10" }
   }
};
  // send JSON
AE_AJAX_SERVICE_UTIL.postJSON(
       // task-client service url
    "http://localhost:8080/active-bpel/services/JSON/AeB4PTaskClient-taskOperations",
       // req.
    getMyTasksRequest,
       // success callback of JSON data
    function(aJsonResponse) {
       // display results in a table
      populateTaskList(aJsonResponse);
    },
       // fault callback of JSON data
    function(aJsonFault) {
       alert("Fault!");
    },
       // http error callback
    function(aStatusCode, aStatusMessage) {
      alert("Transport error: " + aStatusCode + " " + aStatusMessage);
    }
);
//
// This function takes a getMyTasksResponse element (in JSON) and populates
// the tasks in an HTML table.
//
function populateTaskList(aJsonResponse) {
      // Note: aJsonResponse contains getMyTasksResponse object)
      // clear current contents by removing all children of the table (table rows)
    $("#taskList").empty();
      // check to make sure response exists.
    if (!aJsonResponse.getMyTasksResponse) {
        alert("Response 'getMyTasksResponse' expected.");
        return;
    }
      // get list of task abstracts (//htdt:getMyTasksResponse/htdt:taskAbstract)
    var taskAbstracts = AE_JSON_NODE_UTIL.getElements
                                (aJsonResponse.getMyTasksResponse.taskAbstract);
    var i;
      // for each taskAbstract, build the html row and append to table tbody.
    for (i = 0; i < taskAbstracts.length; i++) {
      var html =  "<tr>";
      html += "<td>" + AE_JSON_NODE_UTIL.getText(taskAbstracts[i].id, "n/a") + "</td>";
      html += "<td>" + AE_JSON_NODE_UTIL.getText(taskAbstracts[i].status, "n/a") + "</
td>";
      html += "<td>" + AE_JSON_NODE_UTIL.getText(taskAbstracts[i].actualOwner, "n/a") +
"</td>";
      html += "<td>" + AE_JSON_NODE_UTIL.getText(taskAbstracts[i].presentationName,
"n/a") + "</td>";
      html += "<td>" + AE_JSON_NODE_UTIL.getText(taskAbstracts[i].presentationSubject,
"n/a") + "</td>";
      html += "</tr>";
      // append to table
      $("#taskList").append(html);
```

```
    };
}
```

Using AE_AJAX_SERVICE_UTIL Functions

The AE_AJAX_SERVICE_UTIL Javascript object (in `ae-avc-util.js`) has the following utility functions related to making JSON invokes:

- `getActiveVOSServiceURL`
- `postJSON`

getActiveVOSServiceUrl(serviceNamePath)

A helper function that creates the endpoint URL to the given service name by appending the service name to a predefined engine endpoint. The engine URL should be assigned (at the start of your script) to the global variable `AE_ACTIVEVOS_ENGINE_URL`. If a value to `AE_ACTIVEVOS_ENGINE_URL` is not assigned, then `http://localhost:8080/active-bpel` is used.

If a value is assigned to the global variable `AE_ACTIVEVOS_PROXY_URL`, then this function returns the proxy URL with service URL in the query string in a parameter named `destination`. See example below.

```
// Define engine context URL and optional proxy URL at the start of your script.
// If the engine URL is not given, 'http://localhost:8080/active-bpel' is used.
// Note: AE_ACTIVEVOS_ENGINE_URL is global variable declared in ae-avc-util.js.
//       You only need to assign it a value (to server /active-bpel context).
AE_ACTIVEVOS_ENGINE_URL = "http://demoserver:81/active-bpel";

// Optionally, if you are using a proxy for AJAX requests, you can
// assign it a value here:
// AE_ACTIVEVOS_PROXY_URL = "http://localhost:8080/proxy";
// for example, JSON service URL.
var loanServiceXmlUrl = AE_AJAX_SERVICE_UTIL.getActiveVOSServiceUrl("JSON/
humantaskProcessDemoService");

// loanServiceXmlUrl is now equivalent to:
//      AE_ACTIVEVOS_ENGINE_URL + "/services/" + "JSON/humantaskProcessDemoService"
// for example,
//      http://demoserver:81/active-bpel/JSON/humantaskProcessDemoService
// If a value to AE_ACTIVEVOS_PROXY_URL is assigned, the 'loanServiceXmlUrl' becomes
part of
// the proxy URL string parameter named 'destination'. For example,
//      http://localhost:8080/proxy?destination=http://demoserver:81/active-bpel/JSON/
humantaskProcessDemoService
// For example, XML service URL.
var loanServiceXmlUrl = AE_AJAX_SERVICE_UTIL.getActiveVOSServiceUrl("XML/
humantaskProcessDemoService");
```

postJSON(aServiceUrl, aJsonData, aOnSuccessFn, aOnFaultFn, aOnErrorFn, aTimeOut)

Sends the JSON request to the server using HTTP POST and calls back with the JSON response. This function also handles error responses and callbacks either the JSON fault handler or the HTTP transport level error handler. Only the `aServiceUrl` and `aJsonData` are required.

```
var serviceUrl = "http://localhost:8080/active-bpel/JSON/humantaskProcessDemoService";
  // or serviceUrl=AE_AJAX_SERVICE_UTIL.getActiveVOSServiceUrl("JSON/
humantaskProcessDemoService");
  // send JSON
AE_AJAX_SERVICE_UTIL.postJSON(
  serviceUrl,    // service url
  jsonRequest,   // request JSON object

 function(aJsonResponse) { // success callback of JSON data
     // handle success response
   alert("Success!");
 },

 function(aJsonFault) { // fault callback of JSON data
```

```
    alert("Fault!");
  },

  function(aStatusCode, aStatusMessage) { // http error callback
    alert("Transport error: " + aStatusCode + " " + aStatusMessage);
  }
);
```

JSON Process Invoke Examples

Complete working JSON process invoke examples can be found in the `/examples/javascript/` directory.

- `json-loanRequest.html`: shows how to submit an HTML form to invoke the Loan process using JSON.
- `json-getMyTasks.html`: shows how to interact with the WSHT `getMyTasks` API operation using JSON.
- `json-getMyTasks_JSONP.html`: same as above, but using JSONP.
- `json-getInstance.html`: shows how to interact with the Informatica WSHT API extensions. This example uses the `getInstance()` operation to fetch task instance details, including input and output data

To use these examples, you must deploy all files in this folder to the application or Web server where the Process Server engine is deployed. For example, when using the Process Developer and its embedded server, you can deploy the provided `activevos-javascript-examples.war`:

1. Navigate to the embedded server's Active-BPELl webapps folder; for example:

   ```
   C:\Program Files\ActiveVOS\Designer\designer\dropins\activevos\eclipse\plugins
   \org.activebpel.enginep_7.1.2\server\webapps\
   ```

2. Copy the `/examples/activevos-javascript-examples.war` war file to the `webapps` folder.
3. Start the embedded server.

The examples should be available at `http://localhost:8080/Human/activevos-javascript-examples/`

Access the WS-Human Task API Using JSON

The topics below describe how to access the WS-HUMAN Task API using JSON

Constants for Common String Literals

Constants for Common String Literals

The `AE_TASK_GLOBALS` in `ae-avc-tasks.js` contains common constants that can be used from your code. For example, the string 'illegalArgument' (WSHT fault name) is defined in `AE_TASK_GLOBALS.TASK_FAULT_NAMES.ILLEGAL_ARGUMENT`. The `http://www.example.org/WS-HT` namespace is defined in `AE_TASK_GLOBALS.NAMESPACES.XMLNS_HTD`. Additional details are available in the `AeTaskGlobals` class in the `ae-avc-tasks.js` script.

Utility Functions

Utility Functions

The `getMyTasks` example provided in the Invoking a Process with jQuery section (the full example is located in `json-getTasks.html`) demonstrated how to send a WSHT `getMyTasks` request to the server using `AE_AJAX_SERVICE_UTIL.postJSON()` utility function. The `ae-avc-tasks.js` script contains the `AE_TASK_UTIL` instance with the following convenience functions:

- `getFaultData`
- `isFault`
- `createGetTasksRequest`

AE_TASK_UTIL.getFaultData(aJsonFault)

Returns an object `{faultName : "wshtFaultName", message : "faultMessageStr"}` or `null` if the `aJsonFault` is not a WSHT SOAP fault. The 'wshtFaultName' values are standard WSDL: illegalArgument, illegalAccess, illegalOperation, and recipientNotAllowed.

```
// code that handles fault due to a human task operation invoke.
// check if this is a WSHT fault
var faultData = AE_TASK_UTIL.getFaultData(aJsonFault);
if (faultData != null) {
  // wsht fault.
  if (faultData.faultName == AE_TASK_GLOBALS.TASK_FAULT_NAMES.ILLEGAL_ARGUMENT) {
      alert("illegal arg fault: " + faultData.message);
  }
} else {
  // system soap fault?
  var soapFaultData = AE_AJAX_SERVICE_UTIL.getSystemFault(aJsonFault);
  if (soapFaultData != null) {
    alert("SOAP fault. Fault name:" + soapFaultData.name
          + ", message:" + soapFaultData.message);
  }
}
```

AE_TASK_UTIL.isFault(aJsonResponse)

Returns true if the JSON object `aJsonResponse` is a known WSHT fault (per WSDL).

AE_TASK_UTIL.createGetTasksRequest(aParams, aGetTasksTemplate)

Creates a generic request `getTasks` request and returns the JSON object. The params object contains the optional values. The options are:

- `taskType`: one of the following strings: ALL, TASKS, or NOTIFICATIONS. The default is TASKS.
- `genericHumanRole`: a string: generic human role such as potential owners.
- `status`: a string or an array of strings for the task status (such as READY).
- `whereClause`: a string
- `maxTasks`: an integer: maximum number of tasks to retrieve. The default is 20.
- `taskIndexOffset`: an integer: task list start offset index.
- `searchBy`: search by string.

Provide a value of null to remove existing elements.

If the optional `aGetTasksTemplate` object, which is an existing JSON request, is given, `aGetTasksTemplate` is first cloned, and then the params are applied.

To create the following request:

```
<aeb:getTasks
 xmlns:aeb="http://schemas.active-endpoints.com/b4p/wshumantask/2007/10/aeb4p-task-state-
wsdl.xsd" >
    <htdt:getMyTasks xmlns:htdt="http://www.example.org/WS-HT/api/xsd">
       <htdt:taskType>TASKS</htdt:taskType>
```

```
        <htdt:genericHumanRole>POTENTIAL_OWNERS</htdt:genericHumanRole>
        <htdt:status>READY</htdt:status>
        <htdt:status>RESERVED</htdt:status>
        <htdt:maxTasks>5</htdt:maxTasks>
    </htdt:getMyTasks>
    <aeb:taskIndexOffset>0</aeb:taskIndexOffset>
</aeb:getTasks>
```

Normally, you will need to create the JSON object:

```
var getTasksRequest = {
  "getTasks" : {
    "xmlns" : "http://schemas.active-endpoints.com/b4p/wshumantask/2007/10/aeb4p-task-
state-wsdl.xsd",
    "getMyTasks" : {
      "xmlns" : "http://www.example.org/WS-HT/api/xsd",
      "taskType" : { "$t" : "TASKS" },
      "genericHumanRole" : { "$t" : "POTENTIAL_OWNERS" },
      "status" : [ {"$t" : "READY"} , {"$t" : "RESERVED"} ],
      "maxTasks" : { "$t" : "5"}
    },
    "taskIndexOffset" : { "$t" : "0" }
  }
}
```

With the `AE_TASK_UTIL.createGetTasksRequest()` function, the code to create the above request is:

```
var getTasksRequest = AE_TASK_UTIL.createGetTasksRequest( {
     taskType : "TASKS",
     genericHumanRole : "POTENTIAL_OWNERS",
     status : ["READY", "RESERVED"],
     maxTasks : 5,
     taskIndexOffset : 0
 });
 // getTasksRequest contains JSON for <getTasks/> element used
 // in getTasks operation for AE specific API extension at AEB4P-aeTaskOperations
 // service.

 // The WSHT getMyTasks request can be extracted in the following way:
 var wshtGetMyTasksRequest = { "getMyTasks" : getTasksRequest.getTasks.getMyTasks };
```

WSHT API

The `AeTaskApi` class (in `ae-avc-tasks.js` script) can be used to access and operate on tasks. A few common functions are shown below. Refer to `AeTaskApi` in `ae-avc-tasks.js` for additional functions and details.

- `getTasks`
- `getInstance`
- `invokeSimpleWshtRequest`
- `setOutput`

getTasks(aGetTasksRequest, aSuccessCallbackFn, aFaultCallbackFn, aErrorCallbackFn)

Invokes the `getTasks` operation for Informatica-specific API extension at `AEB4P-aeTaskOperations` service. On success, the response callback returns the JSON equivalent of the `<htdt:getMyTasksResponse/>` element.

```
 // Assign the server URL to AE_ACTIVEVOS_ENGINE_URL global variable. This is where
// the service requests are sent. (note: AE_ACTIVEVOS_ENGINE_URL is global variable
// declared in ae-avc-util.js.)
AE_ACTIVEVOS_ENGINE_URL = "http://localhost:8080/active-bpel";

// create JSON request for getTasks operation. E.g. get upto 10 unclaimed tasks.
```

```
var getTasksRequest = AE_TASK_UTIL.createGetTasksRequest( {
     taskType : AE_TASK_GLOBALS.GETMYTASKS_TASK_TYPE.TASKS, // same as string "TASKS"
                                        // same as string "POTENTIAL_OWNERS"
     genericHumanRole : AE_TASK_GLOBALS.GENERIC_HUMAN_ROLE.POTENTIAL_OWNERS,
     status : AE_TASK_GLOBALS.TASK_STATUS.READY,  // "READY"
     maxTasks : 10
});

// Create AeTaskApi object and get the task list via getTasks() operation.
var taskApi = new AeTaskApi();
taskApi.getTasks(
   getTasksRequest, // request JSON object

   function(aJsonResponse) { // success callback of JSON data
      // handle success response
      alert("Success!");
      // get list of task abstracts (//htdt:getMyTasksResponse/htdt:taskAbstract)
      var jsonTaskAbstracts =

AE_JSON_NODE_UTIL.getElements(aJsonResponse.getTasksResponse.getMyTasksResponse.taskAbstr
act);
      var i;
      for (i = 0; i < jsonTaskAbstracts.length; i++) {
         // wrap json data in AeTaskAbstract object for convenience accessors.
         // (see ae-avc-tasks.js for AeTaskAbstract)

         var taskAbstractObj = new AeTaskAbstract( jsonTaskAbstracts[i] );
         alert( "Task id: " + taskAbstractObj.getId() );
         alert( "Task name: " + taskAbstractObj.getName() );
         alert( "Task status: " + taskAbstractObj.getStatus() );
      }
   },

   function(aJsonFault) { // fault callback of JSON data
      alert( "Fault!");
   },

   function(aStatusCode, aStatusMessage) { // http error callback
      alert( "Transport error: " + aStatusCode + " " + aStatusMessage);
   }
);
```

getInstance(aTaskId, aSuccessCallbackFn, aFaultCallbackFn, aErrorCallbackFn)

Invokes the Informatica-specific `getInstance()` operation on the extension service at `AEB4P-aeTaskOperations` service. On success, the callback is an `AeTask` object (see `ae-avc-tasks.js`).

```
// ...
// setup AE_ACTIVEVOS_ENGINE_URL, and the like
// ...
var taskApi = new AeTaskApi();
taskApi.getInstance(
   "urn:b4p:1235", // task ID

   function(aTask) { // handle success response
        // aTask is an instance of AeTask (see ae-avc-tasks.js).
      alert( "Got task, id=" + aTask.getId() );
      alert( "Task name: " + aTask.getName() );
      alert( "Task status: " + aTask.getStatus() );
        // aTask.getJson() returns the underlying raw JSON data structer.
        // aTask.getOwner() returns current owner.
      var inputPartJson = aTask.getInput(); (// for input by partname, use
aTask.getInput('nameOfPart');
   },

   function(aJsonFault) {
      alert("Fault!");
   },
   function(aStatusCode, aStatusMessage) {
```

```
      alert("Transport error: " + aStatusCode + " " + aStatusMessage);
   }
);
```

invokeSimpleWshtRequest(aCommand, aTaskId, aSuccessCallbackFn, aFaultCallbackFn, aErrorCallbackFn)

Invokes a simple WSHT operation such as `claim` against the given task ID. The supported list of operation commands are defined in `AE_TASK_GLOBALS.SIMPLE_WSHT_OPERATIONS` constant.

```
var taskApi = new AeTaskApi();
var taskId = "urn:b4p:1234";
      // claim
taskApi.invokeSimpleWshtRequest(
   AE_TASK_GLOBALS.SIMPLE_WSHT_OPERATIONS.CLAIM,
   taskId,
   function(aClaimResponse) {
      // success (json data)
   },
   function(aFaultResponse) {
      // fault (json data)
   },
   function(aStatusCode, aStatusMessage) {
      //error
   }
);

// start
taskApi.invokeSimpleWshtRequest(
   AE_TASK_GLOBALS.SIMPLE_WSHT_OPERATIONS.START,
   taskId,
   function(aStartResponse) {
      // success (json data)
   },
   // ... fault and error handlers omitted for brevity ...
);

// complete
taskApi.invokeSimpleWshtRequest(
   AE_TASK_GLOBALS.SIMPLE_WSHT_OPERATIONS.COMPLETE,
   taskId,
   function(aStartResponse) {
      // success (json data)
   },
   // ... fault and error handlers omitted for brevity ...
);
```

setOutput(aTaskId, aPartName, aOutputPart, aSuccessCallbackFn, aFaultCallbackFn, aErrorCallbackFn)

Sets the task output data given the task ID, the output part name, and the output JSON data:

```
var taskApi = new AeTaskApi();
var taskId = "urn:b4p:1234";
    // message part name per wsdl
var partName = "response";
    // the <loan:loanApprovalResponse /> in JSON.
var outputJson =
   {"loanApprovalResponse":
      {
        "xmlns":
           "http:\/\/schemas.active-endpoints.com\/avoscentral\/LoanRequest\/2009\/07\/
avc-loanapproval.xsd",
        // other attributes and elements omitted for brevity.
      }
};

// set output data
taskApi.setOutput(
   taskId,
   partName,
```

```
   outputJson,
   function(aSetOutputResponse) {
      // success (json data)
   },
   function(aFaultResponse) {
      // fault (json data)
   },
   function(aStatusCode, aStatusMessage) {
      //error
   }
);
```

Accessing AeTask

The `AeTask` class (in `ae-avc-tasks.js` script) is a wrapper for the `<taskInstance />` element. This wrapper provides getters and setters for frequently used properties.

One way to get an instance of `AeTask` is by using the `AeTaskApi.getInstance(...)` function:

```
// ...
// setup AE_ACTIVEVOS_ENGINE_URL etc.
// ...
var taskApi = new AeTaskApi();
taskApi.getInstance(
   "urn:b4p:1235", // task ID

   function(aTask) { // handle success response
      // aTask is an instance of AeTask (see ae-avc-tasks.js).
      alert( "Got task, id: " + aTask.getId() );
      alert( "Task name: " + aTask.getName() );
      alert( "Task status: " + aTask.getStatus() );
      // aTask.getJson() returns the underlying raw JSON data structer.
      // aTask.getOwner() returns current owner.
   },

   function(aJsonFault) {
      alert("Fault!");
   },
   function(aStatusCode, aStatusMessage) {
      alert("Transport error: " + aStatusCode + " " + aStatusMessage);
   }
);
```

Refer to `AeTask` in `ae-avc-tasks.js` for additional functions and details. Getters and Setters to task input/output data are described below:

Functions that you can use are:

- getInputPart
- getOutputPart
- setOutputPart

getInputPart(aPartName)

Returns the task input part data given the part name (per WSDL message). If the `aPartName` is not given, the first available part is returned.

```
//
// Task is an instance of AeTask, obtained via AeTaskApi.getInstance(...)
// (assuming using the Loan Approval human task example)
//
//   <message name="WshtLoanInput">
//      <part name="request" element="loan:loanProcessRequest" />
//   </message>
```

```
//   <message name="WshtLoanOutput">
//      <part name="response" element="loan:loanApprovalResponse" />
//   </message>
var loanRequestInput = task.getInput("request"); // 'request' is the part name

// Note: task.getInput() returns first available part since part name is not given.
alert("Firt Name = " + loanRequestInput.loanProcessRequest.firstName);
alert("Loan Amount = " + loanRequestInput.loanProcessRequest.amountRequested);
```

getOutputPart(aPartName)

Returns the task out part data given the part name (per WSDL message). If the `aPartName` is not given, the first available part is returned. If part data is not available, `null` is returned.

```
var loanOutput = task.getOutput("response"); // 'response' is the output part name
// check for null in case output is not set.
if (loanOutput != null) {
   alert("Approved? = " + loanOutput.loanApprovalResponse.responseToLoanRequest);
}
```

setOutputPart(aPartName, aPartData)

Sets the output part data in the `AeTask` instance (in-memory). This method does not 'save' (invokes WSHT `setOutput` on the server). To save, you must either use `AeTask.saveOutputParts` (preferred) or `AeTaskApi.setOutput(...)`.

```
// First time use, create JSON output:
//  var loanOutput = { {"loanApprovalResponse" : ... }};
// or getOutput() to access current value if the output has already been set.
var loanOutput = task.getOutput("response");

// modify data
AE_JSON_NODE_UTIL.setText(loanOutput.loanApprovalResponse.responseDescription, "Some
Text");

// Set output (in-memory only)
task.setOutput("response", loanOutput);

// Now, save all available parts to the server using the AeTaskApi.
var taskApi = new AeTaskApi();
task.saveOutputParts(
   taskApi,
   function(aSetOutputResponse) {
      // success (json data)
   },
   function(aFaultResponse) {
      // fault (json data)
   },
   function(aStatusCode, aStatusMessage) {
      //error
   }
);
```

Task Attachment URL

Task Attachment URL

In some cases, you may want to display attachment information on a Web page. In order to do this, you need to access the list of attachment meta data (attachment infos) from the WSHT `getAttachmentInfos` operation or via the `attachmentInfo` elements that are part of `taskInstance` (obtained from the `getInstance` extension operation).

The URL to the attachment content can be built using the `AE_TASK_UTIL.getAttachmentUrl(taskId, attachmentName, attachmentId)` function.

```
//
// 1) Using getAttachmentInfos wsht API call.
//
var taskId = "urn:b4p:1235"; // task id
var getAttachmentInfosReq = {
  "getAttachmentInfos" : {
    "xmlns" : "http://www.example.org/WS-HT/api/xsd",
    "identifier" : {
      "$t" : taskId
    }
  }
};
AE_AJAX_SERVICE_UTIL.postJSON(
    // task-client service url
    "http://localhost:8080/active-bpel/services/JSON/AeB4PTaskClient-taskOperations",
    // req.
    getAttachmentInfosReq,
    // success callback of JSON data
    function(aJsonResponse) {
      // handle getAttachmentInfosResponse
      var attachmentInfoList =
AE_JSON_NODE_UTIL.getElements(aJsonResponse.getAttachmentInfosResponse.info);
      var i;
      for (i = 0; i < attachmentInfoList; i++) {
         var info = attachmentInfoList[i];
         var attachmentId = AE_JSON_NODE_UTIL.getText(info.attachmentId);
         var attachmentName = AE_JSON_NODE_UTIL.getText(info.name);  // file name
         var contentType = AE_JSON_NODE_UTIL.getText(info.contentType);  // mime
type.
         var attachUrl = AE_TASK_UTIL.getAttachmentUrl(taskId, attachmentName,
attachmentId);
         // do something with this info such as displaying on page.
         // (the attachUrl can be used for href attribute in <a> tags and src in <img>
tags)
      }
    },
    // fault callback of JSON data
    function(aJsonFault) {
       alert("Fault!");
    },
    // http error callback
    function(aStatusCode, aStatusMessage) {
      alert("Transport error: " + aStatusCode + " " + aStatusMessage);
    }
);
//
// 2) Alternate method of getting attachment info via AeTask object.
//
var taskApi = new AeTaskApi();
taskApi.getInstance(
   "urn:b4p:1235", // task ID

   function(aTask) { // handle success response
      var attachmentInfoList =  aTask.getAttachments();
      var i;
      for (i = 0; i < attachmentInfoList; i++) {
         var info = attachmentInfoList[i].attachmentInfo;
         var attachmentId = AE_JSON_NODE_UTIL.getText(info.attachmentId);
         var attachmentName = AE_JSON_NODE_UTIL.getText(info.name);  // file name
         var contentType = AE_JSON_NODE_UTIL.getText(info.contentType);  // mime
type.
         var attachUrl = AE_TASK_UTIL.getAttachmentUrl(aTask.getId(), attachmentName,
attachmentId);
         // do something with this info such as displaying on page.
         // (the attachUrl can be used for href attribute in <a> tags and src in <img>
tags
      }
```

```
    },

    function(aJsonFault) {
       alert( "Fault!");
    },
    function(aStatusCode, aStatusMessage) {
       alert( "Transport error: " + aStatusCode + " " + aStatusMessage);
    }
);
```

Working with Schema Date and Time Types

Working with Schema Date and Time Types

In cases where the request and response contains `xsd:dateTime` type elements, the contents of these elements take the form `YYYY-MM-DDTHH:MM:SS.sssZ` (UTC time). However, the user interface in a browser deals with local time instead of UTC. You can use the following functions in the `AE_UTIL` object in the included `ae-avc-utils.js`.

Function	Description
`AE_UTIL.xsdDateTimeToJsDateTime(xsdDateTimeStr)`	

	Parses the ISO8601 xsd dateTime (YYYY-MM-DDTHH:MM:SS.sssZ) string and returns JavaScript Date object.
	
	`// task created on date. E.g: "2010-03-15T17:15:00.881Z" (12.15PM Eastern Time/ GMT-5).`
	`var createdOnXsdStr = AE_JSON_NODE_UTIL.getText(taskAbstracts[i].createdOn);`
	`// convert to a JavaScript date object.`
	`var createdDate = AE_UTIL.xsdDateTimeToJsDateTime(createdOnXsdStr);`

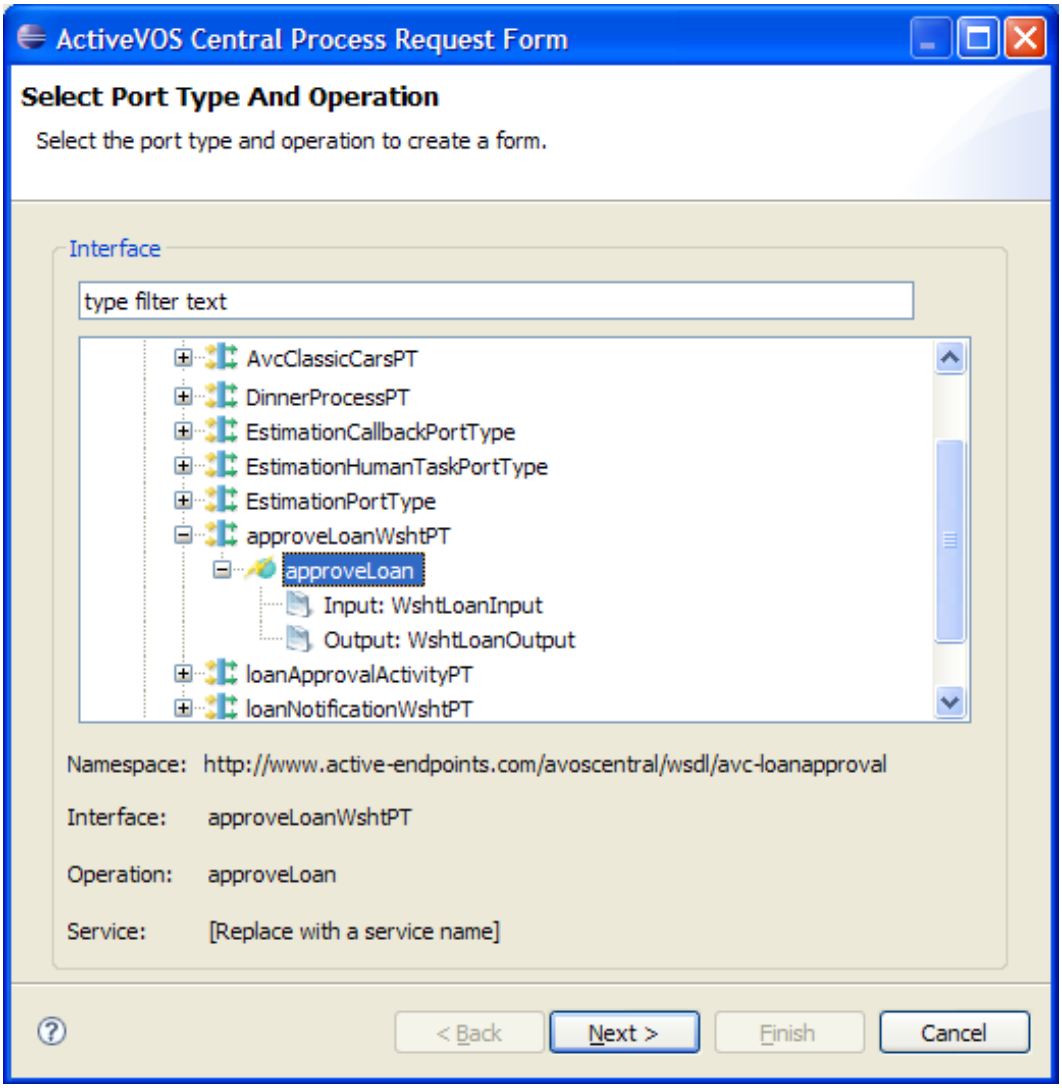
`AE_UTIL.xsdDateToJsDate(xsdDateStr)`	
	Parses the xsd date (YYYY-MM-DD) string and returns JavaScript Date object.
`AE_UTIL.xsdTimeToJsTime(xsdTimeStr)`	
	Parses the XSD time (HH:MM:SS) string and returns JavaScript Date object. The returned date object contains the parsed time (but uses current date).
`AE_UTIL.toXsdDateTime(date)`	
	Converts a JavaScript Date object into a ISO8601 dateTime string (YYYY-MM-DDTHH:MM:SS.sssZ).
`AE_UTIL.toXsdDateTimeFromDateAndTimeStr(date, time)`	

Function	Description
	The date and time parameters are both JavaScript date objects. This function converts a date and time JavaScript Date objects into a ISO8601 dateTime string (YYYY-MM-DDTHH:MM:SS.sssZ).
`AE_UTIL.toXsdDate(date)`	
	Converts a JavaScript Date object into a ISO8601 date string (YYYY-MM-DD). Timezone information is not used.
`AE_UTIL.toXsdTime(date)`	
	Converts a JavaScript Date object into a ISO8601 time string (HH:MM:SS). Timezone information is not used.

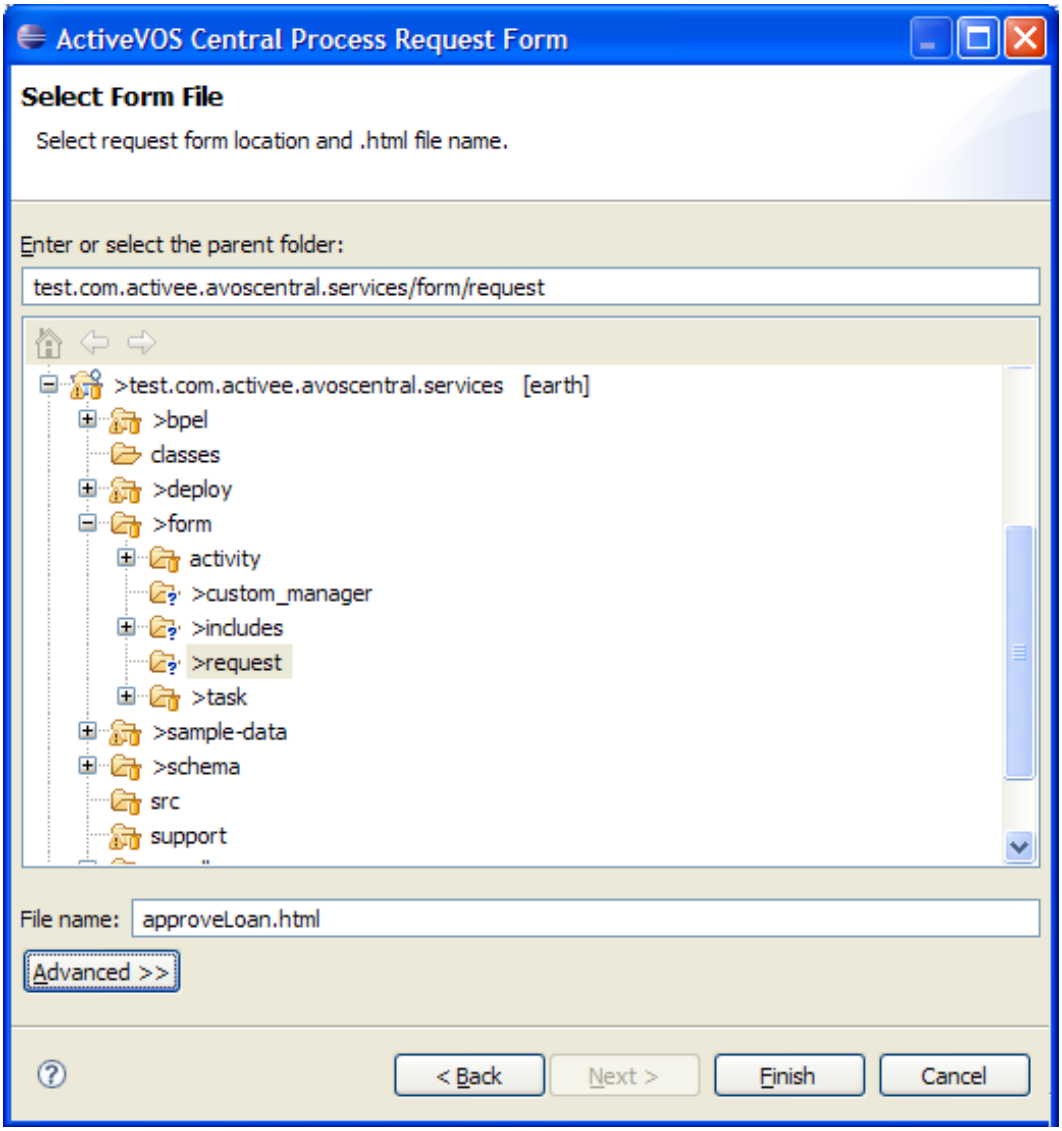
When working with forms that use date and time types, first convert the raw JSON date/time information (ISO8601) to a JavaScript Date object before presenting it to the user (for example, using `AE_UTIL.xsdDateTimeToJsDateTime(...)`). Conversely, the local date/time information from the user should be first converted to a ISO8601 (for exmple, using `AE_UTIL.toXsdDateTime(...)`) before sending it to Process Server.

Using Process Developer to Create HTML Forms

You can create HTML forms and basic supporting JavaScript using ActiveVOS Designer. The `Process Request Form` action available under File > New > Other > Orchestration brings up a wizard style dialog. From this dialog, you can choose your request WSDL, port-type, operation and the destination where the generated HTML file is saved.

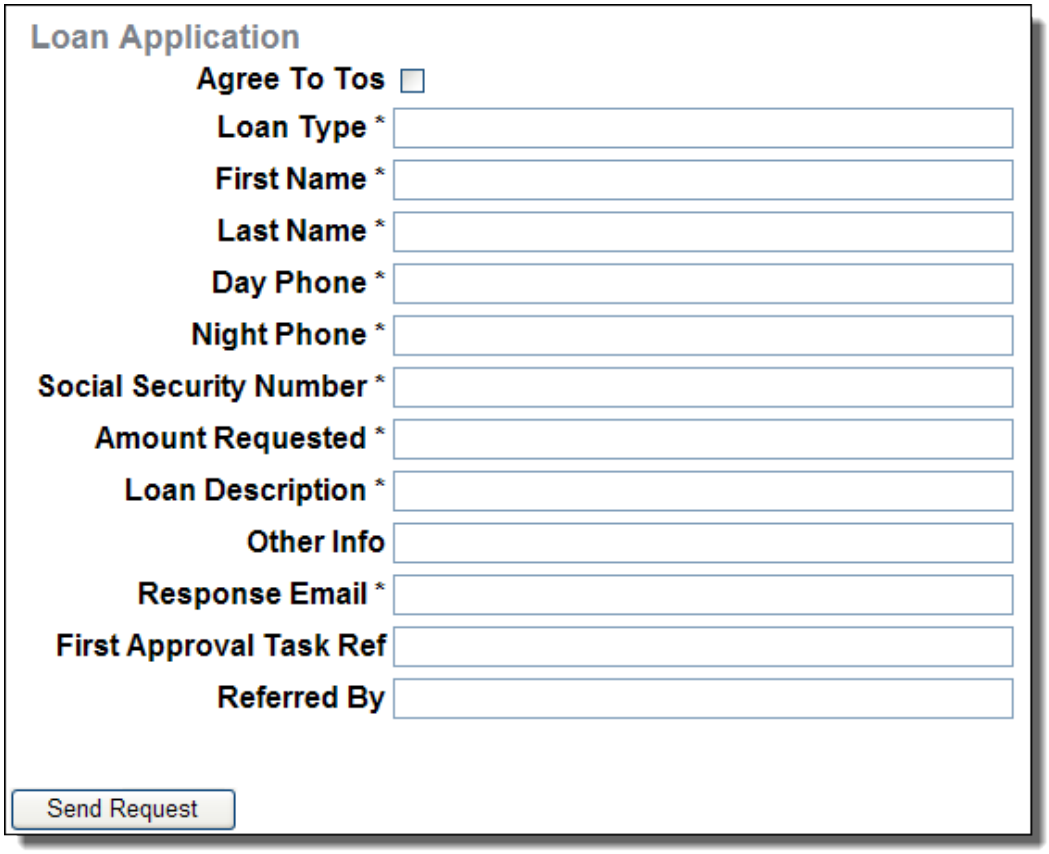


Select the WSDL port-type and operation.



Select the location where the HTML file will be saved.

The generated html forms looks similar to (in the case of Loan Approval process):



The basic structure of a generated request form HTML looks like the following:

```
<html>
   <head>
     <!-- header content -->
   </head>
   <body>
     <!--
           Main DIV that contains all markup
           related to this request form UI
     -->
     <div id="processRequestForm$ID">
         <!-- DIV that contains UI related request form and data entry -->
         <div id="processRequestForm$ID">

             <div>Display Name UI</div>

             <!-- container for the form UI controls -->
             <div>

               <form id="loanApplicationInputForm$ID">
                  <!-- actual form content, for example: -->
                  First Name: <input id="firstName$ID" name="firstName" value=""
size="50" /> <br/>
                   Last Name: <input id="lastName$ID"  name="lastName" value=""
size="50" /> <br/>
               </form>

               <!-- Send button -->
               <input type="button" id="sendRequest$ID" value="Send Request" />

             </div>

         </div>
```

```
            <!--
                 DIV that contains html to show
                 the results after submitting a form (invoking a process).
                 Note this DIV is initially hidden, and only shown when
                 response data needs to be displayed.
            -->
            <div id="responseContainer$ID" style="display: none;">
            </div>
        </div>

        <!-- Script -->
        <script type="text/javascript">
          // <![CDATA[
          // JavaScript to implement JSON process invoke
          // ]]>
        </script>

      </body>
    </html>
```

The HTML above is a skeleton used that illustrates various elements used in Process Developer generated forms. The actual form may be a little more detailed and is targeted for use with jQuery and jQueryUI. Note that all elements used in the form have an `id` attribute. This attribute is used as the primary selector when Process Server needs to access various elements within the HTML document. Also note that the `id` attribute values that are generated end with the `$ID` suffix (for example `processRequestForm$ID` in `<div id="processRequestForm$ID">`).

All elements in an HTML document must have unique values for the `id` attribute. In cases where you need to display the same form more than once (a requirement of Process Central), the elements cannot be duplicated with the same values for the `id` attribute. When the forms are used with Process Central, the forms server automatically replaces the `$ID` suffix with a unique value so that a form can be cloned and used in the same HTML document. For example, at runtime Process Central may have two loan request form instances `<div id="processRequestForm_01">` and `<div id="processRequestForm_02">` (note where `$ID` has been replaced with `_01` and `_02` in this example).

Since this document pertains to using single instance of forms (outside of Process Central), you can use the generated html as is after performing a search-and-replace of `$ID` suffix with any string that does not have a `$`. For example, if all occurrences of `$ID` is replaced with an empty value, you essentially remove the `$ID` suffix; for example, `<div id="processRequestForm$ID">` becomes `<div id="processRequestForm">`.

Using jQuery form input data can be used with the `id` selectors. For example, getting the current value of the `firstName` input field is `$("#firstName$ID").val()`. The field value can be prepopulated with data using `$("#firstName$ID").val("John")`. The complete script that drives the form can be found at the end of the form in the `<script>` section. The basic mechanics of the script looks similar to following skeleton code :

```
// The jQuery 'ready' event is fired once the page (form) is loaded and ready for
processing.
$(document).ready(function() {
   // bind button click event handler code when the Send button is pressed.
   $("#sendRequest$ID").click( function() {
       sendForm();  // see fn below.
   });
});
// send form
function sendForm() {
  // first grab data entered by user.

  var firstNameStr = $("#firstName$ID").val();
  // ..
  // .. get other UI data such as loanType, lastName, phone etc.
  // ..

  // build JSON request
  var loanRequestJSON = {"loanProcessRequest":
```

```
      {"xmlns":"http:\/\/schemas.active-endpoints.com\/sample\/LoanRequest\/2008\/02\/
loanRequest.xsd",
       "loanType":{"$t": loanTypeStr},
       "firstName":{"$t":  firstNameStr},
       "lastName":{"$t":  lastNameStr},
       "dayPhone":{"$t":  dayPhoneStr},
       "nightPhone":{"$t":  nightPhoneStr,
       "socialSecurityNumber":{"$t":  ssnStr},
       "amountRequested":{"$t":  loanAmount},
       "loanDescription":{"$t":  loanDesc},
       "responseEmail":{"$t":  email}
      }
  };

  // JSON endpoint URL (for example,
  //   http://localhost:8080/active-bpel/services/JSON/humantaskProcessDemoService)
  var loanServiceUrl =
      AE_AJAX_SERVICE_UTIL.getActiveVOSServiceUrl("JSON/humantaskProcessDemoService");

  // send JSON request
  AE_AJAX_SERVICE_UTIL.postJSON(
    loanServiceUrl,
    loanRequestJSON,
    function(aJsonResponse) {
      // success result handler:
      // show <status/> element text in the response div.
      var statusText = AE_JSON_NODE_UTIL.getText(aJsonResponse.status);
      // (above is same as aJsonResponse.status.$t)

      // append the status text string to the <div id="responseContainer$ID" /> div
      $("#responseContainer$ID").html("<p>Status=" + statusText + "</p>");

      // show the div (since it was initially hidden)
      $("#responseContainer$ID").show();
    },
    function(aJsonFault) {
       alert("Fault!");
    },
    function(aStatusCode, aStatusMessage) {
      alert("Transport error: " + aStatusCode + " " + aStatusMessage);
    }
  );
}
```

This code demonstrates the approach taken to handle form submissions. The actual code in the generated form is a little more detailed, but still follows this methodology. In the generated code, the script related to a form is enclosed in a function:

```
// Function that encapsulates the form script.
var AeRequestForm$ID = function() {
     // variable defining the name of service
     var mServiceName = "humantaskProcessDemoService";

     //
     // internal functions: pseudo code shown below for brevity
     //

     // This function is called when the form is loaded.
     function documentReady() {
       // initialize and populate UI here.
       // code to bind the SendButton click event to invoke _submitRequest() function
       // also invoke _setupValidation();
     }
     // function is called (by documentReady() )
     function _setupValidation() {
       // optional code to setup & initialize your form data validation
     }

     // Function returns the JSON data structure for this operation
     function getInputRequest() {
```

```
        // code that creates JSON object. For example:
        // var json = { "loanProcessRequest": {...} };
        // return json;
      }

      // function called when the Send Request button is pressed.
      function _submitRequest() {
        // 1. validate the form by calling avcform_validate().
        // 2. var jsonReq = getInputRequest();
        // 3. get form UI data and populate jsonReq object
        // 4. serviceUrl = AE_AJAX_SERVICE_UTIL.getActiveVOSServiceUrl("JSON/" +
mServiceName);
        // 5. invoke json request via:
        //      AE_REQUESTS_UTIL.postJSON(serviceUrl, jsonReq,
_showResponseCallback,  ....);
        //
      }

      // validate the form
      function avcform_validate() {
       // check form date and validate (e.g. verify required fields have data etc.)
       // return true if user submitted data is valid
      }

      // Called by postJSON(..) code in _submitRequest() function.
      function _showResponseCallback(aJsonResponse) {
       // called to display response from a json invoke
      }

      // Called by postJSON(..) code in _submitRequest() function.
      function _showFaultCallback(aJsonFault) {
       // handle fault
      }

      // Called by postJSON(..) code in _submitRequest() function.
      function _communicationErrorCallback(aStatusCode, aError) {
       // error handler code
      }
}
// The jQuery 'ready' event is fired once the page (form) is loaded
// and ready for processing.
$(document).ready(function() {
   // Create Request Form JavaScript object
   var requestForm = new AeRequestForm$ID();
   // initialize form
   requestForm.documentReady();
});
```

Making Cross Domain AJAX Requests

Cross Domain Proxy

One approach to making cross domain AJAX request is to use a proxy. In this scenario, your script calls an endpoint on your Web application server that is hosting your script and application. That endpoint in turn forwards (proxies) all requests to the actual destination (for example, the Process Server engine where your services are hosted).

JSON with Padding (JSONP)

When making JSONP request, the string version of the JSON data must be sent in a parameter named `_json` and the callback function name using parameter named `callback` (if a value is not given, the engine does a callback to `jsonpCallback()`). Since JSONP requests are based on the HTTP GET method, the amount of

data that can be sent in a HTTP query string is limited. This means, JSONP requests are better suited for request messages whose payload is small. Note that attachments are not supported with JSONP.

The basic format of a JSONP request to service deployed on ProcessaServereis:

```
http://host:port/active-bpel/services/JSON/serviceName?callback=s
         callbackFnName&_json=string_version_of_json
```

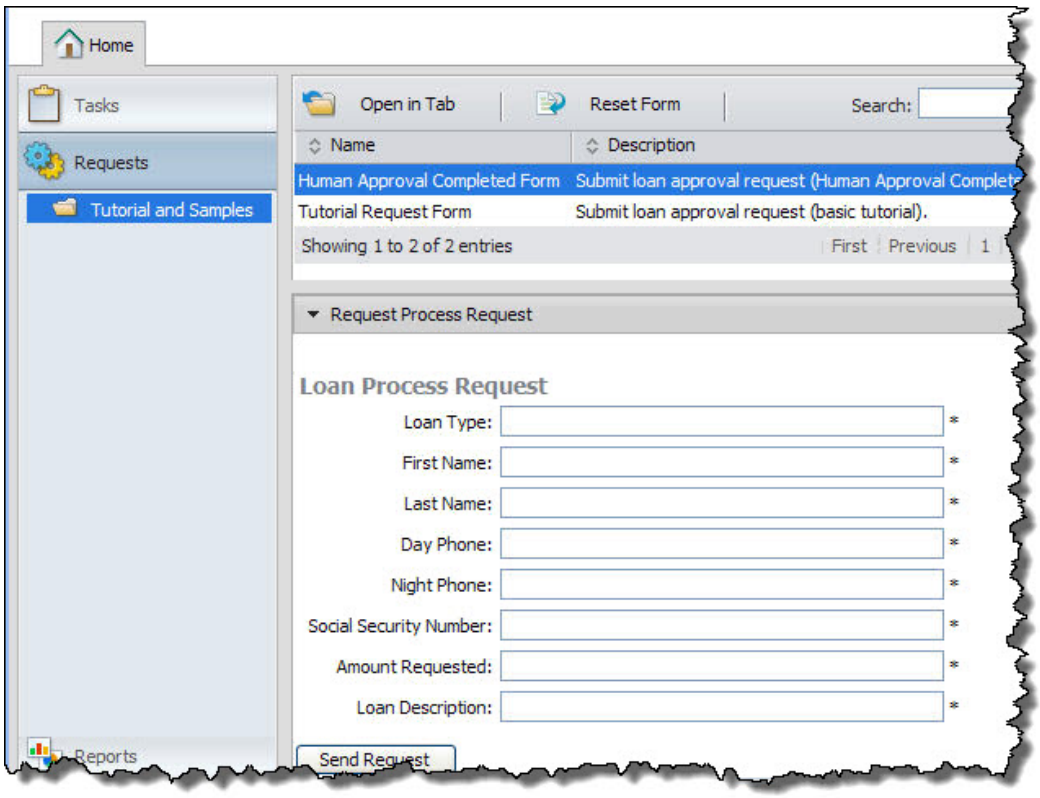
You can use the helper function `AE_AJAX_SERVICE_UTIL.getJSON(...)` for JSONP.

```
var getTasksRequest = {
    "getMyTasks" :
      {
       "xmlns" : "http://www.example.org/WS-HT/api/xsd",
       "taskType" : { "$t" : "TASKS" },
       "genericHumanRole" : { "$t" : "POTENTIAL_OWNERS" },
       "status" : [ {"$t" : "READY"} , {"$t" : "RESERVED"} ],
       "maxTasks" : { "$t" : "5"}
      }
};

// get task list using JSONP
AE_AJAX_SERVICE_UTIL.getJSON(
     // task-client service url
     "http://localhost:8080/active-bpel/services/JSON/AeB4PTaskClient-taskOperations",
     // req.
     getMyTasksRequest,
     // success callback of JSON data
     function(aJsonResponse) {
       alert("Success!");
       // handle result. e.g. display results in a table
     },
     // fault callback of JSON data
     function(aJsonFault) {
        alert("Fault!");
     },
     // http error callback
     function(aStatusCode, aStatusMessage) {
       alert("Transport error: " + aStatusCode + " " + aStatusMessage);
     }
);
```

CHAPTER 7

XML-JSON for Process Central

Process Central is a client application that contains Process Request Forms, Tasks, and Reports. Users of Process Central can start a process, work on tasks, and view reports.



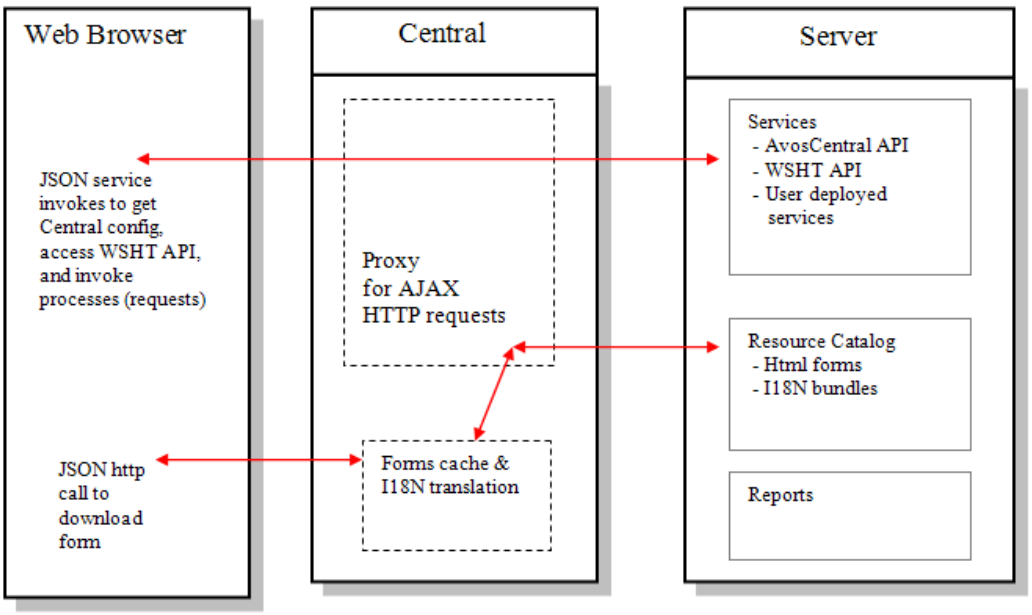
Process Central has the following feature applications that are accessible by selecting an accordion control:

- **Tasks**: Displays list of tasks available to the current user. Once a task is selected, the user is presented with the task form and a set of action UI controls to operate on the task (such as claim, save, and complete).
- **Requests**: Displays a list of forms that can invoke processes. This list is based on configurations done on the Process Server.
- **Reports**: Displays a list of reports.

Component Interaction

Process Central is built on using services exposed by the Process Server. The interaction that occur are shoen in the following figure. Process Central relies primarily on the `AvosCentralApi` service (for configuration) and the WS-HumanTask (WSHT) taskClient services (for tasks) along with access to resources

from the Process Server Catalog using HTTP GETs. The `AvosCentralApi` is a Process Server system service that exposes a few operations to provide Process Central configuration information based on the current users' role. Process invokes made by the browser (nearly all AJAX requests) are sent to the server using a proxy servlet. This is required to support cross-domain asynchronous requests by the browser in cases where Process Central and Process Server are installed on separate hosts



Login

When a user (with a role of `abTaskClient`) logs into Process Central, the login code makes an JSON service invoke to the `AvosCentralApi` service at endpoint `/services/JSON/AvosCentralApi`. Because the endpoint `/services/JSON/AvosCentralApi` (in the server) is secured, the application server (for example, Tomcat) intercepts the request to authenticate the user. Once the user is authenticated and authorized (has role `abTaskClient`), access to the `/services/JSON/AvosCentralApi` service is granted. Once authenticated, the login operation uses the Process Identity system service

Central Configuration Information

After the user logs in, the main page displays the Tasks, Requests, and Reports sections. For example, in the case of the Requests section the list of available requests organized by category as folders are displayed. The configuration containing the list of requests and categories are obtained from the Process Server using the `AvosCentralApi` service. The Request configuration (for example) list includes location of the Request HTML form in the server catalog.

Displaying Forms

When a user selects a Request (for example), the browser's JavaScript code makes an HTTP GET request to Process Central to download the HTML form content (for example, `project:/path/form.html`). If the form is not available locally, Process Central retrieves it from Process Server catalog (using HTTP GET). Before returning the HTML form content to the browser, Process Central may (if translation is available) replace internationalization (I18N) tokens contained in the form after downloading the appropriate I18N bundle from the server's catalog. The `$ID` suffix in HTML element names, attributes, or in JavaScript are replaced with a unique string (an integer for Request forms, and `taskID` for Task forms).

Once the HTML form content is loaded by the JavaScript executing the Request UI, the form is inserted into Process Central's UI (for example, the preview area) so that the form is visible and accessible to the end user.

Submitting Forms

Submitting a Request form consists of creating a JSON request with the form data and sending it the appropriate JSON service endpoint to invoke the process.

Caching

Process Central obtains all forms from the Process Server catalog and caches them. If an updated version of a form is redeployed to the server, Process Central will only request it the next time a user logs. During development, you may need to logout and log back in to see the revised form unless you use Process Central in debug mode.

Process Request Forms

Process request forms are essentially HTML forms with JavaScript that are used to invoke a process. The skeleton markup of a request form shown below illustrates various elements used by the form. The actual form may be a little more detailed and is targeted for use with jQuery and jQueryUI.

```
<html>
   <head>
     <!-- header content -->
   </head>
   <body>
     <!--
           Main DIV that contains all markup
           related to this request form UI
     -->
     <div id="processRequestForm$ID">
         <!-- DIV that contains UI related request form and data entry -->
         <div id="processRequestForm$ID">

             <div>Display Name UI</div>

             <!-- container for the form UI controls -->
             <div>

               <form id="loanApplicationInputForm$ID">
                  <!-- actual form content, for example: -->
                  First Name: <input id="firstName$ID" name="firstName" value=""
size="50" /> <br/>
                   Last Name: <input id="lastName$ID"  name="lastName" value=""
size="50" /> <br/>
               </form>

               <!-- Send button -->
               <input type="button" id="sendRequest$ID" value="Send Request" />

             </div>

         </div>
         <!--
              DIV that contains html to show
              the results after submitting a form (invoking a process).
              Note this DIV is initially hidden, and only shown when
              response data needs to be displayed.
         -->
         <div id="responseContainer$ID" style="display: none;">
         </div>
     </div>

     <!-- Script -->
     <script type="text/javascript">
       // <![CDATA[
```

```
        // JavaScript to implement JSON process invoke
        // ]]>
      </script>

   </body>
</html>
```

All elements used in the form have an `id` attribute. The `id` attribute is used as the primary selector when you need to access various elements within the HTML document. Also note that the `id` attribute values that are generated end with the `$ID` suffix (for example `processRequestForm$ID` in `<div id="processRequestForm $ID">`).

In cases where Process Central needs to display the same form more than once, the element `id` attribute cannot be duplicated with the same values. When the forms are used with Process Central, the forms server automatically replaces the $ID suffix with a unique value. This allows the form to be cloned and used in the same HTML document. For example, at runtime Process Central may have two loan request form instances `<div id="processRequestForm_01">` and `<div id="processRequestForm_02">` (note that `$ID` has been replaced with `_01` and `_02` in this example).

The JavaScript that is generated with the HTML form (in the `<script />` block at the end of the form) is enclosed in a function for scoping and takes the general structure shown below:

```
// Function that encapsulates the form script.
var AeRequestForm$ID = function() {
     // variable defining the name of service
     var mServiceName = "humantaskProcessDemoService";

     //
     // internal functions: psuedo code shown below for brevity
     //

     // This function is called when the form is loaded.
     function documentReady() {
       // initialize and populate UI here.
       // code to bind the SendButton click event to invoke _submitRequest() function
       // also invoke _setupValidation();
     }
     // function is called (by documentReady() )
     function _setupValidation() {
       // optional code to setup and initialize your form data validation
     }

     // Function returns the JSON data structure for this operation
     function getInputRequest() {
       // code that creates JSON object. For example:
       // var json = { "loanProcessRequest": {...} };
       // return json;
     }

     // function called when the Send Request button is pressed.
     function _submitRequest() {
       // 1. validate the form by calling avcform_validate().
       // 2. var jsonReq = getInputRequest();
       // 3. get form UI data and populate jsonReq object
       // 4. serviceUrl = AE_AJAX_SERVICE_UTIL.getActiveVOSServiceUrl("JSON/" +
mServiceName);
       // 5. invoke json request via:
       //      AE_REQUESTS_UTIL.postJSON(serviceUrl, jsonReq,
_showResponseCallback,  ....);
       //
     }

     // validate the form
     function avcform_validate() {
      // check form date and validate (e.g. verify required fields have data etc.)
      // return true if user submitted data is valid
     }
```

```
      // Called by postJSON(..) code in _submitRequest() function.
      function _showResponseCallback(aJsonResponse) {
       // called to display response from a json invoke
      }

      // Called by postJSON(..) code in _submitRequest() function.
      function _showFaultCallback(aJsonFault) {
       // handle fault
      }

      // Called by postJSON(..) code in _submitRequest() function.
      function _communicationErrorCallback(aStatusCode, aError) {
       // error handler code
      }
}
// The jQuery 'ready' event is fired once the page (form) is loaded
// and ready for processing.
$(document).ready(function() {
   // Create Request Form JavaScript object
   var requestForm = new AeRequestForm$ID();
   // initialize form
   requestForm.documentReady();
});
```

When the form is downloaded by Process Central so that it can be displayed to the user, the browser renders the HTML form and begins to execute the JavaScript associated with the form (see `$(document).ready(..)`). The entry point is the `documentReady()` function in the form JavaScript object that is responsible for initializing and displaying the form.

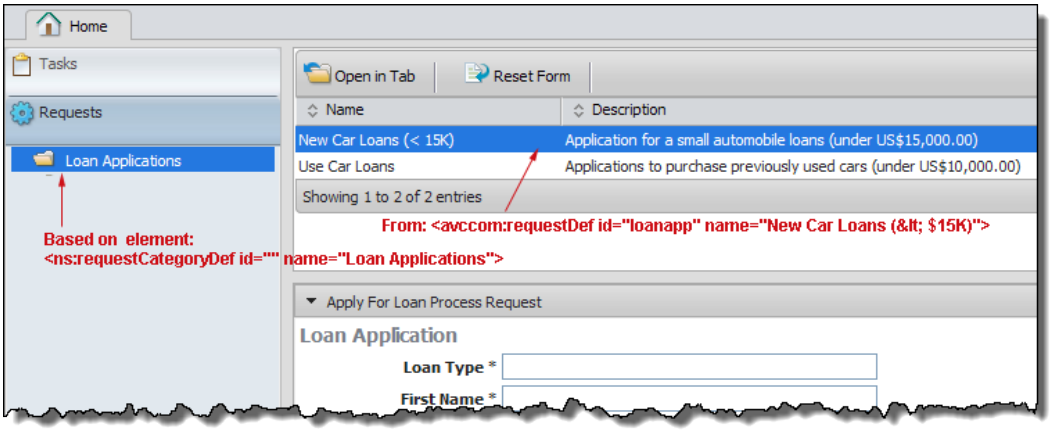
Deploying Request Forms

The HTML forms are deployed to the Process Server resource catalog on the server and are accessible using the `http://host:port/active-bpel/avccatalog/FORM_LOCATION` URL where `FORM_LOCATION` is the location of the form with in the `caXStalog`. (for example, `project:/proj_name/form/request/loanform.html`). In order for Process Central to discover these forms, an `.avcconfig` (Process Central) configuration file must also be deployed to the catalog.

```
<ns:avosCentralConfiguration
    xmlns:ns="http://schemas.active-endpoints.com/avc/2009/07/avoscentral-config.xsd">
   <!-- All Requests are defined in requestCategoryDefs element -->
  <ns:requestCategoryDefs>

     <!--
        requestCategoryDef is represented as a Folder in Central.
        Add additional requestCategoryDef elements to add more 'Folders'.
     -->
    <ns:requestCategoryDef id="" name="Loan Applications">

      <!--
           A folder (category) can have one or more Requests  (requestDef).
           Each requestDef has a unique id, display name, description and
           HTML form location. Central uses this location hint to download
           the actual form and display it.
      -->
      <avccom:requestDef id="loanapp" name="New Car Loans (&lt; $15K)">

         <!--
           You can restrict who sees this Request by specifying one more roles.
           The value should be a group name or a username.
           For example, this request is visible only to members of
           group 'group_A' and user 'jsmith'.
         -->
         <avccom:allowedRoles>
            <avccom:role>group_A</avccom:role>
            <avccom:role>jsmith</avccom:role>
         </avccom:allowedRoles>

         <avccom:description>Application for a small automobile loans (under US
$15,000.00)</avccom:description>
```

```
            <avccom:formLocation>project:/LoanApproval/form/request/loanForm.html</
avccom:formLocation>

        </avccom:requestDef>
      </ns:requestCategoryDef>
    </ns:requestCategoryDefs>
</ns:avosCentralConfiguration>
```

When a user logs in, Process Central fetches an aggregate list of Request categories and requests that the user can access (based on `<avccom:allowedRoles>`) and displays the folders and requests.



Validation

Process Central uses jQuery and the jQuery Validation plugin (see `http://docs.jquery.com/Plugins/Validation`) to perform form data validation. By default, all text input fields are considered to be string type optional fields. In cases where the input text field is required or is not a string (for examplem a number or date), the generated form has its input text fields annotated with CSS class names:

- `avc_required_field`: Makes a field required; for example, `<input id="loanAmount" type="text" class="avc_required_field" />`.
- `avc_type_number`: Checks to see if the data represents a number.
- `avc_type_date`: Validates the text field as a date string. The format depends on the locale.
- `avc_type_time`: Checks to see if the input field contains a time value in HH:MM:SS format.

In addition to these validation CSS classnames, you can use any of the validation classnames that are built into the jQuery Validation plugin. For example `required` (same as `avc_required_field`), `email`, and `url`.

The two main functions within the Request form JavaScript that are related to validation are `_setupValidation()` and `avcform_validate()`. The `_setupValidation()` is where the form validation is initialized as well as where additional validation rules are added. For example, if the HTML form contains an element that is a number with XSD schema `minlength` and `maxlength` rules, the set up code may look like:

```
// Note: this method is auto-generated.
var _setupValidation = function() {
   // bind form to the plugin
   $("#loanForm$ID").validate();

   // limit loan amount input field between 1000 and 5000.
   // (see jQuery Validation Plugin for docs).
   $("#loanAmount$ID").rules("add", {maxinclusive: 5000, mininclusive: 1000});
```

```
    // other validation rules ....
}
```

The setup function and the additional rules in it are normally automatically generated based on the form's schema (XSD) document. The `avcform_validate()` function is called prior to the form's submission (for example, when the user presses the Send Request button). The code in this form should return `true` if the form is valid and Request should be sent to the server. The Request is not sent to the server if this method returns `false`.

```
// Note: this method is auto-generated.
var avcform_validate = function() {
    // let the jQuery form validation plugin first do the basic validation
    if(!$("#loanForm$ID").validate().form()) {
        // return false if not valid
        return false;
    }
    // do further validation if needed (e.g. if A is selected then B and C is required).
    // return t/f
    return true;
}
```

For more information, go to `http://docs.jquery.com/Plugins/Validation`, which describes the query validation plugin.

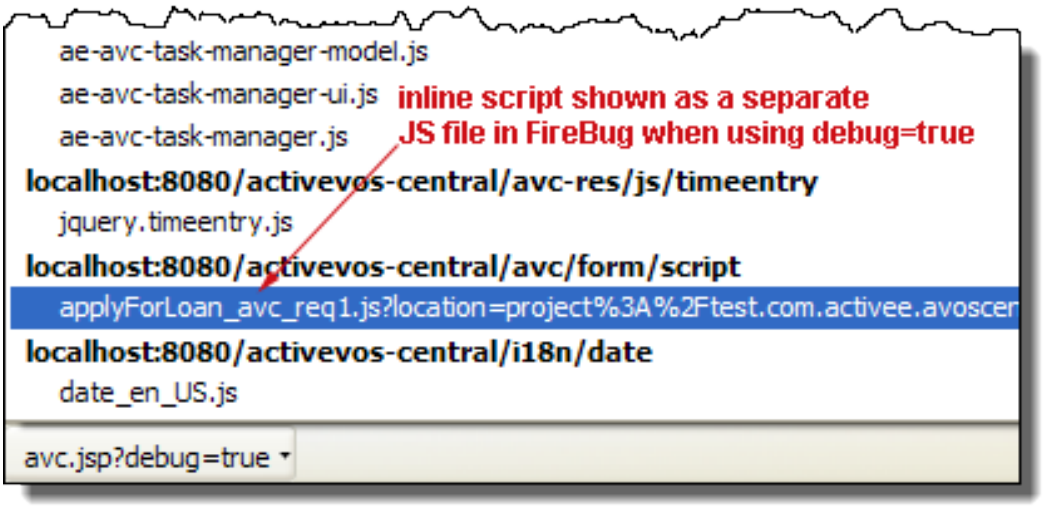
Debugging Request Forms With debug=true QueryString

At some point during development, you may need to debug Process Central forms and scripts. To begin debugging, Process Central must:

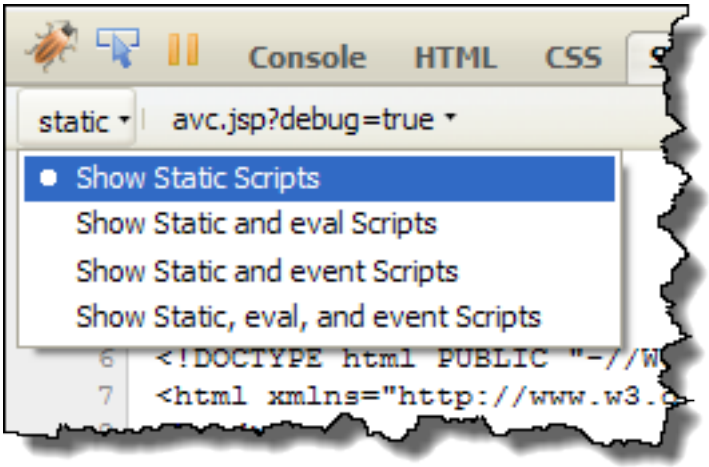
1. Switch off the Form Cache: Process Central caches forms downloaded from the Process Server catalog. When a form (HTML, script or both) is updated and deployed to the catalog, you will need to logout and login again so that Central will clears its cache and downloads the latest version of the HTML form (instead of using an older version). Switching off the cache forces Process Central to bypass the cache.
2. Split Form HTML and Scripts: A Process Central form consists of HTML content inlined with JavaScript code. When the form is loaded into the browser, it is not easy to debug (for example, you cannot set breakpoints in the code) the JavaScript since the code is run in anonymous `eval()` functions and is not visible (or easy to find). To work around this issue, the browser must load the JavaScript (that is embedded in the form) as a separate resource (file).

You can do this if you load the Process Central page with `debug=true` in the query-string (for example, `http://localhost:8080/activevos-central/avc/avc.jsp?debug=true`). When the page is loaded with `debug=true`, Process Central by-passes the forms cache and downloads HTML forms from Process Server for every single request (that is, it always fetches the latest copy). It also serves the JavaScript embedded in form as a separate resource to the browser. Making available the inline JavaScript as a separate resource lets you set break points in your code using a script debugger such as FireBug (for FireFox), Developer Script Console (for Chrome), or Developer Tools (for Internet Explorer 8).

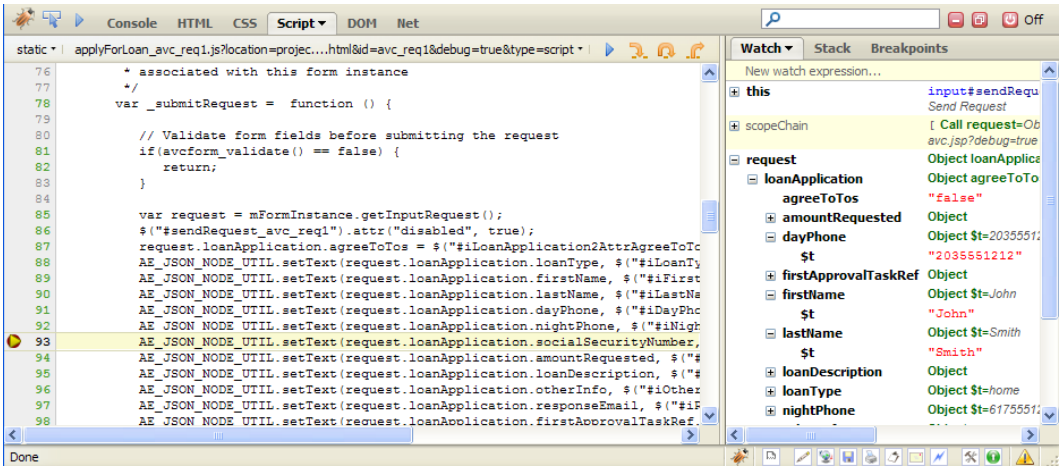
The scripts in FireBug JavaScript debugger are available for debugging using the Script tab. The following figure shows a Request form named `applyForLoan_avc_req1` (which is a runtime name; the actual name in the HTML is `applyForLoan$ID`) listed in FireBug's script menu. If you do not see the script associated with a specific form, press the up and down arrow keys (on the FireBug UI) to scroll through the list of available scripts.



If you see too many scripts, (for example, `eval()` functions), narrow down the list of scripts that is displayed to 'static' files. This option is available in the left drop down menu in FireBug:



Finally, the following FireBug figure is at a break point in the loan application form JavaScript code.



Process Central Includes

When Requests (and Task) forms are displayed in Process Central, they only use the contents of the `<body>` in the HTML form. The HTML elements in the `<head>` are not used. This means any references to `<meta>`, `<script>`, `<link>`, and `<style>` elements are not included in Central's main page.

```
<head>
  <!--
    Include 3rd party script to show tree control plugin.
    This example requires the jQuery treeview script and the css.
  -->
  <link  type="text/css" href="js/treeview/jquery.treeview.css" rel="stylesheet" />
  <script type="text/javascript" src="js/treeview/jquery.treeview.js"/>

  <!-- Include your custom or common scripts that is shared with other forms -->
  <script type="text/javascript" src="js/some_util.js"/>

  <!-- Inline CSS -->
  <style type="text/css">
     p {margin-left: 2em;}
  </style>

  <!-- Inline common script -->
  <script type="text/javascript">
     // execute your code. e.g. jQuery
     $(document).ready( function() {
         // do something
     });
  </script>
</head>
<body>

</body>
```

If the HTML form is as shown above, Process Central discards the contents of the `<head>`, which means your custom script and CSS including third party scripts and CSS (for example, jQuery tree-view) are not loaded and executed. This is because the page that is displaying Central only deals with form content (body). In order to inject (or contribute) custom scripts and style into the main page that is displayed by Process Central, the custom scripts and CSS must be specified by using an include within the `.avcconfig` file. With Central Includes, all custom scripts, link, styles, and meta elements contributed by one or more forms are aggregated and dynamically included in `<head>` of the main display page of Central.

The Central Includes for the above sample HTML is shown below in the `<centralIncludes>` element (in an `.avcconfig` file).

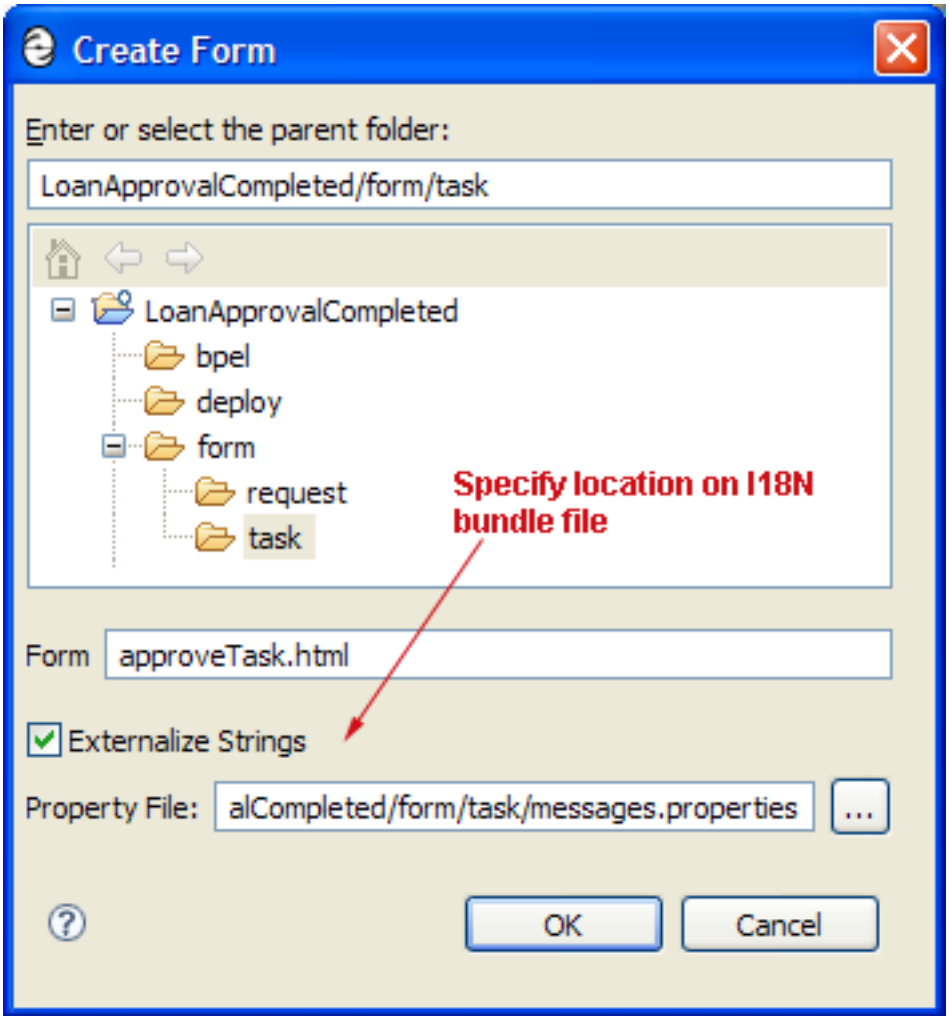
```
<!-- Central Includes in the .avcconfig file -->
<tns:centralIncludes>
  <!--
      Contents of the xhtml head go here as-is, with xhtml namespace.
      The path to resources must be absolute href or relative
      to this .avcconfig file. E.g. js/treeview/jquery.treeview.css
  -->

  <!-- Third party library (jQuery treeview for example)  -->
  <link xmlns="http://www.w3.org/1999/xhtml" t
        type="text/css" href="js/treeview/jquery.treeview.css" rel="stylesheet" />
  <script xmlns="http://www.w3.org/1999/xhtml
        type="text/javascript" src="js/treeview/jquery.treeview.js"/>

  <!-- Custom scripts -->
  <script xmlns="http://www.w3.org/1999/xhtml"
        type="text/javascript" src="js/some_util.js"/>

  <!-- Inline CSS -->
  <style xmlns="http://www.w3.org/1999/xhtml" type="text/css">
```

```
      p {margin-left: 2em;}
    </style>

    <!-- Inline common script -->
    <script xmlns="http://www.w3.org/1999/xhtml" type="text/javascript">
       // execute your code. e.g. jQuery
       $(document).ready( function() {
           // do something
       });
    </script>

  </tns:centralIncludes>
```

Note: The XHTML elements `<meta>`, `<script>`, `<link>`, and `<style>` declared in the `<centralIncludes>` must be in the XHTML namespace (`http://www.w3.org/1999/xhtml`). When deploying the `.avcconfig` file, all resources referenced in the Central Includes (for example, JavaScript files, CSS files) must also be deployed to the server.

Internationalization

Form content can be externalized into standard bundle property files when the form (Request or Task) is first created by selecting the Externalized String option and entering the file name and path to where the bundle is saved. A bundle file (for example `messages.properties`) is automatically created when the form is generated.



The generated HTML contains entries in the form `${bundle_key_name}`. For example, in the case of the Loan Approval form, the label for the First Name is given as:

```
<head>
   ...
   <!--  Specifies the location of the bundle file relative to the HTML form -->
   <link rel="i18n" charset="UTF-8" type="text/plain" href="messages.properties" />
   ...
</head>
<body>
   ...
   <tr>
      <!--  bundle entry for first name -->
      <th align="right">${firstName}</th>
      <td> ... </td>
   </tr>
   ...
</body>
```

The corresponding entry in the bundle file (for example, `messages.properties`) is:

```
firstName=First Name:
lastName=Last Name:
loanType=Loan Type:
```

Process Sever links the HTML form with a bundle file by using a HTML `<link rel="i18n">` element in the HTML head. Therefore, the bundle file must be deployed to Process Server at the time the HTML form is

generated. Additional language bundles can be saved along side of the default bundle and deployed to the server; for example `messages.properties` (default), `messages_fr.properties`, and `messages_fr_CA.properties`. In this case (with multiple bundles), only the default bundle `messages.properties` (one without the language or country) needs to be specified in the `<link>` element.

At runtime, Process Central downloads the required bundle (based on the bundle referenced in the `<link rel="i18n">` element), and applies the translation before returning the form's content to the web browser. The requested bundle resource depends on the browser's `Accept-Language` header information as configured by the user. A URL parameter can also be used to force the use of a specific language bundle. For example to log in and use Process Central using the French (fr) language bundle, append `?lang=fr` to a Process Central URL (for example, `http://localhost:8080/activevos-central/avc/avc.jsp?lang=fr`)

Request Form Attachments

In order to support attachments in process request forms, the following changes must be made to the form.

1. Find the input `<form>` tag and change it to specify `POST` as the method and `multipart/form-data` as the enctype.

   Change the following:

   ```
   <form id="DocumentInputForm$ID" name="DocumentInputForm$ID" action="">
   ```

   to use `POST` method and `multipart/form-data` enctype:

   ```
   <form id="DocumentInputForm$ID" name="DocumentInputForm$ID"
            action="" method="post" enctype="multipart/form-data">
   ```

   Note: This example shows form ID `DocumentInputForm$ID`. You should not change the form ID in your HTML file.

2. On the next line directly below the updated `<form>` tag, add the following hidden input fields:

   ```
   <input type="hidden" name="_json" value="" class="_json$ID" />
   <input type="hidden" name="responseContentType" value="text/html" />
   <input type="hidden" name="responseWrap" value="textarea" />
   ```

3. Add the HTML control for file upload within the HTML form. Here is an example:

   ```
   <input id="Document_attachment_1$ID" name="Document_attachment_1"
             value="" size="50" type="file" />
   ```

4. Move the submit button which currently is located outside of the `<form>` tags into the form so that it is before `</form>`.

5. Change the type of the submit button from `type='button'` to `type='submit'`.

   ```
   <input type="submit" id="sendRequest$ID" value="Send Request"
         class="avc_form_instance_save" />
   ```

6. Delete the following two lines of JavaScript from the `documentReady` function:

   ```
   // Assign handler for saving form data entered by the user
   $("#sendRequest$ID").click(_submitRequest);
   ```

7. Add the following JavaScript to the `documentReady` function:

   ```
   $('#DocumentInputForm$ID').submit(function() {
      _submitRequest();
   ```

```
    // Important always return false to prevent standard browser submit
    // and page navigation
    return false;
});
```

8. Replace the following JavaScript in the `_submitRequest()` function:

```
AE_AJAX_SERVICE_UTIL.postJSON(url, request, _showResponseCallback,
            _showFaultCallback, _communicationErrorCallback);
```

with:

```
var jsonString = JSON.stringify(request);
$("._json$ID").val(jsonString);
var options = {
   dataType : 'json',
   url: url,
   success: function(aJsonResponse) {
      alert("Success");
      $("#sendRequest$ID").attr("disabled", false);
   },
   error: function(aXMLHttpRequest, aTextStatus, aErrorThrown) {
      alert("ERROR");
      $("#sendRequest$ID").attr("disabled", false);
   }
};
$("#sayHelloInputPartInputForm$ID").ajaxSubmit(options);
```

If you use this methodology (with the jQuery Forms plugin), the `success` callback function is used for success and for faults. Your code in the `success` callback function should examine the JSON result to determine if the response is a fault.

Tasks

Task forms are similar to Requests where the HTML contains an input section and an output section driven by scripts. Task forms differ from Requests as follows:

- Task form input is rendered as read-only (WSHT Input) and the form response area contains editable HTML controls (WSHT Output).
- Task forms interact with the WSHT API (instead of directly invoking processes like the Request forms). In Requests, submitting a form consists of POSTing JSON to the appropriate endpoint. In Tasks, 'Saving' a form is equivalent to calling the WSHT API `setOutput()` operation.
- Task forms are driven by a Process Central task manager JavaScript framework. This framework handles common work such as task operations (claim, save) and managing peripheral data associated with a task such as comments and attachments.

The JavaScript object that models a task form is:

```
// Function that encapsulates the task form script.
var AeTaskForm$ID = function() {
   // the task name
   var mTaskName = "ApproveLoan";
   // task id
   var mTaskId = null;
   // task instance (AeTask instance)
   var mTask = null;

   /**
    * Called by the manager to initialize the task form.
    */
   this.avcform_initialize = function(aTaskId, aTaskFormContext) {
```

```
      mTaskId = aTaskId;
      mTaskFormContext = aTaskFormContext;

      // (equivalent to Request's documentReady() fn )
      //
      // Initialize your code here.  You can populate
      // some UI fields, but not the task UI fields.
      // (see avcform_onTaskReady())
      //
      // Set up form validation if needed
   }

   /**
    * Called when the task instance has been loaded from the server.
    */
   this.avcform_onTaskReady = function(aTask) {
      // Called one time when the task instance (AeTask object)
      // is first loaded (via WSHT API). Use this to initialize mTask
      // and display input and outdata from the task.
      mTask = aTask;
      this.showData(); // display task data
   }

   /**
    * Called when the task instance has been updated from the server.
    */
   this.avcform_onTaskUpdated = function(aTask) {
      // called each time the task has changed. For example, task
      // was claimed (state went from READY state to IN_PROGRESS).
      mTask = aTask;
   }
   /**
    * Called when loading task instance from the server failed.
    */
   this.avcform_onTaskLoadError = function(aMessage) {
       // return false to allow AVC system to handle error.
       // or return true to indicate that your code handled it.
       return false;
   }

   /**
    * Called to enable/disable form input fields
    * @param aEnable Indicates if the form should be enabled or disabled
    */
   this.avcform_enable = function(aEnable) {
       // return false to allow AVC system to handle enable/disabling of form fields
       // you can enable/disable your custom controls here.
       // normally a form is enabled when its editable (Claimed)
       return false;
   }

   /**
    * Called to validate the task form data before the task is saved
    */
   this.avcform_validate = function() {
      // Called when the form needs to be validated before saving it (WSHT setOutput
operation).
      // The form is not saved (setOutput not called) if this method returns false.
      // return true for relaxed validation on saving a form that isn't 100% complete
      return true;
   }
   /**
    * Called to validate the task form data before the task is completed
    */
   this.avcform_validateForCompletion = function() {
      // Validation function called before completing a task.
      // Return true if the form is valid or false to prevent WSHT complete()
operation.
      return true;
   }
```

```
   /**
    * Called when the task needs to be saved.
    */
   this.avcform_save = function(aCompleteTask) {
      // This function is called when the user presses the "Save" button.
      // Code save the form (WSHT setOuput) goes here. (normally generated for you).
   }

   /**
    * Binds data to the UI controls.
    */
   this.showData = function() {
      // this method is called when the task is loaded. e.g. via avcform_onTaskReady()
above.
      if(mTask != null) {
          // show task data in UI
      }
   }
}
```

Note that at the end of the script, you will see a function similar to:

```
$(document).ready(function() {
    ACTIVEVOS_TASKFORM_CONTEXT.init(_TASK_INSTANCE_ID_, new AeTaskForm$ID() );
});
```

The `ACTIVEVOS_TASKFORM_CONTEXT` is a JavaScript object that is made available at runtime by the framework. The `_TASK_INSTANCE_ID_` is replaced at runtime with the string value of the task instance ID. For example:

```
// runtime
$(document).ready(function() {
    ACTIVEVOS_TASKFORM_CONTEXT.init("urn:b4p:123", new AeTaskForm_123() );
});
```

The `ACTIVEVOS_TASKFORM_CONTEXT.init(...)` function basically 'registers' the task form instance with the framework. The framework will then call back appropriate functions on the task form instance; for example, `avcform_initialize()`, `avcform_onTaskReady()` and `avcform_onTaskUpdated()`. Specifically, the task form JavaScript object interacts with the Central (framework) in the following way:

1. The `avcform_initialize(aTaskId, aTaskFormContext)` function is called when the form in loaded. Central will then get the task data using the Human Task API.
2. Once the task data is available, Central will call the `avcform_onTaskReady(aTask)` function on the form object passing in the task instance.
   The form UI is updated at this time to reflect the task's input and output values.
3. Process Central's framework handles user interaction on the tasks operation toolbar (such as Claim, Save). When the task state has changes due to a user interaction (for example, a user claimed a task), the framework invokes the WSHT API and later calls the `avcform_onTaskUpdated(aTask)` function.

   The `avcform_enable(bool)` function is also called to enable (when a task is claimed) or disable the form UI.
4. When the Save button is pressed, the framework invokes the task form object's `avcform_validate()` function to validate the UI data, followed by invoking the `avcform_save()` function. The code in `avcform_save()` saves the form by invoking the WSHT `setOutput()` operation.

Embedding Request Forms in Standalone Web Pages

Process Server exposes all its processes and services using XML and JavaScript Object Notation (JSON) bindings in addition to SOAP, REST and JMS. The XML (and JSON) binding makes it easy for application

developers already familiar with XML/JSON-based REST application development to invoke processes and obtain responses from them.

Using XML or JSON bindings frees developers from having to obtain and learn SOAP libraries to build applications that leverage Process Server service-based processes. This approach allows JavaScript developers to use various libraries such as jQuery to build process-enabled applications.

Process Server uses the JSON binding for Process Central request and task forms. This implementation provides functions that developers can use for process-enabled application projects. Examples of the use of these functions are provided in this section of the help.