AVATO

Avato Developers Guide

2020-10-30

Table of Contents

Elided…

Part I. Build Command Reference

Overview

Avato comprises an executable providing a set of core components used in assembling transformation circuits and a set of specialized packages providing interfaces to external servers and clients. These two functionalities — internal Avato components and external interfaces — are distinctly separated. Every Avato installation is likely to use the same set of core components but a set of significantly different interfaces. In fact, the only time an Avato installation might use a non-standard set of core components is when an external server or client requires the development of specialized Java code.

This separation of concerns allows Avato to maintain a stable core yet at the same time provide dynamic functionality that can be changed on-the-fly without requiring recompilation or server reboot. This significantly eases development of new external interfaces and data transformations while radically reducing server downtime costs and recompilation times.

In support of this two-part design, Avato includes a flexible, powerful build system for configuring, compiling, and deploying instances of the server application for local development and for deployment to QA and production servers.

This reference and the process of using the Avato build system is divided into three parts:

The Build Command

The reference manpage for the Ant build command. Please familiarize yourself with the build command before proceeding.

Building and Deploying the Avato Core Server

Building, packing and distributing the Avato core itself, which may also have packages included as part of the core distribution. This is the process used when first standing-up an Avato system.

Building and Deploying Avato

Building, packaging, and deploying of custom Avato packages that leverage a specific pre-compiled version of the Avato core. This is the process most developers will be using as they configure stand-alone clusters of Avato instances for your own business units and environments.

Please see *Appendix C, Document Conventions* for useful information regarding the typography used in this document.

1 The Build Command

Name

ant - a Java based make tool.

Synopsis

ant *task*... [*-Dproperty="value"*]...

Description

ant is a tool to build software. It uses the file `build.xml` to define *tasks* and *properties* for the build process. For Avato, it is used to update, validate, build, and deploy Avato instances.

Tasks

In normal usage it is common to specify multiple tasks to be executed in sequence.

Tasks marked *(core)* are available only to core developers; tasks marked *(system)* are available only to system developers.

build-core *(core)*
: Builds the core Avato web application runtime. Re-compiles core components from source code and creates new bare-bones `.ear` and `.war` prototypes. Once compiled, Avato core never needs to be recompiled except when the core source code is updated.

build-core *(system)*
: Checks that the core Avato runtime is available. If it is missing, calls the system **update** task and prepares the filesystem.

build
: Calls the **build-core** *(system)* and **build-packages** tasks, then generates and validates the server configuration files. Typically used in combination with the **deploy** or **pac** task.

build-packages *(system)*
: Build the packages that have been specified in the environment's `packages.txt` file. Performs dependency resolution and de-tokenization of properties, assembles build resources, and then copies files to the deployment directory.

mkdocs
Builds documentation (PDF and HTML) from Asciidoctor source.

docs
Include documentation in the deployment.

deploy
Merges the core prototypes with packages, creating a full Avato application, and deploys Avato to a local server instance as a `.war` file.

deploy-tmp
Deploys an Avato instance to the local server as an uncompressed directory folder enabling file updates without restarting the server. See also the **deploy-unzipped** task below.

pac *(core)*
Packages the Avato core prototypes, Avato packages, and supporting configuration and build files into a single directory, and a set of zipped files, any of which can then be copied to a server for deployment.

pac *(system)*
Calls the **zip** task.

zip *(system)*
Zips local packages, properties and `build.xml` for distribution. The zip file can be extracted on a remote machine, where the `ant build` task will fetch the Avato core from the online distribution center and build an installer for execution on that remote machine. The output file is produced in the root directory and is named `mobius.install.zip`.

update *(system)*
Downloads the latest Avato bootstrap from the distribution servers configured in `properties/`*`env-name`*`/properties.xml` and, if the download is successful, calls the **update-all** task.

update-all *(system)*
Downloads the latest Avato core, packages, and properties from the distribution servers configured in `properties/`*`env-name`*`/properties.xml` and, if the download is successful, replaces the existing core, packages, and properties folders.

update-and-deploy *(system)*
Called by the server to perform the **update**, **build** and **deploy** tasks. This updates the bootstrap and then executes a fresh build and deploy.

validate-web *(system)*
Validates the deployment file `web.xml`. This is an “expensive” operation, taking tens of seconds to run to completion, as it fetches JBoss schemas from the web.

Elided…

Part II. Properties Configuration

Overview

Development, test, and production environments usually do not share the same resources. Additionally, correct configuration of resources may require values to conform to certain specifications: for example, a URL that is correctly formed, an integer within a certain range, or the selection of one of a limited number of valid values.

Avato provides an elegant solution to these challenges:

- Property files that provide environment-specific and package-specific settings;
- A build system that is aware of environments and packages;
- A tokenization system that allows you to use generic property names in place of their values;
- Schematization of properties, so that properties and configurations can be validated prior to building the system.

Development and configuration of the deployment environment and packages takes place in the *root*`/properties/` folder. These are illustrated in the next two diagrams:

Avato Properties Directory Structure

```
Avato
└── properties ❶
    ├── local ❷
    |   ├── Analytics.Google.properties.xml ❸
    |   ├── Development.JIRA.properties.xml
    |   ├── Path.To.Package.Properties.xml
    |   └── properties.xml ❹
    └── production ❺
        ├── Analytics.Google.properties.xml ❻
        ├── Development.JIRA.properties.xml
        ├── Path.To.Package.Properties.xml
        └── properties.xml ❼
```

❶ Properties configuration root directory.

❷ ❺ Environment folders: “local” for the developer running on a local workstation and “production” for a deployment to a server cluster. When building Avato you may use the *env-name* property to create and populate environment folders.

❸ Property files specific to each package, each configured for the local developer environment.

❹ Global properties for the developer environment.

❻ Property files specific to each package, each configured for the production environment.

❼ Global properties for the production environment.

Avato Packages Directory Structure

```
Avato
└── packages ❶
    ├── Analytics ❷
    |   └── Google ❸
    |       ├── package.xml ❹
    |       ├── properties.xml ❺
    |       ├── properties.xsd ❻
    |       └── subfolders ❼
    └── Development ❽
        └── JIRA ❾
            ├── package.xml ❿
            ├── properties.xml ⓫
            ├── properties.xsd ⓬
            └── subfolders ⓭
```

❶ Packages root directory.

❷ ❽ Package groups: packages may be arbitrarily grouped; not shown here are other packages providing analytics or development support.

❸ ❾ Package folders: each package is self-contained within a folder.

❹ ❿ Package Configuration: each package must have a configuration file that defines the interfaces, components, connections, data sources, and other facets required by the build system to install a package on the server.

❺ ⓫ Package Property Template: each package may use tokens in its configuration file; this property template provides definitions for each token. This file is copied to *root*`/properties/` environment folders the first time the package is added to an environment, as shown in the Avato *Properties Directory Structure* example above.

❻ ⓬ Package Schema: an optional schema will be used to validate the package's *root*`/properties/` property files.

❼ ⓭ Package functionality is usually distributed among various folders (e.g. `catalog`, `xsd`, `src`, `xslt`, etc.)

The following chapters examine the Properties system in detail.

4 About Property Files

Property files provide the mapping through which uniquely-identified tokens, in the form `@token.name@`, are replaced by a value during the build process. The value selected for the token is selected by considering the target environment and the file that is being processed.

In most cases the processed file is a package configuration file found somewhere in the *root*`/properties/` directory tree. During the build process tokens in a package's own `properties.xml` file are preferred to those in the environment's `properties.xml` file.

There are other build configuration files involved in building and deploying Avato and some of these also use token replacement. In those cases, the build system looks only in the appropriate environment `properties.xml` file.

The structure of a properties file also defines the name of a token. To better understand this, let us examine a properties file in more detail.

4.1. Structure of a Properties File and Token Naming

Structure of a Properties File

Avato XML Properties files have an arbitrary root wrapper element with descendent simple elements containing *either* simple elements (containing only simple elements) *or* text content:

```
<root-element
  xmlns="namespace_uri">
  <!-- Content: (element)* -->
</root-element>
```

```
<element>
  <!-- Content: (element | string)? -->
</element>
```

In practice, a Properties file might look something like this:

```
<properties
  xmlns="http://worldwest.local/analytics/Google/properties/1.0">
  <google>
    <analytics>
      <key>0df49fc6-c4b3-4a5d-836d-138a31dc524b</key>
    </analytics>
```

```
  </google>
</properties>
```

This property file generates a token named `@google.analytics.key@`. The convention used to generate token names is described below:

Token Naming

Tokens are wrapped with `@` (at) signs and their name is formed by dot-separating the element path to the token value after discarding the arbitrary root wrapper element. In other words, token names look like `@path.to.token.value@`.

This rule can be illustrated using the following simple example containing three tokens:

```
<properties
  xmlns="http://my.org/package/path/properties/1.0">
  <PKGNAME>
    <A>1</A>
    <B>2</B>
    <C>
      <D>
        <E>3</E>
      </D>
    </C>
  </PKGNAME>
</Package>
```

Applying the simple dot-path rule, we get these name-value pairs:

Token Name	Value
@PKGNAME.A@	1
@PKGNAME.B@	2
@PKGNAME.C.D.E@	3

Note

The tokens defined in a package's `properties.xml` file are used only for that package. Other packages can not access those tokens.

When creating a package properties file there is no need to worry about name collisions, with one notable exception: Packages can redefine token values found in the environment's `properties.xml` file. The redefined value is applicable to that package only: no other package will be affected, nor will the private system files accessed by the build process. However, such a situation has high potential for creating confusion and should be avoided.

Important

It is best practice to create token names that use the package name as the first name element, thus avoiding any token name confusion.

Token Replacement Example

The addition of highlighting to an example can help illustrate the token replacement process. Given this properties file:

```
<properties
  xmlns="http://worldwest.local/analytics/Google/properties/1.0">
  <google>
    <analytics>
      <key>0df49fc6-c4b3-4a5d-836d-138a31dc524b</key>
    </analytics>
  </google>
</properties>
```

And this package file:

```
<Package
  xmlns="http://worldwest.local/package/1.0"
  xmlns:int="http://worldwest.local/interface/1.0" >
  <Name>Google Analytics</Name>
  <Description>Adds support for Google Analytics, including embedding the tag in
 each JSP page.</Description>
  <Component>
    <Resources type="web">web</Resources>
  </Component>
  <Platform>
    <int:ContextParameter name="google.analytics.key">
      @google.analytics.key@
    </int:ContextParameter>
  </Platform>
</Package>
```

The build system will derive this final package file:

```
<Package
  xmlns="http://worldwest.local/package/1.0"
  xmlns:int="http://worldwest.local/interface/1.0" >
  <Name>Google Analytics</Name>
  <Description>Adds support for Google Analytics, including embedding the tag in
 each JSP page.</Description>
  <Component>
    <Resources type="web">web</Resources>
  </Component>
  <Platform>
```

Elided…

Part III. Interface Developer Guide

Elided…

8.5. Dispatcher

The *Dispatcher* transforms an incoming request into multiple outgoing upstream service requests and then processes the responses to return a single response to the downstream service caller.

The Dispatcher is useful for implementing Model-View-Controller (MVC) patterns, for aggregating or combining data from different systems, and for implementing conditional flow dependent on the requests and responses incurred during the fulfillment of an individual service request.

8.5.1. Configuration Example

The following configures a component that is listening on the `/po/dispatcher` service endpoint. The messages it receives are transformed by `ProcessPurchaseOrder.xsl` which will, based on the message content, send message requests to other services and process their responses to generate a response message.

```
<Dispatcher
  name="purchaseorder-dispatcher"
  wsdl="/msg/po/dispatcher?wsdl"
  xformRequest="/msg/purchaseorder/xsl/ProcessPurchaseOrder.xsl" >
  <DisplayName>Purchase Order Dispatcher</DisplayName>
  <Description>Dispatches POs to the database and forwards select POs for
 further processing</Description>
  <Listening>
    <Address>/po/dispatcher</Address>
  </Listening>
</Dispatcher>
```

8.5.2. Type Information

8.5.2.1. Namespace

`http://worldwest.local/interface/1.0`
`<Dispatcher/>` belongs to the Interface namespace.

8.5.2.2. Type Definition

`<Dispatcher/>` is defined by an XSD 1.0 type named `int:DispatcherType`.

8.5.2.3. Class

The Dispatcher component is implemented by the `ca.worldwest.mobius.web.servlet.Dispatcher2` class.

8.5.24. Inherited Components

The `<Dispatcher/>` element inherits elements and attributes from

- *int:ComponentType*.
- *int:Forwarder*.
- *int:Transformer*.

8.5.2.5. Derived Components

No component type definitions extend `int:DispatcherType`.

8.5.3. Simple Schema Model

```
<Dispatcher
[inherited from Transformer]
  [xformRequest? = server url]
  [parse-request? = boolean]
  [xformResponse? = server url]
  [parse-response? = boolean]
  [requestContextXPath? = XPathExpression]
[inherited from Forwarder]
  [upstreamServiceName? = string]
  [httpMethod? = string]
  [connectTimeout? = integer]
  [readTimeout? = integer]
  [eraseCookies? = boolean]
  [append-uri-tail? = boolean]
  [append-uri-params? = boolean]
  [append-root? = string]
  [ssl-ignore-host? = boolean]
  [allow-direct? = boolean]
[inherited from ComponentType]
  [name = string]
  [class? = string]
  [loginTokenXPath? = string]
  [contentType? = string]
  [faultHttpResponseCode? = string]
  [xformFault? = server url]
  [wsdl? = server url]
  [xformWSDL? = server url]
  [urlErrorOutputPath? = server url]
  [log-level? = "TRACE" | "DEBUG" | "INFO" | "WARN" | "ERROR" | "FATAL"]
  [recordingDisabled? = boolean]
  [writeToSequence? = string]
  [writeToSequenceName? = string]
  [xformRecordTitleRequest? = string]
```

```
  [xformRecordTitleResponse? = string]
 >
  <!-- Content: ()
[inherited from Transformer]
                ([Expect?])
[inherited from Forwarder]
                ([Headers?] [Forwarding?])
[inherited from ComponentType]
                ([DisplayName?] [Description?] [Listening] [Parameters?])
  -->
</Dispatcher>
```

```
<Headers
  mode = "passthrough", string
  for? = tokens >
  <!-- Content: (Add | Delete | Modify)* -->
</Headers>
```

8.54. XML Model

```
<Dispatcher
  name="Component Name" ❶
  [inherited attributes from Transformer]
  [inherited attributes from Forwarder]
  [inherited attributes from ComponentType] >
  [inherited elements from Transformer]
  [inherited elements from Forwarder]
    <Headers for="tokens"/> ❷
  [inherited elements from ComponentType]
</Dispatcher>
```

❶ **`name`**

The name of the component. It must be unique within the system.

❷ `Headers`, `for`

The `<Headers/>` element inherited from *Forwarder* is extended with the attribute `for`, used to identify groups of header modifications.

If `for` is not set, the `name` attribute of a dispatched `<esb:XmlRequest/>` will be used to match against the `name` attribute of `<Add/>`, `<Modify/>`, and `<Delete/>` header modifications.

Otherwise, `for` may contain a space-separated list of tokens, in which case the `header-group` attribute of dispatched `<esb:XmlRequest/>`'s will be used to match `<Add/>`, `<Modify/>`, and `<Delete/>` header modifications.

This is particularly useful when a *Dispatcher* is collating information from a number of services that require the same header modification, but where the `name` of each `<esb:XmlRequest>` needs to be unique so that the various responses can be used as data sources.

Otherwise, *Dispatcher* configuration is identical to *Transformer* configuration.

8.5.5. Using Dispatcher's esb:Service Elements

The *Dispatcher* extends *Transformer* by defining a set of uniquely namespaced elements that may be emitted by a transformation. When Dispatcher detects one of these elements, it performs a request to a service endpoint and then feeds that endpoint's response back into the transformation. A record of all requests and responses is maintained, allowing the transformation to return a response comprised of data obtained from multiple sources.

These *Dispatcher*-specific elements are in the `http://worldwest.local/esb` namespace. The recommended namespace prefix is `esb`. See the XML models *Dispatcher Services Simple Schema Model* and *Dispatcher Services XML Model*, below, for markup details.

8.5.5.1. Dispatch esb:Services Simple Schema Model

```
<Services
  xmlns="http://worldwest.local/esb">
  <!-- Content: (XmlRequest? HttpRequest?) -->
</Services>
```

```
<XmlRequest
  xmlns="http://worldwest.local/esb"
  uri = url
  name? = string
  faultaction? = "terminate" | "ignore" | "dispatch"
  response-as? = "content" | "name"
  content-type? = "xml/text" | string
  http-method? = "GET" | "HEAD" | "POST" | "PUT" | "DELETE" | "CONNECT" |
 "OPTIONS" | "TRACE" | "PATCH"
  context? = string
  header-group? = string
  >
  <!-- Content: (any) -->
</XmlRequest>
```

```
<HttpRequest
  xmlns="http://worldwest.local/esb"
  URL = url
```

```
  name = string
  faultaction? = "terminate" | "ignore" | "dispatch"
  response-as? = "name" | "content"
  content-type? = "xml/text" | string
  http-method? = "GET" | "HEAD" | "POST" | "PUT" | "DELETE" | "CONNECT" |
 "OPTIONS" | "TRACE" | "PATCH"
  context? = string
  header-group? = string
  >
  <!-- Content: (HttpHeaders? HttpBody) -->
</HttpRequest>
```

```
<HttpHeaders
  xmlns="http://worldwest.local/esb"
  >
  <!-- Content: (HttpHeader*) -->
</HttpRequest>
```

```
<HttpHeader
  xmlns="http://worldwest.local/esb"
  name = string
  >
  <!-- Content: (text) -->
</HttpRequest>
```

```
<HttpBody
  xmlns="http://worldwest.local/esb"
  >
  <!-- Content: (text) -->
</HttpRequest>
```

8.5.5.2. Dispatcher esb:Services XML Model

```
<Services
  xmlns="http://worldwest.local/esb" >
  <XmlRequest
    uri="Request Endpoint" ❶
    name="Response Name" ❷
    faultaction="terminate|ignore|dispatch" ❸
    response-as="name|content" ❹
    content-type="text/xml|other content-type" ❺
    http-method="GET|HEAD|POST|PUT|DELETE|CONNECT|OPTIONS|TRACE|PATCH" ❻
    context="???" ❼
    header-group="???" ❽
    >
    request message ❾
```

```
  </XmlRequest>
  ⋮
  <HttpRequest
    uri="Request Endpoint" ❿
    name="Response Name" ⓫
    faultaction="terminate|ignore|dispatch" ⓬
    response-as="name|content" ⓭
    content-type="text/xml|other content-type" ⓮
    http-method="GET|HEAD|POST|PUT|DELETE|CONNECT|OPTIONS|TRACE|PATCH" ⓯
    context="???" ⓰
    header-group="???" ⓱
    >
    <HttpHeaders>
      <HttpHeader
        name="header name"> ⓲
        header value
      </HttpHeader>
      ⋮
    </HttpHeaders>
    <HttpBody> ⓳
      body content
    </HttpBody>
  </XmlRequest>
  ⋮

</Services>
```

❶❿ **`uri`**

A URL identifying the endpoint destination for this request.

❷⓫ `name`

A name for the response content, allowing easy template matching when processing the response. See `response-as`, below, for further details.

❸⓬ `faultaction`

Identifies the action taken if the endpoint returns a fault. `terminate` will immediately pass the fault upstream, ending all processing of the transformation. `ignore` will continue processing the transformation; the fault will be available in the dispatch results document. `dispatch` is not currently supported.

❹⓭ `response-as`

By default, the endpoint response will be fed back into the transformation as the default context; it can be identified using a match against its elements, i.e. `/*:Envelope/*:Body/`*`response root element name`*. Set to `name`, the endpoint response will be returned in a document named

after the request; it can be matched as `/*[name()='name']` or retrieved as `doc('name')`. It is normally preferable to use named responses.

❺ ⓮ `content-type`

Assigns a content type to the request. The default content-type is `text/xml`.

❻ ⓯ `http-method`

Configures the HTTP method for the request. The default method is `GET`.

❼ ⓰ `context`

Where `name` is used to join a dispatched request and response in the `DispatchResult` document, `context` is used to pass information to the transformation for use during response matching and transformation. Usually, this information will be computed during the dispatch request phase, rather than information contained in the request itself (which can always be accessed using `doc('name')`.

❽ ⓱ `header-group`

Identifies a header action (Add, Delete, Modify) defined in the Dispatcher component definition. See the *Interface Developer Guide* section on *Dispatcher* for details.

⓲ `<esb:HttpHeader>`

Defines an HTTP request header with a given name and value.

⓳ `<esb:HttpBody>`

Defines the HTTP request body. Note that the content of the `<esb:HttpBody/>` element is processed as XSL and subject to its rules for whitespace handling, entity encoding, and disallowed characters. Use a CDATA section to pass raw text that would otherwise be modified by the XSL processor.

❾ request message

The child content of the `<esb:XmlRequest>` extension is comprised of arbitrary XML content generated through XSL instructions or hard-coded text. Generally, calls to SOAP wrapper or other XML message construction functions will be performed, passing various information extracted from the current context or through other calculations. The end result will be an XML message suitable for consumption by the service endpoint.

8.5.6. Processing of Dispatcher's esb:Service Elements

The `esb:Service` elements are used in XSL and XQuery transformations to perform asynchronous service requests.

In an typical transformation, handled by the *Transformer* component, the result document is comprised of information gleaned exclusively from the document(s) passed into the transformation, and is built on-the-fly as the source document is consumed. There is little support in standard XSLT or XQuery to query external sources: the `doc()` function has limited access to file and http resources, and that access is synchronous: processing halts until the resource is obtained. These limitations are often acceptable: a lot of data transformation work simply converts a message containing one data structure to a message containing a different but parallel data structure, enabling communication between proprietary software products.

For more complex scenarios, where a response must consolidate information from multiple sources, the *Dispatcher* component provides support for asynchronous, non-blocking querying of endpoints. In this processing model, the transformation is run multiple times: once for the source document(s), again for each response to a service request, and a final run against the collated request-response collection.

On the initial run, template matches are performed against the source document (the initial request message), just like a normal transformation. Unlike a normal transformation, though, the final result document is not built on-the-fly as the source document is consumed. Rather, an intermediate `<esb:Services/>` document is generated, comprising a set of requests to collect additional documents and data. Only when these requests have been fulfilled (or timed out) is the final result document composed, using information gleaned from the set of request-response documents.

When a transformation emits `<esb:Services/>` documents, Dispatcher:

A. Sends out the requests and waits for a response or time-out;

B. Stashes the request-response pair in a collection document; and

C. Sends the responses back into the transformation. A response may be matched by a template in order to extract information for a subsequent request; if it is not matched, the transformation returns nothing.

This intermediate stage continues as long as the Dispatcher receives `<esb:Services/>` documents or responses to requests (and does not receive a fault that requires termination). When all requests have been fulfilled or timed-out, it can procede to the final stage of processing.

In the final stage the collection document, with a root node named `DispatchResult`, is sent into the transformation. There it will be matched by a template, which will in turn generate the final result document by composing an XML document using information sourced from the collection of requests and responses.

This final result document is the response to the initial request, and is returned upstream by the Dispatcher.

8.5.7. Example XSL Use of Dispatcher's esb:Service Elements

The following (simplified) XML and XSLT presents a usage pattern for Dispatcher. Unnecessary detail has been eliminated: in real-world use there would be need for namespace definitions, fault handling, and so on.

In this example, the Dispatcher receives a SOAP message:

```
<s12:Envelope>
  <s12:Body>
    <GetSalesReport>
      <Employee id="1000"/>
      <Employee id="1001"/>
      <Employee id="1002"/>
    </GetSalesReport>
  </s12:Body>
</s12:Envelope>
```

From this, a report listing employees by name and sales is to be generated. The initiating request message lacks the required information: we must use Dispatcher to query service endpoints that can return employee names and sales information. In our scenario these services are provided by separate endpoints; bizarrely, but conveniently for our demonstration, the sales information is indexed by employee name, not their ID!

First Stage: handle the initiating request

To begin, a template must match our initiating request. It will generate a set of requests for employee names, so that we may subsequently query the sales information service. The names-providing service communicates using SOAP messaging.

```
<xsl:template match="/s12:Envelope/s12:Body/GetSalesReport">❶
  <esb:Services>
    <xsl:for-each select="Employee">❷
      <esb:XmlRequest name="GetEmployeeName-{@id}" uri="http://names-endpoint">
        <s12:Envelope>
          <s12:Body>
            <getNameFromId><ID>{@id}</ID></getNameFromId>
          </s12:Body>
        </s12:Envelope>
      </esb:XmlRequest>
    </xsl:for-each>
  </esb:Services>
</xsl:template>
```

❶ Match the initiating request. For safety, a less-specific template match might capture unrecognized SOAP requests and return a fault indicating that this service can not handle the request.

❷ Iterate through employees, creating a request for each one.

The above template consumes the source document, emitting the following XML content on completion:

```
<esb:Services>
  <esb:XmlRequest name="GetEmployeeName-1000" uri="http://names-endpoint">
    <s12:Envelope>
      <s12:Body>
        <getNameFromId><ID>1000</ID></getNameFromId>
      </s12:Body>
    </s12:Envelope>
  </esb:XmlRequest>
  <esb:XmlRequest name="GetEmployeeName-1001" uri="http://names-endpoint">
    <s12:Envelope>
      <s12:Body>
        <getNameFromId><ID>1001</ID></getNameFromId>
      </s12:Body>
    </s12:Envelope>
  </esb:XmlRequest>
  <esb:XmlRequest name="GetEmployeeName-1002" uri="http://names-endpoint">
    <s12:Envelope>
      <s12:Body>
        <getNameFromId><ID>1002</ID></getNameFromId>
      </s12:Body>
    </s12:Envelope>
  </esb:XmlRequest>
</esb:Services>
```

Intermediate Stages: send out requests, transform responses

The Dispatcher takes the `<esb:Services/>` result document above and issues three separate SOAP requests to `http://names-endpoint`. The names endpoint returns responses in the form:

```
<s12:Envelope>
  <s12:Body>
    <NameFromId id="id">
      <givenname>first name</givenname>
      <surname>last name</surname>
    </NameFromId>
  </s12:Body>
</s12:Envelope>
```

The Dispatcher sends those responses back into the transform as they are received, where they will be matched by the following template. This time, we have to emit specially-crafted HTTP requests to the endpoint server:

```
<xsl:template match="/s12:Envelope/s12:Body/NameFromId">❶
```

```
  <esb:Services>
    <esb:HttpRequest name="GetEmployeeSales-{@id}" uri="http://sales-endpoint">
      <esb:HttpHeaders>
        <esb:HttpHeader
          name="PROP-SERVICE">GetEmployeeSales</esb:HttpHeader>
        <esb:HttpHeader
          name="Content-type">text/plain; charset=UTF-8</esb:HttpHeader>
        <esb:HttpHeader
          name="Content-length">{string-length(surname||','||givenname)}</
esb:HttpHeader>
      </esb:HttpHeaders>
      <esb:HttpBody>{surname},{givenname}</esb:HttpBody>
    </esb:XmlRequest>
  </esb:Services>
</xsl:template>
```

The above template emits a `<esb:Services/>` document that Dispatcher will convert to an HTTP request:

```
GET http://sales-endpoint HTTP/1.1
PROP-SERVICE: GetEmployeeSales
Content-type: text/plain; charset=UTF-8
Content-length: 28

surname,givenname
```

The request is sent to the sales endpoint, which returns an XML document:

```
<EmployeeSales>
  <Employee>surname,givenname</Employee>
  <Sales>dollar figure</Sales>
</EmployeeSales>
```

The sales endpoint response documents are themselves fed back into the transformation. Because we have no need to emit more `<esb:Services>` requests, there is no need to match against these responses. The Dispatcher, having emptied its `<esb:Services>` queue, can now send the request-response collection into the transformation for the final processing stage.

Final Stage: process the DispatchResult document

The initiating request, the various `<esb:XmlRequest>` and `<esb:HttpRequest>` requests, and their responses are collated into a single document with a root node named `DispatchResult`. This node must be matched by a template that will emit a response message for the upstream service:

```
<xsl:template match="DispatchResult">
  <s12:Envelope>
    <s12:Body>
```

```
      <SalesByEmployee>
        <xsl:for-each select="doc('Request')//Employee">❶
          <Employee>
            <Name>{doc('GetEmployeeName-'||@id)//givenname} 
{doc('GetEmployeeName-'||@id)//surname}</Name>❷
            <Sales>${doc('GetEmployeeSales-'||{@id})//Sales}</Sales>❸
          </Employee>
        </xsl:for-each>
      </SalesByEmployee>
    </s12:Body>
  </s12:Envelope>
</xsl:template>
```

❶❶ The `Request` document contains the initiating request, where we can iterate through the employee list.

❷❸ Using the employee ID, we can re-generate the names used for issuing the `<esb:Services/>` requests, using them to fetch the correct document and "drill down" to the required data value.

Alternative First Stage: handle the initiating request

As an alternative to using an imperative for-each loop, `<xsl:apply-templates/>` can be used to create a declarative pattern-matching solution to generate the initial `<esb:Services/>` requests:

```
<xsl:template match="/s12:Envelope/s12:Body/GetSalesReport">
  <esb:Services>❶
    <xsl:apply-templates/>
  </esb:Services>
</xsl:template>

<xsl:template match="Employee">
  <esb:XmlRequest name="GetEmployeeName-{@id}" uri="http://names-endpoint">
    <s12:Envelope>
      <s12:Body>
        <getNameFromId><ID>{@id}</ID></getNameFromId>
      </s12:Body>
    </s12:Envelope>
  </esb:XmlRequest>
</xsl:template>
```

❶ Note that `<esb:Services/>` wraps all the requests. A valid XML document must have only one root node: hence, the wrapper must be placed around the `<xsl:apply-templates/>` instruction.

Alternative Final Stage: process the DispatchResult document

As with the alternative first stage, `<xsl:apply-templates/>` provides a pattern-matching solution for handling the `DispatchResults`. Note the

use of XSL `mode` to prevent matching against the templates that generate `<esb:Services>` requests.

With this solution, employee names are not guaranteed to be listed in the same order as the initiating request.

```
<xsl:template match="DispatchResult">
  <s12:Envelope>
    <s12:Body>
      <SalesByEmployee>
      <xsl:apply-templates
        select="document(//Response/Document[starts-
with(@name,'GetEmployeeName-')]/@name)"
        mode="dispatchresult">
      </SalesByEmployee>
    </s12:Body>
  </s12:Envelope>
</xsl:template>

<xsl:template match="/s12:Envelope/s12:Body/NameFromId" mode="dispatchresult">
  <Employee>
    <Name>{givenname} {surname}</Name>
    <xsl:apply-templates
      select="document(//Response/Document[starts-
with(@name,'GetEmployeeSales-'||@id)]/@name)"
      mode="#current">
  </Employee>
</xsl:template>

<xsl:template match="EmployeeSales" mode="dispatchresult">
  <Sales>${Sales}</Sales>
</xsl:template>
```

Elided…

8.9. Logger

The *Logger* component logs information about the incoming request or response HTTP message headers and body. If the message is a request, it is then forwarded to its configured endpoint (or, if no endpoint is specified, returns the request to the caller); if the message is a response, it is forwarded back to the downstream service. Messages can be logged to a log file and/or to the Avato Sequence database.

The Logger component is especially useful for reverse engineering existing protocols, as it can be inserted into an existing communication channel between a client and its server without unexpected side-effects. The component in its default state passes on all information exactly as it was received to a configured upstream server, and returns the server's responses intact, without otherwise having any impact on the connection.

The Logger supports modification of the logged message. This can be used to obfuscate sensitive information, add content containing additional log information, or to reduce the log entry to a subset of the message information.

Note

A Message can be transformed only if it is well-formed XML with a MIME type starting with `text/xml`, `application/soap+xml`, or `application/xml`.

Warning

If a *Logger* component fails to process the message, due to invalid XML, inaccessible transformation, database failure, inability to write to the system log file, or any other reason, **the message content is written in plain text as received, unmodified** and the Circuit will continue to execute as planned.

For SOAP messaging, Avato supplies a *Logger* transformation `/xsl/auth/MaskLogMessages.xsl` that adds a variety of useful information about the component, server, and message as a SOAP Header nodes, and provides some basic masking matches.

The package `Privacy/Logging` provides a framework for developing custom log masking functionality. Models from which to adapt your own custom log masking can be found in the various `Banking/`*package*`/xsl/logging` directories.

When logging at `DEBUG` level, HTTP header fields are recorded.

8.9.1. Configuration Example

The following configures a component that is listening on the `/ui/activity/log` service endpoint. This component logs the request message before forwarding it to a Dispatcher.

```
<Logger
  name="activity-logger"
  content-log-level="WARN"
  log-level="DEBUG" >
  <DisplayName>Activity Logger</DisplayName>
  <Description>Logs requests to the user activity service</Description>
  <Listening>
    <Address>/ui/activity/log</Address>
  </Listening>
 <Forwarding>
    <Address>/ui/activity/dispatcher</Address>
  </Forwarding>
</Logger>
```

8.9.2. Type Information

8.9.2.1. Namespace

`http://worldwest.local/interface/1.0`
    `<Logger/>` belongs to the Interface namespace.

8.9.2.2. Type Definition

`<Logger/>` is defined by an XSD 1.0 type named `int:LoggerType`.

8.9.2.3. Class

The Logger component is implemented by the `ca.worldwest.mobius.web.servlet.Logger` class.

8.9.24. Inherited Components

The `<Logger/>` element inherits elements and attributes from

- *int:ComponentType*
- *int:Forwarder*

8.9.2.5. Derived Components

The following type definitions extend `int:LoggerType`

- *int:RecorderType*

8.9.3. Simple Schema Model

```
<Logger
  xformRequest? = server url
  xformResponse? = server url
  content-log-level? = "TRACE" | "DEBUG" | "INFO" | "WARN" | "ERROR" | "FATAL"
[inherited from Forwarder]
  [upstreamServiceName? = string]
  [httpMethod? = string]
  [connectTimeout? = integer]
  [readTimeout? = integer]
  [eraseCookies? = boolean]
  [append-uri-tail? = boolean]
  [append-uri-params? = boolean]
  [append-root? = string]
  [ssl-ignore-host? = boolean]
  [allow-direct? = boolean]
[inherited from ComponentType]
  [name = string]
  [class? = string]
  [loginTokenXPath? = string]
  [contentType? = string]
  [faultHttpResponseCode? = string]
  [xformFault? = server url]
  [wsdl? = server url]
  [xformWSDL? = server url]
  [urlErrorOutputPath? = server url]
  [log-level? = "TRACE" | "DEBUG" | "INFO" | "WARN" | "ERROR" | "FATAL"]
  [recordingDisabled? = boolean]
  [writeToSequence? = string]
  [writeToSequenceName? = string]
  [xformRecordTitleRequest? = string]
  [xformRecordTitleResponse? = string]
  >
  <!-- Content: ()
[inherited from Forwarder]
                ([Headers?] [Forwarding?])
[inherited from ComponentType]
                ([DisplayName?] [Description?] [Listening] [Parameters?])
  -->
</Logger>
```

8.94. XML Model

```
<Logger
```

```
  name="Component Name" ❶
  xformRequest="/server/path/to/transformation" ❷
  xformResponse="/server/path/to/tranformation" ❸
  content-log-level="TRACE|DEBUG|INFO|WARN|ERROR|FATAL" ❹
  [inherited attributes from Forwarder]
  [inherited attributes from ComponentType] >
  [inherited elements from Forwarder]
  [inherited elements from ComponentType]
</Logger>
```

❶ **`name`**

The name of the component. It must be unique within the system.

❷ ❸ `xformRequest`, `xformResponse`

The server path to transformation scripts to modify the logged content. See *Protecting Sensitive Information* for an example of its use.

❹ `content-log-level`

Configures the log priority for the message body.

This is distinguished from the component-level log level, which pertains to log messages generated by the component itself.

For more information about logging, please consult Apache's *Logging Services* documentation.

8.9.5. Logger Output

The default configuration for server logging writes `DEBUG` and higher-rated messages to both the terminal console and the application's `server.log` file. A typical example of logger output looks like this:

Typical Logger Output

```
09:49:08,346 INFO  [ca.worldwest.esb.application.recorder.ForwardServlet]
 (default task-26) Request headers =

SOAPAction=undefined
referer=https://worldwest.ca/esb/ui/circuits
accept-language=en-US,en;q=0.9
cookie=JSESSIONID=c3G2382cv6vYoDbPbkfY-kSukiTh-P0kHVIxQO4C
origin=https://worldwest.ca:8443
content-type=*
Host=worldwest.ca
accept=*/*
user-agent=Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36
```

```
09:49:11,644 INFO  [ca.worldwest.mobius.svn.subversion-logger.ServletBase]
 (default task-2) xformFault is https://worldwest.ca/esb/xsl/fault/
FaultResponse-1.2.xsl
09:49:12,070 INFO  [ca.worldwest.mobius.svn.subversion-logger.Logger] (default
 task-2) Processing subversion-logger request from 127.0.0.1
09:49:12,074 INFO  [ca.worldwest.mobius.svn.subversion-
logger.Logger.MessageContent] (default task-2) Body=

<s12Env:Envelope xmlns:s12Env="http://www.w3.org/2003/05/soap-envelope">
   <s12Env:Header>
      <Token xmlns="http://worldwest.local/auth">8daa1019-4c56-4c74-85d9-
c3206f96e2c2</Token>
   </s12Env:Header>
   <s12Env:Body>
      <History xmlns="http://worldwest.local/svn">
         <RepoURL xmlns="">http://worldwest.ca/repo/wonka</RepoURL>
         <From xmlns="">1</From>
         <To xmlns="">-1</To>
         <Username xmlns="">xxxx.xxxxxx</Username>
         <password xmlns="">xxxxxx</password>
      </History>
   </s12Env:Body>
</s12Env:Envelope>

09:49:12,075 INFO  [ca.worldwest.mobius.svn.subversion-logger.ForwardServlet]
 (default task-2) Processing subversion-logger request from 127.0.0.1 for /svn/
ws/service
09:49:12,075 INFO  [ca.worldwest.mobius.svn.subversion-logger.ForwardServlet]
 (default task-2) Forwarding to https://worldwest.ca/esb/svn/ws/service
```

8.9.6. Protecting Sensitive Information

Configure `xformRequest` and `xformResponse` to use a modified identity transformation to match against sensitive message element and attribute names, replacing their attribute and text content with obfuscated data.

Avato provides a logging library from which you may compose custom log filters. The following example is taken from a banking interface

```
<xsl:stylesheet
  exclude-result-prefixes="#all" version="2.0"
  xmlns:xd="http://www.oxygenxml.com/ns/doc/xsl"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import
    href="/xsl/logging/common.xsl" /> ❶
```

```
  <xd:doc><xd:desc><xd:p>
  The “block” variable contains a sequence of UPPERCASE element names containing
 data that is to be shown blocked in the log at all times.
  </xd:p></xd:desc></xd:doc>

  <xsl:variable as="xs:string*"
    name="block"
    select="'DATEOFBIRTH',
            'PASSWORD',
            'SIN',
            'SPOUSESIN'" /> ❷

  <xd:doc><xd:desc><xd:p>
  The “obscure” variable contains a sequence of UPPERCASE element names
 containing data that is to be shown obscured in the log at all times.
  </xd:p></xd:desc></xd:doc>

  <xsl:variable as="xs:string*"
    name="obscure"
    select="'BUSINESSCUSTOMERNUMBER',
            'CUSTOMERNUMBER',
            'MPMNO',
            'MPMNUMBER'" /> ❸

  <xd:doc><xd:desc><xd:p>
  The “show” variable contains a sequence of UPPERCASE element names containing
 data that is to be shown verbatim in the log at all times.
  </xd:p></xd:desc></xd:doc>

  <xsl:variable as="xs:string*"
    name="show"
    select="'TOKEN',
            'GUID',
            'COLUMNNAME',
            'OPERAND',
            'USERNAME'" /> ❹
</xsl:stylesheet>
```

❶ Import the log masking library.

❷ The “block” variable lists all elements that will their content replaced with `X` and `0` characters for letters and digits, when the log recording mode is configured for privacy mode.

❸ The “obscure” variable lists all elements that will have their content obscured — which is to say, blocked except for the last four characters — when the log recording mode is configured for privacy mode. In this application, “obscure” mode was used to reveal just enough information

to confirm that the correct information was being returned, without compromising customer privacy.

❹ The “show” variable lists all elements that will be shown verbatim in the log at all times, regardless the log recording mode.

8.9.7. Logging a Subset of Message Content

Configure `xformRequest` and `xformResponse` to extract a subset of the message content, emitting it as XML, plain text, or other format.

8.9.8. User Interface Notes

The server log file may be viewed on the **Navigation Menu → System → Logs** page. Unlike the console view, the web interface view is not colourized and is not updated in real time. See *Viewing the Server Log File* for details.

Logging data may be affected by transformations: in particular, privacy protection may be applied to obfuscate passwords, names, and other information. This protection is entirely dependent on configuration details: it is not automatic. See *Protecting Sensitive Information* for details.

Avato can import server log files, extracting logged messages into a sequence with their original (logged) timing characteristics. See *Load Log File* for details.

When adding a message to a sequence by hand, a logger name is requested. The name of the logger will be displayed in the sequence’s message list. It is useful to use the same *Name* and *Logger* values, so that you can identify the message in both the sequence message list and when viewing the message. See *Creating a New Message in a Sequence* for details.

When viewing a captured message exchange, the *Logged* column indicates the time the message was received, the *Logger* column lists the component that logged the message, *Request* and *Response* provide the name of the message’s root element, and *ms* lists the time it took to process and respond to the request. See *Viewing Message Details* for details.

The search command can be used to locate log entries from specific interfaces or components. See *Search Examples* for details.

When debugging a circuit, seeing the messages that have passed through a Logger is essential to understanding what is happening. Avato makes it incredibly easy to do this — select the “hamburger” menu at the top right of any component that is based on Logger and choose **Find messages**. Instantly, Avato opens the Sequencer Messages page with a full list of the messages that were logged by that component. See *Searching for a Message in a Sequence* for details.

Elided…

8.22. Transformer

The *Transformer* Component is used to convert XML requests and responses that pass through it from one XML dialect into another.

The *Transformer* supports *XSLT version 3.0*, *XQuery 3.1*, and *XPath 3.1*, the official World Wide Web Consortium (W3C) recommendations.

Like many integration Components, the Transformer is typically used in a circuit, where each component forwards its potentially modified input to an upstream service, then potentially modifies the response returned by the upstream service and returns that response to its caller. A *Transformer* Component configured with an upstream service address implies synchronous operation, as the component waits for the response from the upstream service before returning,

An optional *xformRequest* parameter defines an XSL or XQuery transform that is applied to the inbound request. If an upstream service has been configured, the transformed request is sent to the upstream service, otherwise it is returned back to the caller. If no request transform is configured, the request received is passed on to the upstream service unmodified, or returned to the caller unmodified if no upstream service was configured.

An optional *xformResponse* parameter defines an XSL or XQuery transform that is applied to the response returned from the configured upstream service prior to returning the response to the caller. If no upstream service is configured, this parameter has no effect.

By configuring an *xformRequest* parameter but no upstream service provider, the *Transformer* Component can be used to implement a simple microservice that operates on its input and returns some transformed variation of that input. An example of a microservice implemented using the *Transformer* Component is the LUHN validation service provided with the Banking Card Microservices package.

8.22.1. Configuration Example

In this scenario a downstream service has made a call into the OFS subsystem. The subsystem returns a SOAP response that contains a proprietary data format which is not understood by the downstream service, so the response message has been forwarded to this Transformation component. The transformation translates the proprietary data into an XML structure that can be validated, using the “OFSResponseToXML” transformation. The transformed response is then returned to the downstream service.

```
<Transformer
  name="fed-uad-ofs"
  wsdl="/wsdl/ccs-fed-uad/ccs-fed-uad.wsdl"
  xformResponse="/xsl/ofs/OFSResponseToXML.xsl" >
  <DisplayName>UAD to OFS</DisplayName>
  <Description>Transforms an OFS response to XML</Description>
```

```
  <Listening>
    <Address>/uad/fed-uad-ofs</Address>
  </Listening>
</Forwarder>
```

8.22.2. Type Information

8.22.2.1. Namespace

`http://worldwest.local/interface/1.0`
`<Transformer/>` belongs to the Interface namespace.

8.22.2.2. Type Definition

`<Transformer/>` is defined by an XSD 1.0 type named `int:TransformerType`.

8.22.2.3. Class

The Transformer component is implemented by the `ca.worldwest.mobius.web.servlet.Transformer` class.

8.22.24. Inherited Components

The `<Transformer/>` element inherits elements and attributes from

- *int:ComponentType*.
- *int:Forwarder*.

8.22.2.5. Derived Components

The following type definitions extend `int:Transformer`

- *int:AuthChallengeType*
- *int:DispatcherType*
- *int:ValidatorType*

8.22.3. Simple Schema Model

```
<Transformer
  xformRequest? = server url
  parse-request? = boolean
  xformResponse? = server url
  parse-response? = boolean
  requestContextXPath? = XPathExpression
[inherited from Forwarder]
  [upstreamServiceName? = string]
  [httpMethod? = string]
  [connectTimeout? = integer]
```

```
  [readTimeout? = integer]
  [eraseCookies? = boolean]
  [append-uri-tail? = boolean]
  [append-uri-params? = boolean]
  [append-root? = string]
  [ssl-ignore-host? = boolean]
  [allow-direct? = boolean]
[inherited from ComponentType]
  [name = string]
  [class? = string]
  [loginTokenXPath? = string]
  [contentType? = string]
  [faultHttpResponseCode? = string]
  [xformFault? = server url]
  [wsdl? = server url]
  [xformWSDL? = server url]
  [urlErrorOutputPath? = server url]
  [log-level? = "TRACE" | "DEBUG" | "INFO" | "WARN" | "ERROR" | "FATAL"]
  [recordingDisabled? = boolean]
  [writeToSequence? = string]
  [writeToSequenceName? = string]
  [xformRecordTitleRequest? = string]
  [xformRecordTitleResponse? = string]
  >
  <!-- Content: (Expect?)
[inherited from Forwarder]
                ([Headers?] [Forwarding?])
[inherited from ComponentType]
                ([DisplayName?] [Description?] [Listening] [Parameters?])
  -->
</Transformer>
```

```
<Expect>
  <!-- Content: (content-type+)
</Expect>
```

```
<content-type>
  <!-- Content: string -->
</content-type>
```

8.224. XML Model

```
<Transformer
  name="Component Name" ❶
  xformRequest="/server/path/to/transformation" ❷
```

```
  parse-request="true|false" ❸
  xformResponse="/server/path/to/tranformation" ❹
  parse-response="true|false" ❺
  requestContextXPath="XPath Expression" ❻
  [inherited attributes from Forwarder]
  [inherited attributes from ComponentType] >
  <Expect> ❼
    <content-type>MIME Type</content-type> ❽
    ⋮
  </Expect>
  [inherited elements from Forwarder]
  [inherited elements from ComponentType]
</Transformer>
```

❶ **`name`**

The name of the component. It must be unique within the system.

❷❹ `xformRequest`, `xformResponse`

A server path to a transformation script that will be used to transform the XML message request or response, respectively. Used to transform an input message into an output message, like converting a SOAP message from system A into a message compatible with system B, or vice-versa

If no `xformRequest` or `xformResponse` parameter is specified, the request or response is passed on unmodified to any endpoints configured in the `<Forwarding/>` element, or to the upstream request originator.

If no `<Forwarding/>` elements are configured but an `xformRequest` is specified, the request is transformed directly into a response using the specified `xformRequest`.

❸❺ `parse-request`, `parse-response`

When set to `false` the source document will converted to document node containing a singleton `xs:string` atomic value. The initial context node for the transformation may be matched against the document root node (`/`), current node (`.`), or wildcard (`*`). In XSLT, the transformation may also be initiated using `<xsl:initial-template/>`. In all cases processing will be continued by passing the singleton `xs:string` node to a function, as there is no XML content to be matched.

❻ `requestContextXPath`

An XPath to a value contained in the request message that will be passed as a parameter to the response transformation as a parameter named `requestContext`. This allows the response to access state from the request for use in its XSLT logic.

❼❽ `Expect`, `content-type`

Identifies MIME types to expect from the responding upstream service.

This allows the system to reject non-XML based responses prior to sending the response to the Transformer.

8.22.5. Notable Details

8.22.5.1. Implementing a Mock Service

Create an `xformRequest` transformation that returns responses from the MessageHistory or from a file or URL deployed on the application server based on logic defined in XSLT. See *Creating Mock Services* for an example.

8.22.5.2. XSLT/XQuery Transformer Capabilities

The level of XSLT or XQuery support is dependent on the XSLT processor. Avato uses the Saxon v9.8+ processor for XSLT, XQuery, and XML Schema. The edition of Saxon that is included with your Avato deployment is dependent on your licensing terms. The following capabilities are supported by the various Saxon editions:

SaxonHE v9.8+
: XSLT 1.0, XSLT 2.0 Basic, XSLT3.0 Basic, XQuery 1.0 Basic, XQuery 3.0 Basic, XQuery 3.1 Basic, XPath 2.0 Basic, XPath 3.0 Basic, XPath 3.1 Basic.

SaxonPE v9.8+
: XSLT 1.0, XSLT 2.0 Basic, XSLT 3.0 Basic/Higher-Order Functions, XPath 2.0 Basic, XPath 3.1 Basic/Higher-Order Functions, EXSLT and EXPath extensions, Saxon Extensions (Basic), SQL Extension.

SaxonEE v9.8+
: XSLT 1.0, XSLT 2.0 Basic/Schema Aware, XSLT 3.0 Basic/Higher-Order Functions/Schema Aware/Streaming, XPath 2.0 Basic/Schema Aware, XPath 3.0 Basic/Higher-Order Functions/Schema Aware, XPath 3.1 Basic/Higher-Order Functions/Schema Aware, XML Schema 1.0 Validation, XML Schema 1.1 Validation, EXSLT and EXPath extensions, Saxon Extensions (Basic/Advanced), SQL Extension, Compiled Code Generation, Large Document Projection, Stylesheet Package Export, Multithreading.

Refer to *Saxonica* for more details regarding their XSLT processing products.

8.23. Validator

The *Validator* component performs schema (DTD and XSD 1.0) validation of XML files.

Because the SOAP schemas allow any XML content within the `<SOAP:Header/>` and `<SOAP:Body/>` elements, the Avato validator will extract the header and body content and validate them separately, so as to provide full validation coverage of the message. Schemas are identified by their namespaces, alleviating the need to use `xsi:schemaLocation` in messages.

See *Avato: Catalogs* and *OASIS XML Catalogs Standard* for details about XML Catalogs.

8.23.1. Configuration Example

8.23.2. Type Information

8.23.2.1. Namespace

`http://worldwest.local/interface/1.0`
`<Validator/>` belongs to the Interface namespace.

8.23.2.2. Type Definition

`<Validator/>` is defined by an XSD 1.0 type named `int:ValidatorType`.

8.23.2.3. Class

The Validator component is implemented by the `ca.worldwest.mobius.web.servlet.Validator` class.

8.23.24. Inherited Components

The `<Validator/>` element inherits elements and attributes from

- *int:ComponentType*.
- *int:Forwarder*.
- *int:Transformer*.

8.23.2.5. Derived Components

No component type definitions extend `int:ValidatorType`.

8.23.3. Simple Schema Model

```
<Validator
  catalog? = server url
[inherited from Transformer]
  [xformRequest? = server url]
```

```
  [parse-request? = boolean]
  [xformResponse? = server url]
  [parse-response? = boolean]
  [requestContextXPath? = XPathExpression]
[inherited from Forwarder]
  [upstreamServiceName? = string]
  [httpMethod? = string]
  [connectTimeout? = integer]
  [readTimeout? = integer]
  [eraseCookies? = boolean]
  [append-uri-tail? = boolean]
  [append-uri-params? = boolean]
  [append-root? = string]
  [ssl-ignore-host? = boolean]
  [allow-direct? = boolean]
[inherited from ComponentType]
  [name = string]
  [class? = string]
  [loginTokenXPath? = string]
  [contentType? = string]
  [faultHttpResponseCode? = string]
  [xformFault? = server url]
  [wsdl? = server url]
  [xformWSDL? = server url]
  [urlErrorOutputPath? = server url]
  [log-level? = "TRACE" | "DEBUG" | "INFO" | "WARN" | "ERROR" | "FATAL"]
  [recordingDisabled? = boolean]
  [writeToSequence? = string]
  [writeToSequenceName? = string]
  [xformRecordTitleRequest? = string]
  [xformRecordTitleResponse? = string]
  >
  <!-- Content: ()
[inherited from Transformer]
                ([Expect?])
[inherited from Forwarder]
                ([Headers?] [Forwarding?])
[inherited from ComponentType]
                ([DisplayName?] [Description?] [Listening] [Parameters?])
  -->
</Validator>
```

8.234. XML Model

```
<Validator
```

```
  name="Component Name" ❶
  catalog = '/server/path/to/catalog|/xsd-next-catalog' ❷
  [inherited attributes from Transformer]
  [inherited attributes from Forwarder]
  [inherited attributes from ComponentType] >
  [inherited elements from Transformer]
  [inherited elements from Forwarder]
  [inherited elements from ComponentType]
</Validator>
```

❶ **name**

The name of the component. It must be unique within the system.

❷ `catalog`

A server path to an XML Catalog that can cross-reference URLs to schemas. The default path points to the current catalog. See Avato: Catalogs and OASIS XML Catalogs Standard for further information.

Part IV. Advanced Development Topics

9 Writing SOAP Service Tests

In addition to providing XML validation through schemas, Avato supports the creation of arbitrarily complex tests of business rules, useful for validating the data returned by SOAP services. These tests can include complex test set-up involving calls to other services to obtain data for the true test, can use test data in calculating whether response data is correct, and can selectively ignore errors that are inconsequential to fulfilling the rules.

In addition to reading this rough guide to tests, look at the WSDL Template Service tests in the Avato packages directory: they have code documentation, more detail than provided in the code samples in this section, and are functioning to-spec.

9.1. Test Flow Overview

Note

Example source code in this overview section has been simplified for clarity, removing some required attributes and elements that are not important for understanding how tests are run. Later sections provide full coverage of the code.

Test Scenarios

Tests are guided by a collection of *Test Scenarios*. Among other things, each test scenario identifies the test entry template and a collection of values to be used by the template.

Simplified Test Scenario and Test Values

```
<Scenario template="s11.GetOperationTemplate">❶
  <title>Get a GetListOfOperations Template</title>
  <datasource idref="getOperationTemplate" />❷
</Scenario>

<DataSource xml:id="getOperationTemplate">❸
  <parameter name="operationName" value="GetListOfOperations" />
  <parameter name="portName" value="WSDLPort11" />
  <parameter name="serviceName" value="WSDLService" />
  <parameter name="serviceNamespace" value="http://worldwest.local/wsdl/
service" />
</DataSource>
```

❶ A name that matches the test entry template.

❷ Identifies the collection of values (a data source node) that will be used by the test.

❸ The test's data source node.

Test Runner

The test runner loops through test scenarios, generating an element on-the-fly using the scenario's template value; this element will be matched by the template providing the entry point for the test.

Test Runner

```
<xsl:for-each select="$testcases/Tests/TestCases/TestCase/Scenario">
  <xsl:variable name="dsid" select="datasource/@idref" />
  <xsl:variable name="elem">❶
    <xsl:element name="{@template}" namespace="http://worldwest.local/test" />
  </xsl:variable>
  <xsl:apply-templates select="$elem">❷
    <xsl:with-param name="token" select="$token" tunnel="yes" />
    <xsl:with-param name="esbnode" select="$esbnode" tunnel="yes" />
    <xsl:with-param name="test.scenario" select="current()" tunnel="yes" />❸
    <xsl:with-param name="data.matrix" select="$testcases//DataSource[@xml:id =
 $dsid]" tunnel="yes" />❹
  </xsl:apply-templates>
</xsl:for-each>
```

❶ Creates a node using the template name identified in the Scenario.

❷ `$elem` will be matched by the test entry template.

❸ The test scenario itself.

❹ The test's data source node.

Test Entry Template

The matching test entry template emits a request that will generate a response that will be tested.

Test Entry Template, Emitting a Request

```
<xsl:template match="t:s11.GetOperationTemplate">❶
  <xsl:param name="token" required="no" tunnel="yes" />
  <xsl:param name="esbnode" required="no" tunnel="yes" />
  <xsl:param name="test.scenario" required="no" tunnel="yes" />
  <xsl:param name="data.matrix" required="no" tunnel="yes" />

  <xsl:variable name="test.name"
 select="concat('s11.GetOperationTemplate.', $test.scenario/xml:id)" />❷
  <esb:Services>
    <esb:XmlRequest name="$test.name" uri="{concat($esbnode,'/wsdl')}">
      <xsl:call-template name="Soap11Wrapper">
```

```
        <xsl:with-param name="header">
          <auth:Token id="$token" />
        </xsl:with-param>
        <xsl:with-param name="body">
          <ws:GetOperationTemplate  ❸
            operationName="{f:parameter('operationName', $test.scenario,
 $data.matrix)}"  ❹
            portName="{f:parameter('portName', $test.scenario, $data.matrix)}"
            serviceName="{f:parameter('serviceName', $test.scenario,
 $data.matrix)}"
            serviceNamespace="{f:parameter('serviceNamespace', $test.scenario,
 $data.matrix)}"
            />
        </xsl:with-param>
      </xsl:call-template>
    </esb:XmlRequest>
  </esb:Services>
</xsl:template>
```

❶ The template matches the element that was created on-the-fly by the test runner using the name provided by the test scenario.

❷ The test ID must be appended to the test name so that the request and response are uniquely associated with the test. Please see important notes about test naming in the section called “Staged Testing Templates”.

❸ The service that is being tested.

❹ `f:parameter` extracts a value from the test’s data source node.

Handling Responses

Many tests will be relatively simple, with a single request generating a single, testable response.

Other tests will require some amount of test set-up. In such a cases, the test entry response issues its own request: either a request fetching more data for the test request; or the ultimate test request itself.

In all cases, Dispatcher sends responses back into the transformation, wrapping it with an element identified by the `<esb:XmlRequest />` `name` attribute. These responses **must** be matched. We suggest using `<xsl:mode on-no-match="fail" />` to ensure missing responses are matched. Keep in mind that each test will generate a unique name for the response.

Here is one way to handle the response:

```
<xsl:template match="/*[starts-with(name(),'s11.GetOperationTemplate') and
 namespace-uri()='']" />
```

This example uses `starts-with` to ignore the test ID. Because the `test.name` variable in the previous example uses a name identical to the one used for the

test entry match, we test for an empty `namespace-uri`; otherwise this template would match both the element that is intended to match the test entry template as well as the response.

Alternatively, one could use a `test.name` that doesn't have the same start-with string: `test.s11.GetOperationTemplate` would alleviate the need to use `namespace-uri` to distinguish the two template matches.

Testing Responses

When all test set-up requests and test requests have been sent a `<DispatchResult />` document is sent into the transformation. This document is captured by the test support framework, which will extract the final response for each test request/response sequence and send it back into the transformation using `mode='test'`. Note that the response **is not wrapped** in an `esb:XmlRequest` name.

Assuming the `GetOperationTemplate` request above asked for the template for the "GetListOfOperations" service, the following template tests that the response is correct:

Testing the Response

```
<xsl:template match="/s11Env:Envelope[*:Body/*:GetListOfOperations]"
 mode="test">❶

  <xsl:variable name="expected">❷
    <s11Env:Envelope xmlns:ws="http://worldwest.local/wsdl/schemas">
      <s11Env:Header />
      <s11Env:Body>
        <ws:GetListOfOperations renderAs="soap|html|xml" wsdlURI="http://
example.com" />
      </s11Env:Body>
    </s11Env:Envelope>
  </xsl:variable>

  <xsl:variable name="test" select="t:CompareDocuments($expected,self::*)" />❸

  <xsl:variable name="errors">
    <xsl:sequence select="f:simple.elements($test)/*[not(self::*:Header)]" />❹
    <xsl:sequence select="f:simple.attributes($test)/*" />❺
    <xsl:sequence select="f:simple.content($test)/*" />❻
  </xsl:variable>

  <xsl:sequence select="t:TestResults($errors, $test)" />❼

</xsl:template>
```

❶ A match on the response that is to be tested. A predicate test is used, preserving the SOAP envelope as the context.

❷ The expected response, against which the received response will be tested.

❸ Performs the comparison, returning a complex test results document.

❹ `f:simple.elements` extracts errors where element names did not match or were out of sequence; we exclude the SOAP Header from those results.

❺ `f:simple.attributes` extracts errors where attribute names did not match.

❻ `f:simple.content` extracts errors where element or attribute content did not match.

❼ Emits a test results document.

During the test phase, all the scenario's requests and responses are available for use in tests, enabling you to use knowledge of the request data to identify correct response data; for instance, a customer name used in a request might be used to check that the same customer name is being returned in the response.

Displaying Results

Avato UI receives the test results document and displays the test results and the requests and responses generated in running the test.

Test Scenario Design

Tests comprise a series of *Test Cases* each with one or more *Scenarios*. Each test case corresponds to a particular service request, and each scenario presents a different set of parameters against which to test the service. Parameters can be shared between tests and within a scenario the default parameter values may be overridden. Tests can be flagged as "expected to fail", enabling you to test both positive and negative outcomes. Finally, an alternative expected result can be provided, making it somewhat easier to confirm that fault responses are correct.

Note that the test scenarios file can be validated using the `TestData.xsd` schema.

Structure of a Test Scenarios File

```
<Tests
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="TestData.xsd">

  <TestCases
    uri="/test/wsdl"   ❶
    xml:id="wsdl-tests"   ❷
    >
    <title>WSDL Service Test Suite</title>

    <TestCase xml:id="TC-001">
      <title>Get a List of Operations</title>
```

```
      <description>Gets a list of operations available through the service.</
description>

      <Scenario
        template="s11.GetListOfOperations" ❸
        xml:id="TC-001.01"  ❹
        >
        <title>Get a List of Operations – SOAP11</title>
        <description>Gets a list of operations available through the service.</
description>
      </Scenario>

      <Scenario template="s12.GetListOfOperations" xml:id="TC-001.02">
        <title>Get a List of Operations – SOAP12</title>
        <description>Gets a list of operations available through the service.</
description>
      </Scenario>
    </TestCase>

    <TestCase xml:id="TC-002">
      <title>Get a GetListOfOperations Template</title>
      <description>Gets a request template for the GetListOfOperations
 request.</description>

      <Scenario template="s11.GetOperationTemplate" xml:id="TC-002.01">
        <title>Get a GetListOfOperations Template – SOAP11</title>
        <description>Gets a request template for the GetListOfOperations
 request.</description>
        <datasource idref="getListTemplate" /> ❺
      </Scenario>

      <Scenario template="s12.GetOperationTemplate" xml:id="TC-002.02">
        <title>Get a GetListOfOperations Template – SOAP12</title>
        <description>Gets a request template for the GetListOfOperations
 request.</description>
        <datasource idref="getListTemplate" />
      </Scenario>
    </TestCase>

    <TestCase xml:id="TC-003">
      <title>Get a GetOperationTemplate Template</title>
      <description>Gets a request template for the GetOperationTemplate
 request.</description>

      <Scenario template="s11.GetOperationTemplate" xml:id="TC-003.01">
        <title>Get a GetOperationTemplate template – SOAP11</title>
```

```
        <description>Gets a request template for the GetOperationTemplate
 request.</description>
        <datasource idref="getOperationTemplate" />
      </Scenario>

      <Scenario template="s12.GetOperationTemplate" xml:id="TC-003.02">
        <title>Get a GetOperationTemplate template – SOAP12</title>
        <description>Gets a request template for the GetOperationTemplate
 request.</description>
        <datasource idref="getOperationTemplate" />
      </Scenario>
    </TestCase>

    <TestCase xml:id="TC-004">
      <title>Bad GetOperationTemplate Requests</title>
      <description>Get a request template for an operation that does not exist</
description>

      <Scenario template="s11.GetOperationTemplate" xml:id="TC-004.01">
        <title>Get a BogusOperation template - SOAP11</title>
        <description>Override the default template, substituting the name of a
 template that does not exist.</description>
        <datasource idref="getOperationTemplate" />
        <negative>true</negative> ❻
        <override name="operationName" value="BogusOperation" /> ❼
        <expected> ❽
          <s11Env:Envelope xmlns:s11Env="http://schemas.xmlsoap.org/soap/
envelope/">
            <s11Env:Header />
            <s11Env:Body>
              <s11Env:Fault>
                <s11Env:Code>
                  <s11Env:Value>env:Sender</s11Env:Value>
                </s11Env:Code>
                <s11Env:Reason>
                  <s11Env:Text xml:lang="en">Exception</s11Env:Text>
                </s11Env:Reason>
                <s11Env:Detail>
                  <errorcode>SYS012</errorcode>
                  <message>An empty sequence is not allowed as the result of
 call to f:getOperationByNameFromBinding; SystemID: https://localhost:8443/esb/
services/wsdl/api/wsdlAPI-utility.xslt; Line#: 289; Column#: 34</message>
                </s11Env:Detail>
              </s11Env:Fault>
            </s11Env:Body>
          </s11Env:Envelope>
        </expected>
```

```
      </Scenario>
    </TestCase>

  </TestCases>
  <TestData>
    <DataSource xml:id="getListTemplate">❾
      <title>getListTemplate</title>
      <parameter name="operationName" value="GetListOfOperations" />
    </DataSource>
    <DataSource xml:id="getOperationTemplate">
      <title>getOperationTemplate</title>
      <parameter name="operationName" value="GetOperationTemplate" />
    </DataSource>
  </TestData>
</Tests>
```

❶ The test suite entry endpoint, defined in the test suite web fragment.

❷ An identifier for this test suite, unique among all test suites.

❸ The `local-name()` of the test entry template, used to create an element that will be matched by the template.

❹ An unique identifier for this particular test.

❺ A cross-reference to a `<DataSource />` element providing parameters for the test.

❻ A flag to indicate that the test is expected to fail (and thus passes the test.)

❼ A parameter that overrides the value provided in the `<DataSource />` element in ❺.

❽ The expected response to this test, overriding the response hardcoded in the test.

❾ An element providing parameters for a test.

It is possible to “flag” some overrides for special purposes; for instance, overriding a parameter and using a value of `today` will generate today’s date in MM-DD-YYYY format. This functionality can be customized; see *the section called “Test Template Design”* below.

Test Template Design

The following template can be used almost verbatim, requiring customization of only the test inclusions and the test data URI.

```
<xsl:stylesheet
  xmlns:esb="http://worldwest.local/esb"
```

```
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0" >

  <!-- include the test framework -->
  <xsl:import
    href="/msg/sys/TestFramework/xsl/TestRunner.xsl" /> ❶

  <!-- bring in the WSDL tests -->
  <xsl:include href="/msg/sys/TestSuites/WSDL-Tests/xsl/services/
s11.GetListOfOperations.xsl" />
  <xsl:include href="/msg/sys/TestSuites/WSDL-Tests/xsl/services/
s11.GetOperationTemplate.xsl" />
  <xsl:include href="/msg/sys/TestSuites/WSDL-Tests/xsl/services/
s12.GetListOfOperations.xsl" />
  <xsl:include href="/msg/sys/TestSuites/WSDL-Tests/xsl/services/
s12.GetOperationTemplate.xsl" /> ❷

  <xsl:param
    name="testcases.uri"
    select="'/msg/sys/TestSuites/TestData/wsdl.TestData.xml'" /> ❸

  ❹

</xsl:stylesheet>
```

❶ Importing the TestRunner allows you to override the framework's templates and functions.

❷ Include all the tests.

❸ This value will be passed by parameter in the future.

❹ Everything else is provided by the test framework.

The `f:parameter` function in the TestRunner.xsl file provides "flags" that can be used in the TestData overrides to provide calculated data values. Copy the function into your test template and modify to your satisfaction.

9.2. Demonstration: WSDL GetOperationTemplate Test

9.2.1. Test Entry Template

Every test enters on a template identified by the test scenario `@template` attribute. This template must belong to the *`http://worldwest.local/test`* namespace to prevent accidental name collisions. It will always take four parameters as shown in the listing below.

Tests use the Dispatcher service to emit service calls and process the responses. In a simple test there will be two templates: one to emit the test request using values from its test scenario, and another to process the response.

Complex tests may require collection of data to set up the test, emitting several requests to other services, and using the responses as a source of values to be used in the test. Ultimately, a test request will be emitted and a final template will match and process its response.

In the source code example below, the test requires some set-up before it can be executed: before a `GetOperationTemplate` call can be made, the port, service, and service name for the operation must be obtained. To get this information, a call to `GetListOfOperations` is made; the response will be captured by name and used in the creation of the test request.

Testing: Test Entry Template

```
<!--
  SOAP11 STEP ONE
  Get List of WSDL Template Service Operations
  The eventual call to GetOperationTemplate will require the serviceName,
  serviceNamespace, and portName; this call will get that data.
  The operation name is supplied by the test data
  token: The authorization token for the test suite. Required.
  esbnode: The server on which to run the tests. Required.
  test.scenario: The test scenario, from the test's TestData.xml document.
 Required.
  data.matrix: The data for the test scenario, from the test's TestData.xlm
 document.
-->

<xsl:template match="t:s11.GetOperationTemplate">❶
  <xsl:param name="token" required="yes" tunnel="yes" />❷
  <xsl:param name="esbnode" required="yes" tunnel="yes" />
  <xsl:param name="test.scenario" required="yes" tunnel="yes" />
  <xsl:param name="data.matrix" required="no" tunnel="yes" />
  <!-- for the sake of human readability,
       prepend the name of the service to the test id -->
  <xsl:variable
    name="test.name"
    select="concat('test.stage.1.s11.GetOperationTemplate.', $test.scenario/
@xml:id)" />❸
  <esb:XmlRequest
    name="{$test.name}"
    uri="{concat($esbnode,'/wsdl')}">❹
    <xsl:call-template name="Soap11Wrapper">
      <xsl:with-param name="header">
        <auth:Token id="{$token}" />
        <t:test
```

```
          esbnode="{$esbnode}"
          name="{$test.name}"
          id="{$test.scenario/@xml:id}"
          operationName="{f:parameter('operationName', $test.scenario,
 $data.matrix)}" /> ❺
      </xsl:with-param>
      <xsl:with-param name="body">
        <ws:GetListOfOperations
          renderAs="html"
          wsdlURI="{concat($esbnode,'/wsdl?wsdl')}" /> ❻
      </xsl:with-param>
    </xsl:call-template>
  </esb:XmlRequest>
</xsl:template>
```

❶ The test entry template must use the `http://worldwest.local/test` namespace and its local-name must match the value provided in the `template` attribute of its corresponding scenario.

❷ The template must include these four parameters as written.

❸ `<esb:XmlRequest />` elements must be named using their test scenario `xml:id` attribute. When writing a test set-up request, the name **must** include the prefix `test.stage` or `test-stage` so that the set-up responses can be excluded from test analysis.

❹ The set-up request is going to be sent to the same server node that is being tested. This treads closely to testing the server node; if this request failed, the overall test would unexpectedly fail. A better test design might specify a known-good server node in the scenario parameters.

❺ In complex tests it is convenient to pass parameters by including them in the request header. These can be extracted later by requesting the request document.

❻ This makes a service call to fetch information we need in order to set up the test. The response will be matched on its `<esb:XmlRequest />` name.

Staged Testing Templates

The previous source code listing provided a name for the `<XmlRequest />` that fetches additional data for the test itself. This name is matched by the next template in the test set-up sequence, which may issue additional data-gathering requests, or may emit the ultimate test request.

Important

When writing a test set-up request, the name **must** include the prefix `test.stage` or `test-stage` so that the set-up responses can be excluded from test analysis.

In this example, it emits the test request using values from both the test entry request and the subsequent set-up request.

Complex Testing: Template Stages

```
<!--
  SOAP 11 STEP TWO
  Request a Specific Template from the WSDL Template Service.
  Step one provided values for several parameters; others are provided by the
 scenario.
-->

<xsl:template
  match="/*[starts-with(name(), 'test.stage.1.s11.GetOperationTemplate')]">❶
  <xsl:variable
    name="request.doc"
    select="document(concat('/Requests/', name(.)))" />❷
  <xsl:variable
    name="test.name"
    select="concat('s11.GetOperationTemplate.', $request.doc//t:test/@id)" />❸
  <xsl:variable
    name="esbnode"
    select="$request.doc//t:test/@esbnode" />❹
  <esb:XmlRequest
    faultaction="ignore"
    name="{$test.name}"
    uri="{concat($esbnode,'/wsdl')}" >
    <xsl:call-template name="Soap11Wrapper">
      <xsl:with-param name="header">
        <auth:Token id="{$request.doc//auth:Token/@id}" />
      </xsl:with-param>
      <xsl:with-param name="body">
        <ws:GetOperationTemplate
          operationName="{$request.doc//t:test/@operationName}"  ❺
          portName="{//ws:service/ws:port[ws:type='soap11']/ws:name}"
          serviceName="{//ws:service/ws:name/text()}"
          serviceNamespace="{//ws:service/ws:namespace}"
          wsdlURI="{concat($esbnode,'/wsdl?wsdl')}"
          />
      </xsl:with-param>
    </xsl:call-template>
  </esb:XmlRequest>
</xsl:template>
```

❶ This predicate excludes the scenario `xml:id` component of the name, allowing every scenario to be matched.

❷ This fetches the test entry request document for a specific scenario. The name of the document includes the scenario `xml:id` component.

❸ The response to this request will be one that we want to test, so it does not get the `test.stage` prefix. We have to jump through some hoops to get the required test ID. It has been stashed in the test entry request SOAP Header.

❹ Similarly, we recall the server node on which the test is to be run.

❺ And the operation name was also stashed in the `t:test` header. Other parameters are obtained from the current context, the `GetListOfOperations` response document.

Don't forget to handle the response that will be generated by this request!

Complex Testing: Discard the Final Response

```
<!--
  SOAP 11 STEP THREE
  Dispatcher sends the (named) response back in: simply discard it.
  The DispatchResult handler will eventually handle everything in "test" mode.
-->

<xsl:template
  match="/*[starts-with(name(),'s11.GetOperationTemplate')
  and namespace-uri()='']" />
```

This completes the set-up of the test request, and the handling of all the responses that it generates. Next, the test framework will send the final responses back into the transformation using `mode='test'`.

Test Response Analysis

Recall that a `<DispatchResult />` document is sent into the transformation as the last step of a dispatched service. Thus, the last step in a test is to write a template that uses a predicate to match the expected response, using `mode="test"` to ensure that it does not capture responses that were generated during a test set-up stage.

Analysis of the response returned by a test request comprises four parts:

- The expected response, which may include data values determined by calculation or extracted by XPath from test requests and responses; and which may include `t:position` attributes that allow some amount of flexibility in determining if a response is valid.
- A call to the received vs. expected response comparison utility.
- Simplification of the test analysis result.

- Creation of a `<TestResults />` document.

Most test suites will test any given service operation using a variety of test parameters. The responses to these tests will often be similar, allowing a single programmatically-defined expected response to provide coverage for most scenarios. When necessary, a static expected response can be provided in the test scenario; these are especially useful when a test is designed to fail with a fault.

Test Response Analysis

```
<!--
  SOAP 11 STEP FOUR
  Test the response of a GetOperationTemplate request.
  Compares the received response to an expected result.
  A template for GetListOfOperations was requested.
  Ignores:
    Header elements.
  Return:
    A TestResults element.
-->

<xsl:template
  match="/s11Env:Envelope[*:Body/*:GetListOfOperations]"
  mode="test" >❶

  <xsl:param name="expected">❷
    <s11Env:Envelope>
      <s11Env:Header />
      <s11Env:Body>
        <tns:GetListOfOperations
          renderAs="soap|html|xml"
          wsdlURI="http://example.com"
          xmlns:tns="http://worldwest.local/wsdl/schemas" />
      </s11Env:Body>
    </s11Env:Envelope>
  </xsl:param>

  <xsl:variable name="test" select="t:CompareDocuments($expected,self::*)" />❸

  <xsl:variable name="errors">❹
    <xsl:sequence select="f:simple.elements($test)/*[not(self::*:Header)]" />
    <xsl:sequence select="f:simple.attributes($test)/*" />
    <!--<xsl:sequence select="f:simple.content($test)/*" />-->❺
  </xsl:variable>

  <xsl:sequence select="t:TestResults($errors, $test)" />❻
```

```
</xsl:template>
```

❶ The predicate test ensures that the template context remains at the root element level; the use of `mode="test"` prevents this match from capturing responses generated by set-up stages.

❷ The expected response does not have to be complete; here, we have eliminated Header descendents; attributes and content may also be selectively discarded.

❸ The `CompareDocuments` function returns a complex result document, which we subsequently filter.

❹❺ Use `f:simple.*` to extract elements that are in error. Errors that we do not care about can be excluded by applying a predicate test, as seen here to exclude the Header errors (a result of not providing a complete `<s11Env:Header />` expectation). Content errors are excluded here by commenting-out their extraction.

❻ Finally, we call `TestResults` to emit the final test document, to be processed by Avato to generate its web UI.

Writing Test Expectations

The heart of a test is performed in comparing the received response with an expected response.

Validation of a response against schemas will confirm that the XML structure is correct, and that data contained within are of the correct type and meet certain value specifications, but can not always determine whether the data itself is correct.

Because we are writing our tests in XSL and running them using the Avato *Dispatcher*, we have full access to all test set-up requests and responses, and can calculate values dynamically, enabling testing of business rules. Two simple examples follow:

Example Programmatically Defined Test Value

```
<xsl:variable
  name="request"
  select="doc(concat('Requests/',//*:Message/@document))" />❶

<!-- Test that the sequence number was incremented -->
<SequenceNumber>
  <xsl:value-of
    select="xs:integer($request//t:test/@seqnum)+1" />
</SequenceNumber>

<!-- Test that the operation succeeded or failed as expected -->
<OperationSuccessful>
```

```
  <xsl:value-of
    select="if ($request//t:test/@isvalid = 'true') then 1 else 0" />
</OperationSuccessful>
```

❶ Avato adds some information to response SOAP Headers, including an identifier for the originating request. We can then retrieve values that were passed along in the header, allowing us to dynamically calculate response values.

Element Position Hints

A `t:position` attribute can be added to an expected response element to provide hints about where an element should appear, allowing a certain amount of flexibility in dealing with elements that may be repeated or omitted.

- `first`: the element must be the first child of its parent.
- `last`: the element must be the last child of its parent.
- any: the element may appear in any position among the children of its parent; it may be repeated.
- `omit`: omit the element and its children from the test (the element is still needed to keep the structural pattern synchronized between the received and expected response.)
- integer value: the element must appear as the *n*th child of its parent.

Example Positional Test

```
<!-- A repeating element -->
<MemberAccounts>
  <Account
    t:position="any" />
</MemberAccounts>

<!-- New data elements will be first in the list and its sequence number will
 match that of the request that generated the new element -->
<Transactions>
  <Transaction
    seqnum="{xs:integer($request//t:test/@seqnum)}"
    t:position="first" />
  <Transaction
    t:position="any" />
</Transactions>
```

Writing Error Filters

The comparison utility returns a complex document detailing the nature of every failure and its associated elements. Three utility functions are provided to allow

you to filter these errors by their type, by the failed element, or by the type of failure.

Error Categories

Utility functions are provided to enable selective selection and rejection of errors:

- `f:simple.elements`: returns a list of elements with failed children.
- `f:simple.attributes`: returns a list of elements with failed attributes.
- `f:simple.content`: returns a list of elements with failed content.

If you need finer-grained control of error filtering, you can inspect the **Errors** report in the test results, where the raw test output is provided. You can then write predicates that, for instance, filter out elements that are optional:

Example Error Filter

```
<xsl:copy-of
  select="f:simple.elements($test)/*[not(self::*:Header
  or self::*:Restrictions[t:received = 'false'])]" />
```

This example rejects errors where the `Restrictions` element does not appear in the received response.

Final Words

The WSDL *Get Operations* and *Get Templates* test suite is included in your packages folder. Inspect its `package.xml` and `web-fragment.xml` files to learn more about packaging; and review its test data (`wsdl.TestData.xml`), test entry (`Run.xsl`) and service tests (in the `services` subfolder) to learn more about writing a test suite.

Elided…

Part V. Appendices

Build Process Details

The following information is optional reading.

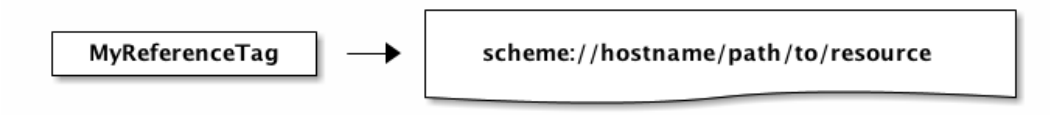
The build process performs tokenization in the following order, to a range of files:

1. The version token in *root*/properties/version.properties is applied to *root*/conf/web-interfaces.xml.
2. A set of hardcoded tokens based on the value of the -Dcontext-root build parameter (default of esb) is applied to the environment properties file *root*/properties/*env-name*/properties.xml.
3. The environment properties tokens and context-root tokens are applied to *root*/conf/web-interfaces.xml
4. For each package, its package properties tokens, the environment properties tokens, and context-root tokens are applied to its properties.xml configuration file.
5. The environment properties tokens and context-root tokens are applied to *root*/your-platform-name/jboss-web.xml
6. The environment properties tokens and context-root tokens are applied to *root*/your-platform-name/mobius.catalog.xml

XML Catalogs

B.1. Overview

XML Catalogs is a W3C Specification for entity management or, in other words, a standard for cross-referencing an object identifier with its object, so that these objects may be shared between information systems.



Avato makes use of two sets of XML Catalogs:

- One for resolving URIs for `<include/>` and `<import/>` elements (used in XSLT, WSDL, and XSD) and XPath references; and
- One for resolving schemas through their DOCTYPE Public or System ID, a schema location hint, or a targetNamespace URI.

B.2. Resolving URIs for Includes, Imports, and XPath References

Several XML schemas permit modularization of source files through the use of `<include/>` or `<import/>` elements. These typically take a form similar to `<include href="/path/to/file" />`. Some XPath functions also refer to files, including the commonly-used `doc()` function.

There are three common paths to files on an Avato instance:

- `/path/to/file` — used to locate resources stored in the Avato file system, rooted on the *`root`*`/web/` directory, using absolute referencing.
- `path/to/file` — used to locate resources stored in the Avato file system, from within the *`root`*`/web/` directory, but using relative referencing. Note the lack of a leading slash ("/").
- `/msg/sys/path/to/file` — used to locate resources stored by the Avato database servlet.

Note that when modularizing packages that web resources are copied to the server *`root`*`/web/` directory. Packages **must not** use folders that duplicate system servlet paths like `/msg/` or `/ui/`.

A package that uses relative references to files within its directory (`href="path/to/file"`) will continue have those resources resolved correctly.

A package may also use absolute references to files within the server web directory (`href="/path/to/file"`); these resources will be resolve to *root*`/web/path/to/file`, allowing you to access core functions like `xsl/SoapWrappers.xsl`.

A package can also use references to files within the server database (`href="/msg/sys/path/to/file"`).

B.2.1. Configuration of Avato URI Resolution

Resolution of such URIs is handled by a catalog and Avato component identified in the *root*`/src/CatalogManager.properties` configuration file:

`CatalogManager.properties`

```
catalogs=mobius.catalog.xml
relative-catalogs=no
static-catalog=no
catalog-class-name=org.apache.xml.esbresolver.ESBResolver
verbosity=99
prefer=public
```

As indicated by the `catalogs` setting on the first line of the configuration file, the root catalog file is named `mobius.catalog.xml`. Prior to building the application this file is located at *root*`/conf/wildfly12/mobius.catalog.xml`. After the application is built it resides in `Avato.war` in the `WEB-INF` subdirectory as a sibling to the `CatalogManager.properties` file.

`mobius.catalog.xml` before building Avato

```
<catalog
  xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <rewriteURI
    rewritePrefix="@esb.address.insecure@"
    uriStartString="http://worldwest.local" />
</catalog>
```

Note that this pre-build catalog makes use of a properties token: `@esb.address.insecure@`. This value is configured in a *root*`/properties/`*env-name*`/properties.xml` file. At build time, the @-delimited token is replaced by an appropriate value, such as `http://www.myserver.com/esb`. See the *Properties Configuration* section of the Avato Developers Guide for token replacement details.

`mobius.catalog.xml` after building Avato

```
<catalog
```

```
  xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <rewriteURI
    rewritePrefix="http://localhost:8080/esb"
    uriStartString="http://worldwest.local" />
</catalog>
```

The URI resolution catalog is very simple: it rewrites prefixes that match the specified `uriStartString` with the value in `rewritePrefix`. In other words, it replaces `http://worldwest.local/path/to/file` with `http://localhost:8080/esb/path/to/file`.

B.3. Resolving Schemas

When validating an XML file, a corresponding schema must be identified. There are several mechanisms available:

- Using a DOCTYPE instruction before the root element. This instruction typically looks like `<!DOCTYPE root-element PUBLIC "public identifier" "system identifier" />`.
- Using an schema location hint in the root element attributes. This typically looks like `<root-element xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://uri path/to/schema.xsd" />`. Note the space in `xsi:schemaLocation` separating the schema namespace from the XSD location URI.
- When validating a WSDL or XSD, using the root element target namespace attribute. This typically looks like `<root-element targetNamespace="uri" />`.

In all cases, schemas are resolved using the *Catalog* servlet, which iterates through catalogs located in the Avato database, looking for a match for the schema URI and resolving that match to a file location.

B.3.1. Configuration of Avato Schema Resolution

The Avato *Validator* component calls upon the *Catalog* servlet to perform schema URI resolution. As with all servlets, the Catalog component is configured in a `web.xml` deployment descriptor file. It is typically located at `https://server_url/context_root/xsd-next-catalog`.

The Catalog component searches the Avato filesystem and database for catalogs, modifying them on-the-fly to link them together through the use of the catalog `<nextCatalog/>` element. This enables catalog resolution to be dynamic, changing as Avato packages are loaded or unloaded. These additional catalogs will be accessed at `https://server_url/context_root/xsd-next-catalog?i=n` where *n* is `1` or greater.

Document Conventions

The schemas and XML samples use the following conventions when describing content:

C.1. Data Types

string, *integer*, *boolean*, *anyURI*, *token*, *Name*, *ID*
: A value conformant to its type definition in W3C XML Schema.

package file path
: A path to a `package.xml` file, starting from the root `/packages` directory. This path will have a leading slash. `/Social/LinkedIn/package.xml` references /path/to/mobius `/packages/Social/LinkedIn/package.xml`.

relative file path
: A path relative to the directory containing the current `package.xml` file. This path will not have a leading slash. Given a package file /path/to/mobius `/packages/Social/LinkedIn/package.xml`, a reference to `wsdl/LinkedIn.xsd` points to /path/to/mobius `/packages/Social/LinkedIn/wsdl/LinkedIn.xsd`.

absolute file path, *absolute directory path*
: A absolute path to a file or directory pointing to a fixed location on disk. This path will have a leading slash. An example of an absolute path is `/var/tmp/myfile.xml`.

server url
: A URL for a resource on the Avato server, with the scheme, host name, port, and Avato root path removed, e.g. `http://myserver:8080/root/myApp/index.html` becomes `/myApp/index.html`. This path will have a leading slash.

url
: A URL for an arbitrary resource on any server, complete with scheme, host name, and port, e.g. `http://anyserver:8080/some_app/index.html` or `\mailto:alertme@example.com`.

XPathExpression, *XSLTMatchPattern*
: An expression conformant to its definition in XPath 3.1 and XSLT 3.0, respectively.

C.2. Schema Models

The notation of a schema model is as follows:

- Required attributes are shown with their name in bold text.
- Optional attributes are shown in plain text and are suffixed with a question mark (?).
- Value descriptions or types are *italicized* and are not quoted.
- Fixed attribute values are shown as "quoted plain text".
- Alternative attribute values are separated by a vertical bar (|).
- Unless the element must be empty, a comment specifies the allowable content.
- Elements or element groups separated by a space ( ) may appear in any order.
- Elements or element groups separated by a comma (,) must appear sequentially.
- Elements or element groups separated by a vertical bar (|) are choices.
- Elements that are required (exactly one) are bold and do not have a suffix.
- Elements that are required (one or more) are bold and are suffixed with a plus sign (+).
- Elements that are optional (zero or one) are suffixed with a question mark (?).
- Elements that are optional (zero or more) are suffixed with an asterisk (*).
- Inherited attributes and elements are wrapped in heavy brackets (⟦ and ⟧)
- The description of text content is italicized.

```
<ns:example-element
  id = string
  activate? = "yes" | "no" >
  <!-- Content: ((Name Description?), Device*) -->
</Package>
```

This example defines an element `ns:example-element`. It has a mandatory `id` attribute with a string value. It has an optional `activate` attribute that must have a value of `yes` or `no`. The content of the element includes a required `<Name/>` element and an optional `<Description/>` element. The optional `<Device/>` element may be repeated and must be used after the other elements (,).

C.3. XML Models

The notation of the XML model is similar to that of the schema model:

- Required attributes and elements are shown with their name in bold text.
- Optional attribute and element names are shown as plain text.
- Attribute value choices are shown in quoted plain text, with vertical bars (|) separating the choices.
- Default attribute values are shown in bold text.
- Descriptions of attribute and element values are *italicized*.
- Alternative values are separated by a vertical bar (|).
- Elements that may be repeated are followed by a continuation character (⋮) or (...).
- Complex element content may be called out as a “See...” separate section.
- Inherited element content may be called out in heavy brackets (⟦ and ⟧)
- Enumerated call-outs provide additional information.

```
<ns:example-element
  id = "uuid" ❶
  activate = "yes|no"> ❷
  <Name>Element Name</Name> ❸
  <Description>Element Description</Description> ❹
  <Device>See Section C.3, “XML Models”.</Device>
  ⋮ ❺
</Package>
```

❶ **`id`**

Italicized text is descriptive.

❷ `activate`

Vertical bars indicate choices.

❸❹ **`Name`**, `Description`

Bold indicates required elements; non-bolded elements are optional.

❺ `Device`

The vertical continuation character indicates that the element may be repeated.

C.4. Keycap References

Please note that Macintosh and Windows keyboards use different names for the modifier keys:

Modifier Key Name Conventions

In this Manual	Macintosh Keyboard	Windows Keyboard
Alt	Option	Alt
Cmd	Cmd	Ctrl
Enter	Return	Enter
Shift	Shift	Shift

C.5. Typography

The following typographical conventions are used to help differentiate different word meanings:

Typographic Conventions

Typographic Style	Meaning
Normal text	No special meaning.
Emphasis	Important text.
Strong Emphasis	Very important text.
`Value to be typed`	A value to be typed into an input box using the keyboard.
`source code`	Source code
`computer text`	A line of source code, computer output, file name, or other code- or computer-oriented data.
Key name + **key name**	A command or shortcut key or key combination to be typed on the keyboard.
Control Label	The name of a control on the screen.
Menu → **Submenu** → **Command**	The “path” to select the specified command or control.
Document Reference	The title of a document or a heading within a document.
Document Reference	A selectable link to a document.

C.6. User Actions

The following conventions are used when referring to actions performed using the mouse, touch screen, or other pointing device:

Action Conventions

Verb item on which to act	**Action**
Select **item**	Point the mouse cursor at the item and then click and release the left (primary) mouse button; or point the trackpad cursor at the item and then click and release the trackpad; or touch the screen at the item and then tap the screen.
Select/De-select **item**	When referring to an item in a list, turn on the highlight (select) or turn off highlighting (deselect) by selecting the item. For some items you may multi-select by pressing a modifier key (Shift or Cmd) while selecting.
Select/De-select "*content*"	Select the text content by click-drag-releasing the left (primary) mouse button, sweeping the cursor over the desired text.
Drag **item**	Select the item without releasing the mouse button, then move the item to a new position or place, releasing the mouse button to drop the item.
Key name + Drag **item** (e.g. Ctrl+Drag item)	Hold down the named key, then drag the item.
Choose **Menu** → **Sub-menu** → **Command**	Select the named menu, then choose the sub-menu and/or command.

C.7. Tips and Admonitions

The following symbols are used to identify important information:

Symbol Conventions

Tip

A hint that may make things easier or help you be more productive.

Caution

Information that should not be ignored because recoverable data loss might occur.

Note

Supplemental information qualifying an important points or that applies to a special case.

Warning

Information that must not be ignored because irrecoverable data loss might occur.

Important

Information should not be ignored but will not cause data loss.

AVATO

999 Canada Pl Suite 404
Vancouver, BC V6C 3E2
1–604–600–7715
http://avato.co
info@avato.co