Unified CVP API Introduction

Unified CVP's API design has three goals: to be simple to use, to provide all the information a developer may need, and to allow the use of as many programming languages as possible. The API is used for simple tasks such as getting the ANI of the call or complex tasks such as creating a custom voice element. The API defines some mechanisms for custom components to integrate with Call Studio, though the API is primarily used to create components for integration with VXML Server.

Unified CVP provides a Java API, which is the most efficient way to interface with Unified CVP software. The Java API is also the most comprehensive; all components can be constructed using it. It is also the only way to build the components that require integration with Unified CVP Studio. Many components can be built with an equivalent API called the XML API. This API works by sending and receiving XML documents over an HTTP connection. This scheme allows the use of any programming language with the ability to create and parse XML and handle HTTP connections. Languages such as Perl or ASP in conjunction with a web server like Tomcat are sufficient to interface with VXML Server using the XML APIs. The requirement for integrating with Call Studio must be through Java, so the XML API is used to construct only those components that need to integrate with VXML Server. The table below lists each component, which API can be used to construct it, and whether that component must integrate with Call Studio.

Table 1: Unified CVP Components with Different APIs

Unified CVP Component	Build With Java API	Build With XML API	Call Studio Integration
On Call Start / On Call End Actions	Yes	Yes	No
Dynamic Element Configurations	Yes	Yes	No
Generic Action and Decision Elements	Yes	Yes	No
Hotevents	Yes	No	No
Say It Smart Plugins	Yes	No	Yes
On Application Start / End Actions	Yes	No	No
Loggers	Yes	No	No
On Error Notification	Yes	No	No

Java API

There are two parts of the Unified CVP Java API: a set of Java interfaces and classes that are implemented or extended to build a custom component and a set of classes used by those components to obtain information on the environment in which the call is occurring. Each component implements or extends a different class, though many of them share a common base. Similarly, the class used to obtain environment information differs for each component, though each of those classes shares a common base class.

The classes used to obtain and change environment information are referred to as the Session API. All components receive an instance of one of the classes to act as the conduit between the component and VXML Server. The classes in the Session API are organized into a hierarchy where the classes for each component add unique capabilities to the common base with regards to what data is available to it and what it is allowed to modify.

When building a component, the design requires the component to implement a single execution method VXML Server uses to access the component. This method can be seen as the "main" method for that component; it is where VXML Server leaves its context and enters the component's. It is this execution method that receives as a parameter a class belonging to the Session API to provide the component access to environment information.

The execution method is used exclusively by VXML Server. Two components, custom configurable elements and Say It Smart plugins, require integration with Builder for Call Studio. For those components, the API additionally requires methods that define how to render it.

For those components that need to produce VoiceXML (primarily configurable voice elements and hotevents), Unified CVP provides another set of Java API classes called the Voice Foundation Classes (VFCs). These classes act as an abstraction layer to VoiceXML and allow Unified CVP components to work seamlessly on any supported voice browser. Building VoiceXML using the VFCs is very much like building VoiceXML statically, except in a Java environment.

The API Javadocs contain detailed descriptions for each of the classes in the Java API, including the Session API and the VFCs.

Design Considerations

A few notes on VXML Server and how it interacts with custom components written in Java is warranted. This information is important to keep in mind since how a developer approaches the design of the components they wish to build is impacted by them:

- Each application is run by VXML Server in its own separate classloader. The classloader's focus includes all Java classes found in the local application's `java` folder, all classes found in VXML Server's `common` folder, and the other classes available in the application server's `CLASSPATH`. The advantage of this approach is that developers need only worry about class name conflicts within an individual application. One consequence, however, is that static class variables are static *only within each application*, even if they appear in classes stored in `common`.

  Additionally, when an application is updated, a new classloader is created for the application, replacing the previous one. This is not a problem unless dealing with static variables, which would be reset once the application is updated. While knowledge of classloaders is not required in order to know how to build custom components, it can be useful to understand how classloaders work in Java to understand how custom component code integrates with VXML Server.

- An application is loaded into memory when VXML Server first starts up or the application is updated. During this process, a *single instance* of each element (both standard and configurable) is created and cached for the application to use. Whenever a call to that application encounters an element, VXML Server will call the execution method of that single instance. This means that a single element instance will handle requests made across all calls to the application. This applies to multiple uses of an element type in the call flow (for example, if the call flow contains two Digits elements, VXML Server will actually use the same instance for both across all calls). This is very important because in this design, the element class acts as if it is *static*.

  The consequence of this is that all member variables defined in the element class act as static variables, meaning that once changed, every caller experiencing that element type is exposed to the same value. Use only static final member variables, store any persistent data in the session (which the API provides access to), and keep all other variables local to the execution method. Everything an element needs is provided by the API so while this is important to be aware of, this design restriction should not prevent the developer from implementing any desired functionality within the element.

- VXML Server runs in a multi-threaded environment. If the guidelines above are followed, such as avoiding member variables and non-final static variables this does not pose a problem. The developer does not need to worry about architecting their code with synchronization in mind when dealing with local or session variables. They would, however, when performing tasks such as accessing external resources such as files.

Compiling Custom Java Components

Once a component is constructed in Java, the process for compiling and deploying these classes is very simple. The VXML Server `lib` directory includes JAR files containing everything a developer requires to compile custom components. The main JAR file of interest is `framework.jar`, which defines the entire Unified CVP Java API (including the VFCs).

To create custom components, all that is needed is to ensure that this JAR file appears in the compiler's `CLASSPATH` or referred to in a Java IDE project file. Some Java IDEs may require additional JAR files, `servlet.jar` and `xalan.jar`, to appear in the `CLASSPATH` since the classes within `framework.jar` refer to classes defined in those JAR files and these IDEs cannot compile without definitions for these additional classes. The command-line Java compiler does not require `servlet.jar` or `xalan.jar` to appear in the `CLASSPATH`. The developer is then responsible for adding to the `CLASSPATH` any additional JAR files their custom code requires.

Deployment

Once compiled, component class files are deployed separately for Call Studio and VXML Server. Within these deployments, a developer can choose to associate component classes with a specific application or with the system as a whole so that the components can be shared across all applications. The deployment process for Call Studio and VXML Server are described in the following sections. A third section provides details on the specific deployment directories developers should use.

Call Studio Deployment

Call Studio provides a location to place components that are to be shared across all applications: `Call Studio\eclipse\plugins\com.audiumcorp.studio.library.common`. Individual classes are placed in the `classes` subdirectory and JAR file archives are placed in the `lib` subdirectory.

Note The only classes that need to be placed here are custom configurable elements and Say It Smart plugins since all other components have no Call Studio interface and hence require deployment only on VXML Server.

Once deployed in this folder, custom configurable elements will appear in the element pane directly under the *Elements* folder alongside with Unified CVP-provided elements.

For component classes that apply to a specific application only, a developer uses the `deploy\java` directory found in that application's project folder. Within Call Studio, compiled classes and/or JAR files can be dragged from outside Call Studio to the appropriate subdirectory in the `deploy\java` folder.

An alternative method that does not require Call Studio to be running would be to copy the files into the appropriate subdirectory of the `deploy\java` folder using the file system.

The application project folder can be found in `Call Studio\eclipse\workspace` (unless the developer sets the workspace to a custom directory) and the `deploy\java` folder within the application will appear here exactly as it appears in the Call Studio application project window. This can also be done while Call Studio is running, though to view the copied files in Call Studio, the `deploy` folder should be selected and the Refresh option chosen from the contextual menu.

Custom configurable elements placed in the `deploy\java` directory will appear in Call Studio's element pane under the folder named *Local Elements*. The call flow editor for that application must be closed and reopened in order for newly copied local elements to appear in the element pane.

When the application is deployed from within Call Studio, the VXML Server folder created for that application will contain a folder named `java` whose contents is identical to the `deploy\java` folder in the Call Studio project.

It is important to mention that you can have a Java library with *.zip filenames in your development environment (Call Studio), because classpaths are explicitly specified. However, the VXML Server runtime environment has to preload all library classes implicitly. The naming convention the VXML Server enforces the *.jar filename extension and will not load classes with *.zip filenames. Therefore you must use *.jar filenames for classes that are developed on Call Studio and will run on VXML Server.

VXML Server Deployment

When an application is deployed through Call Studio, a folder is created that encapsulates all the information for that application, including all Java code the developer placed in the Call Studio project as per the instructions given in the previous section. If the application is to change in any way, from changes to the call flow, to the addition or subtraction of required Java files, those changes must be done through Call Studio and then redeployed to VXML Server.

One deployment requirement that must be performed by the developer is to ensure that the Java components and utility libraries stored in the `Call Studio\eclipse\plugins\com.audiumcorp.studio.library.common` folder are also placed in the VXML Server `common` folder. When an application is deployed from Call Studio, only that application's files are created, any common code is not included. As a result, it is the developer's responsibility that the contents of the `common` folder in Call Studio also appear in the VXML Server common folder.

Note When VXML Server initializes, it first loads the classes in `common` and then loads each application's classes. Due to the way Java classloaders work, if a Java class appears in both the `common` folder and an application's `java` folder, the one in `common` will take precedence and the one in the application's `java` folder *will not be loaded*.

Note Due to the order in which these classes are loaded, the developer cannot place a class in `common` that refers to a class that only appears in an application's `java` folder since the classes in `common` are loaded first. Keep in mind that some application servers have advanced options to change this precedence to *parent-last*, meaning that the application-level classloaders take precedence. By default, all application servers should be configured to be *parent-first*.

Subdirectories of the Java Folder

The `java` folder of a Call Studio project and a VXML Server application folder contain two subdirectories named `application` and `util`. Each folder encapsulates Java classes used for different purposes, their distinctions applying primarily to how the application works within VXML Server.

The `application` folder should contain all Java code for components that are used by the application.

Note In Call Studio, any custom configurable elements that are utilized only by the application would be placed in this folder and will appear in Call Studio's element pane under the folder named *Local Elements*. The call flow editor for that application must be closed and reopened in order for newly copied local elements to appear in the element pane.

The second subdirectory of the `java` folder is the `util` folder. This is used for Java libraries that provide the application with utilities unaffiliated with Unified CVP (such as math libraries, XML parsing libraries, etc.).

Notes concerning which folder to use:

- Any class that refers to Unified CVP-specific API classes cannot be deployed in the `util` folder. If the class is application-specific, it must be placed in the `application` folder of that application. If the class is to be shared across all applications, it must be placed in the `common` folder of VXML Server.
- The classes in the `util` folder *will not* be reloaded when the application is updated using the `updateApp` administration script. If this behavior is not desired or the utility libraries are frequently updated, place these files in the `application` folder. See User Guide for Cisco Unified CVP VXML Server and Unified Call Studio for more information.
- Utility classes that do not refer to Unified CVP classes at all (such as third-party libraries) can be placed anywhere within the CLASSPATH of the application server. For example, on the Tomcat application server, a library for connecting to a mainframe system can be placed in `TOMCAT_HOME\lib` rather than any Unified CVP directory.

XML API

The philosophy behind the XML API is to provide as much of the functionality found in the Java API as possible in a way that can be accessed by non-Java developers. This is managed by using XML, which can be produced and parsed easily, and by using HTTP connections, which can be handled by many different programming languages. Interpreted languages such as Perl or PHP are just as effective in interfacing using this API as other languages such as ASP or CGI via C++.

The API works in a similar fashion as the Java API. VXML Server creates one or more XML documents and places their contents in POST arguments in an HTTP request. These documents contain the same environment

information available through the Java Session API classes. This request is sent to a developer-specified URL whose purpose is to produce an XML document and return it as the HTTP response. As with the Java API, the XML documents returned by various components differ to reflect the different functionality each component possesses. All these XML documents comprise the XML API. The DTDs for the XML API documents are found in the `dtds` directory in VXML Server. The DTDs exist as a reference, a DOCTYPE line is not required in either request or response XML documents. The format of each XML document will be described in detail in each component's section in this document.

A component using the Java API has the ability to access methods that interface with VXML Server whenever needed. The XML API, since it is executed over HTTP in a request / response fashion does not have that luxury. VXML Server does not know in advance what session information the component will need. Providing a separate interface for every piece of information desired would cause unnecessary overhead since a component could potentially access this information dozens of times, each time requiring a new HTTP request and response. To resolve this issue, VXML Server sends to the component *all* information in several XML documents passed as POST parameters. While this may seem like a lot to put into a single document, especially if the component does not need more than a few pieces, a typical application will not possess so much session information as to adversely affect the performance of the XML API.

Another consequence of the request/response mechanism for accessing the XML API is that while the Java API can call a method to read information and set information, the XML API must separate the read and write functionality. The HTTP request XML documents produced by VXML Server contains all desired information to read while the XML response sent back from the component specifies how to manipulate the desired information. Since the tasks each component can perform are different, the response XML document will differ for each component type.

While the XML API provides the same functionality as the Java API, there are some small deficiencies. Firstly, there is additional overhead involved in the XML API since it involves the creation and parsing of XML as well as the overhead inherent in HTTP communications. This overhead, though, is not large as the XML documents involved are typically small and only a single HTTP request and response are used per component. However the developer should test their system to ensure that any overhead introduced is acceptable. Secondly, some components, both Unified CVP and custom built, utilize Java classes as efficient mechanisms for storing data. Using the Java API, these classes can be accessed and modified directly. The XML API, however, will not be able to because it is a text-only interface designed to work identically using different programming languages. A developer must be aware of this restriction before designing components that rely on Java-only concepts.

Due to the nature of XML and HTTP and the complexity of some Unified CVP components, the XML API is available as an alternative only for some components.

The components that can utilize the XML API are:

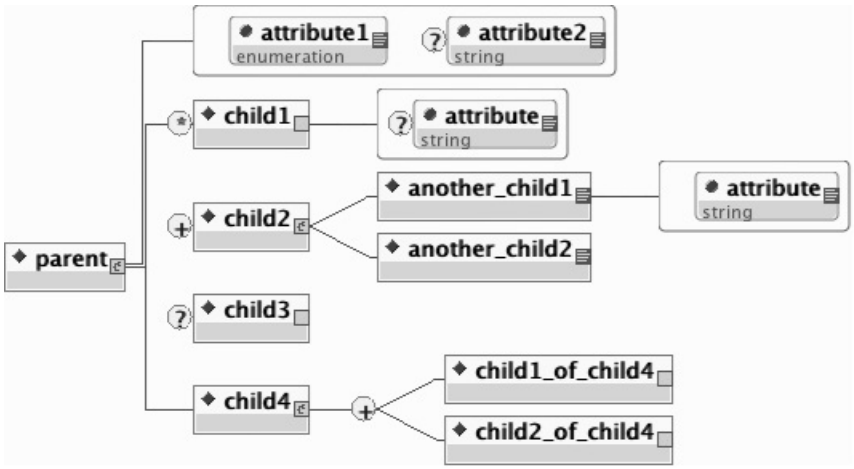
- standard action and decision elements
- dynamic element configurations
- start and end of call actions

Since configurable elements require more integration with Call Studio and VXML Server, they can only be created using the Java API.

DTD Diagrams

While the Java API has Javadocs explaining what can be done with the classes and methods in the API, the DTDs for every XML document sent either as a request or a response is described within this document. A quick introduction to DTD diagrams is warranted at this point in order to fully understand the XML API.

Figure 1: Sample DTD Diagram for XML



The preceding figure shows a sample DTD diagram that contains most of what can be found in an XML DTD. A DTD diagram is a graphical representation of a DTD, which explains how an XML document is formatted. Due to the nature of XML, syntax is very important and DTD diagrams describe the syntax of an XML document. The diagram shows the hierarchical structure of XML by using a sideways tree display, listing tags and their child tags from left to right. The diagram shows the attributes of tags as well as how those tags can be added to their parent tags.

The characteristics of the sample DTD diagram are:

- The root tag is named *parent*. A Tag is denoted by a blue diamond in its box.
- The *parent* tag has two attributes: *attribute1* and *attribute2*, displayed in a red box emerging from the tag. An attribute is denoted by a blue circle in its box.
    - The type of each attribute is shown in the *gray section* of the box (enumeration and string in this example).

      An enumerated type is an attribute whose value can be one of a select group of options (for example, “apple”, “orange” or “grape”).
    - In this example, *attribute2* is an *optional* attribute, denoted by a *question mark* to its left.
    - In this example, *attribute1* is a *required* attribute, denoted by having *no symbol* to its left.
- The *parent* tag has four child tags that can appear within the encapsulating parent tag. This is denoted by a red bracket encapsulating all the tags.
    - A bracket indicates that the child tags must appear within the parent tag in the order set in the diagram.
    - A "*" at the left edge of the tag indicates it can appear from 0 to many times.

- A "+" indicates it can appear from 1 to many times (that is, it must appear at least once in the document).
- A "?" indicates it can appear from 0 to 1 time (if it appears, it can only be once).

- The *child2* tag contains its own tags, *another_child1* and *another_child2*.

  This time, however, an angled red line is used to encapsulate the tags. This indicates that either one or the other can appear, but not both.

- The *child4* tag has a similar situation, but a + sign appears before its child tags, indicating that the child tags can appear any number of times, in any order, as long as there is at least one appearance.

The following code is an example XML document that conforms to the above DTD:

```
<parent attribute1=”something”>
   <child1 attribute=”a value”/>
   <child1/>
   <child2>
      <another_child1 attribute=”this is required”>
         Some value for the another_child1 tag.
      </another_child1>
      <another_child2>
         The content for another_child2.
      </another_child2>
   </child2>
   <child4>
      <child1_of_child4/>
      <child2_of_child4/>
      <child2_of_child4/>
      <child1_of_child4/>
   </child4>
</parent>
```

Note The *child1* tag is allowed to appear multiple times because it has a * next to it. One *child1* tag does not have an attribute because it is optional. Additionally, since these tags do not encompass any additional content, the tag is closed with a “/>” so there is no need for a closing tag.

Note The *child2* tag contains its two child tags. The *another_child1* and *another_child2* tags encapsulate text so they have an open and close tag. The *another_child1* tag must specify its required attribute.

Note The *child3* tag can be omitted because it is optional.

Note The *child4* tag contains any number of its child tags in any order. It would be a syntax error to not include any child tag at all since at least one is required.

Note The order in which the child tags of parent must conform to the diagram. If *child4*, for example, appeared before the *child1* tag, that would be a syntax error.

Deployment

It is up to the developer to set up the environment necessary to support the requests made by VXML Server. Since the API is accessed over HTTP, the XML content must be served by a web or application server. This content could potentially be served on the same machine as VXML Server though the act of parsing XML and the details of using HTTP will add additional overhead to the performance of VXML Server. To get the best performance, the setup should consist of a separate system handing the requests on the same subnet as the machine on which VXML Server is installed. Maintaining the two systems on the same subnet will reduce any network overhead as well as allow the administrator to restrict communication to occur only within the same subnet.

Keep in mind that, unlike Java, changes made to components using the XML API are not graceful. Components using the Java API are deployed on top of VXML Server and therefore take advantage of the graceful nature of VXML Server administration activity, while components using the XML API are hosted on separate systems. Any changes made to system that would affect the XML sent as response to VXML Server requests would be available immediately. The administrator must ensure that maintenance activity be performed on both systems to ensure callers do not experience changes within a single phone call. For handling updates to applications using components using the XML API, suspend the application before changes are made to the server hosting the XML.