# Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio

**Release 6.0(1)**

November 2007

**Corporate Headquarters**
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-4100

# Chapter 1: Introduction

Universal Edition software has been designed to be easy to use but highly extendable. While the software provides enough to produce high quality voice applications out of the box, many users will want to extend the functionality of the software by building custom components that perform very specific tasks. This document describes the processes and application programming interfaces (APIs) provided for a developer to construct and deploy these components.

The components a developer uses the APIs to construct are: configurable action, decision, and voice elements, standard action and decision elements, dynamic element configurations, start and end of call actions, start and end of application actions, the on error notification, hotevents, Say It Smart plugins, and global and application loggers.

## Requirements

All components require programming effort to construct. In order to build these components, Universal Edition provides a Java API as well as an API that allows the use of other programming languages. Therefore, the reader should at least possess a familiarity with programming concepts.

Some components can only be constructed using the Java API and so for these components, the reader should possess a familiarity with the Java programming language. The reader should understand Java interfaces and abstract classes, extending classes and overriding methods, static variables and methods, Java collections, and compiling and deploying Java classes and JAR files. Universal Edition does not require very complex Java coding, knowing the basics of the Java language will be sufficient to start working with the Universal Edition Java API. Most of the information about the Java API is encapsulated in the API's Javadocs. This document will serve to provide a starting point and the Javadocs will provide the details.

Some components are used to produce VoiceXML, the language used to communicate with a voice browser. When building these components, a familiarity with VoiceXML is essential. The Java API used to produce VoiceXML follows the same design as the VoiceXML language, so a developer that understands how to write VoiceXML will be able to produce it using the API much easier and faster than a developer not familiar with VoiceXML.

Finally, the reader should familiarize themselves with the way Cisco Unified Call Services, Universal Edition (Call Services) works by reading the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio, especially Chapter 2, which explains the purpose for each of the components described in this document. Familiarity with Cisco Unified Call Studio (Call Studio) and Builder for Call Studio is also recommended for understanding how to construct configurable elements and Say It Smart plugins since they define how they are displayed within the Builder.

# Chapter 2: Universal Edition API Introduction

Universal Edition'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 Call Services.

Universal Edition provides a Java API, which is the most efficient way to interface with Universal Edition 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 Universal Edition 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 for 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 Apache are sufficient to interface with Call Services 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 Call Services. The table below lists each component, which API can be used to construct it, and whether that component must integrate with Call Studio.

| Universal Edition 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 Universal Edition 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 share 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 Call Services. 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 Call Services uses to access the component. This method can be seen as the "main" method for that component; it is where Call Services 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 Call Services. 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), Universal Edition provides another set of Java API classes called the Voice Foundation Classes (VFCs). These classes act as an abstraction layer to VoiceXML and allow Universal Edition 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 Call Services 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 Call Services 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 Call Services' `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 Call Services.

- An application is loaded into memory when Call Services 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, Call Services 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 (e.g. if the call flow contains two Digits elements, Call Services 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. It is highly recommended to 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.
- Call Services 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 Call Services `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 Universal Edition 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 Call Services. 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 Call Services 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_5.1.0`. Individual classes are placed in the `classes` subdirectory and JAR file archives are placed in the `lib` subdirectory. Note that 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 Call Services. Once deployed in this folder, custom configurable elements will appear in the element pane directly under the "Elements" folder alongside with Universal Edition-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 Call Services 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.

*Call Services 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 Call Services.

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_5.1.0` folder are also placed in the Call Services `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 Call Services `common` folder.

Note that when Call Services 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. Also note that 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 Call Services 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 Call Services.

The `application` folder should contain all Java code for components that are used by the application. Note that 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 Universal Edition (such as math libraries, XML parsing libraries, etc.).

There are several notes that must be made concerning which folder to use:

- Any class that refers to Universal Edition-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 Call Services.
- 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 the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio, Chapter 3 in the section describing the update capabilities for more information.
- Utility classes that do *not* refer to Universal Edition classes at all (such as third-party libraries) can be placed anywhere within the `CLASSPATH` of the application server. For example, on the Apache Jakarta Tomcat application server, a library for connecting to a

mainframe system can be placed in `TOMCAT_HOME/common/lib` rather than any Universal Edition 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. Call Services 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 Call Services. 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 Call Services whenever needed. The XML API, since it is executed over HTTP in a request / response fashion does not have that luxury. Call Services 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, Call Services sends to the component *all* information in several XML documents passed as a 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 Call Services 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 Universal Edition 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 Universal Edition 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, and start and end of call actions. Since configurable elements require more integration with Call Studio and Call Services, 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.

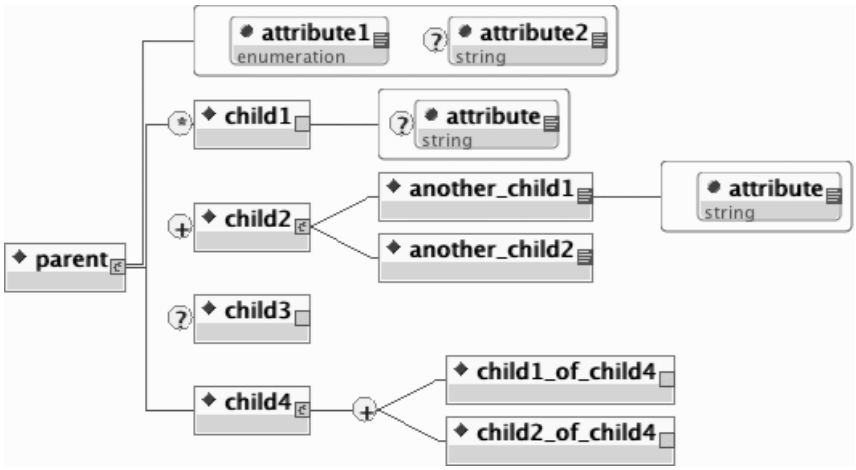


*Figure 2-1*

Figure 2-1 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, that 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.

In the above diagram, the root tag is named *parent*. Tags are denoted by a blue diamond in its box. *parent* 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 those attributes are shown in the gray section of the box. An enumerated type is an attribute whose value can be one of a select group of options (for example, “apple”, “orange” or “grape”). *attribute2* is an optional attribute, denoted by a question mark to its left. *attribute1* is a required attribute, denoted by having no symbol to its left.

*parent* 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 * next to the tag indicates it can appear from 0 to many times. A + indicates it can appear from 1 to many times (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 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. *child4* has a similar situation, but a + sign appears, indicating that the child tags can appear any number of times in any order, as long as there is at least one.

The following 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>
```

Notes:

- *child1* 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.

- *child2* contains its two child tags. The *another_child1* and *another_child2* tags encapsulate text so they have an open and close tag. *another_child1* must specify its required attribute.
- *child3* can be omitted because it is optional.
- *child4* contains any number of its child tags in any order. It would be a syntax error to not include any child tags at all since at least one is required.
- 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 Call Services. 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 Call Services though the act of parsing XML and the details of using HTTP will add additional overhead to the performance of Call Services. 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 Call Services 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 Call Services and therefore take advantage of the graceful nature of Call Services 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 Call Services 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. A recommended method for handling updates to applications using components using the XML API is to suspend the application before changes are made to the server hosting the XML.

# Chapter 3: Session API

As described in the previous chapter, Universal Edition provides a mechanism for the developer to access and change information having to do with the phone call or the session. Through this API one can get environment information such as the phone number of the caller (ANI), the time the call began, and application settings such as the default audio path. This API is also the conduit for the developer to set element or session data, send custom logging events, or access the user management system. A subset of the Session API, called the Global API provides access to data that exists beyond individual sessions.

Any custom component built by the developer will be sent this API to interface with the session. This section of the document describes the API and what it can be used for. Both the Java and XML versions of this API are described. Subsequent chapters will detail the APIs used to actually construct components.

## Java API

As described previously, every Universal Edition component is constructed by implementing a Java interface or extending a Java class and overriding a single execution method. One argument to this method is a Universal Edition-specified Java class that acts as the API to the session. Methods in this class are used to get or change information stored in the session, such as element or session data.

A different API class is used depending on the component. All API classes are derived from the base class `APIBase`, though all non-logger API classes directly extend `ComponentAPI` (both are found in the `com.audium.server.session` package). `APIBase` defines information retrieval functions any component accessed within a call session can use, such as:

- Obtaining telephony information such as the ANI and DNIS.
- Obtaining application setting data such as the gateway adapter name, default audio path, maintainer, etc.
- Getting element or session data created by components run prior to the current component.
- Retrieving a list of elements and the exit states encountered by the caller prior to the current component.
- Obtaining information on where the current application resides in order to aid in the loading of custom content found there.

`ComponentAPI` adds to this the ability to alter some environment settings:

- Getting access to the User Management system, allowing the component to create, modify, or query information on users.

- Creating session data. This class does not allow the creation of element data because only elements can do so (the start of call class cannot, for example).
- Adding custom content to the activity log and warnings to the error log.
- Triggering custom logging events and warning events that are picked up by loggers.
- Setting the maintainer, default audio path, application language and encoding, as well as the call's session timeout. At any point in the application, these settings can be changed.
- Accessing the Global API to get and set application and global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on application and global data).

The following lists the API classes that are used for various components (also found in the `com.audium.server.session` package). A detailed description of what each class provides is given in the individual section for that component.

- **CallStartAPI**. This class is sent as an argument to the start of call class.
- **CallEndAPI**. This class is sent as an argument to the end of call class.
- **ElementAPI**. This class is used by all standard and configurable element classes as well as dynamic configuration classes. The following classes extend `ElementAPI` to provide additional functionality required for different kinds of elements.
  - **ActionAPI / ActionElementData**. The `ActionAPI` class is used by generic action element classes and is extended by `ActionElementData` which is used by configurable action element classes.
  - **DecisionElementData**. This class is used by configurable decision element classes.
  - **VoiceElementData**. This class is used by configurable voice element classes.
- **LoggerAPI**. This class is sent as an argument to a logger's execution method for handling a session-specific logging event.

# XML API

When a component uses the Java API, the Session API is accessed via an object passed to the execution method. A similar setup exist when a component uses the XML API. The entire contents of the Session API is made available via a set of XML documents passed to the component in the HTTP request. Each component will receive this information whether they need it or not, since Call Services does not know in advance what information the component could require. The component can choose to ignore these documents if the information contained within are not required, or use a fast, event-based parser to extract only the desired information from the documents.

Each component receives two POST arguments containing complete XML documents representing the Session API. The first, named "inputs", lists the session information

representing the state of the application up to the point when the component was reached. The second argument, named “settings”, lists the current value for the application settings.

*XML Document Sent in “inputs”*

Figure 3-1 shows the DTD diagram for the XML document sent to all components in the “inputs” argument. Its DTD is defined in the file `ElementRequest.dtd` in the Call Services `dtds` folder.

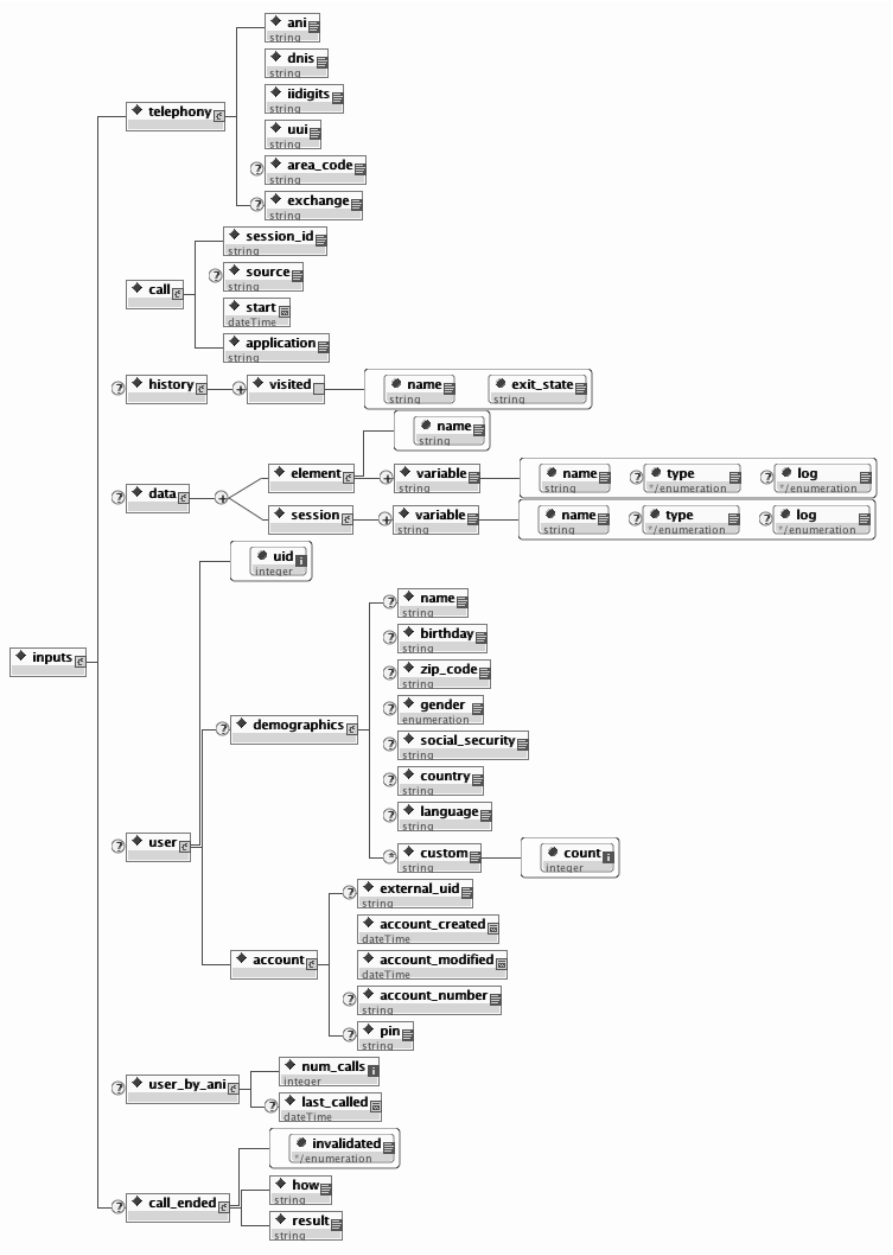


*Figure 3-1*

The tags in this XML document are:

- **telephony** – This tag holds information about the call itself such as the ANI. It also contains the area code and exchange of the ANI. All values are "NA" if not sent by the voice browser (area code and exchange won't appear at all in the case that the ANI is not sent).
- **call** – This tag holds call information. The session ID is used by Call Services to identify the call. The <source> tag, if applicable, contains the name of the application that transferred to this one (the tag does not appear if application visit is a new call). <start> contains when the call started. <application> contains the name of the application.
- **history** – This tag holds the history of elements visited so far in the call. The name and exit state of the element is included as attributes to a <visited> tag. Multiple tags are listed in the order in which the elements were visited in the call. The <history> tag will not appear if no elements were visited before this one (i.e. the start of the call).
- **data** – This tag holds all the element and session data created so far in the call. The <element> tag's name attribute holds the name of the element. All the variables created by this element appear in this tag. The log attribute indicates whether this variable's value will appear in the activity log file (no session variables appear in the log). The <data> tag will not appear if no element or session data exist. If the session data variable holds a Java class, the tag will contain the results of the toString() method called on that object.
- **user** – This tag appears only if the application is configured to use the user management system and the call has been associated with a particular UID. The <demographics> tag holds the user demographic information. The <account> tag contains information about the account such as when it was created and modified, the account number and pin (if applicable), etc. This data appears exactly as in the user database. See Chapter 4: User Management in the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more on user management.
- **user_by_ani** – This tag appears only if the application is configured to use the user management system, though unlike <user>, the tag will appear even if the call is not associated with a UID. The tag holds information about the number of calls made to this application by the current phone number and the last time a call was received to the application by that number.
- **call_ended** – This tag appears *only* when being sent as a request to an end of call event. It defines how the call ended and the result of the call. The possible content of <how> are: *hangup* (the caller hung up), *disconnect* (the application hung up on the caller), *application_transfer* (the application visit ended by transferring to another application), *call_transfer* (a blind transfer took place) and *app_session_complete* (the application visit ended even if the call itself continued - such as via a CTI event). The possible content of <result> are *normal*, *max_ports* (the caller hung up while on hold waiting to enter the application), *suspended* (the caller called into a suspended application), *error* (an error occurred during the call), *timeout* (the session timed out) and *invalidated* (the session was

invalidated by an element). Note that the `invalidated` attribute of `<call_ended>` also indicates if the session was invalidated by an element.

*XML Document Sent in "settings"*

Figure 3-2 shows the DTD diagram for the XML document sent to all components in the "settings" argument. Its DTD is defined in the file `Settings.dtd` in the Call Services `dtds` folder. Note that this document shares the same DTD as the static application settings file settings.xml created when an application is deployed from Call Studio to Call Services. This document is simply an XML representation of the application settings in Call Studio's project preferences for the application.

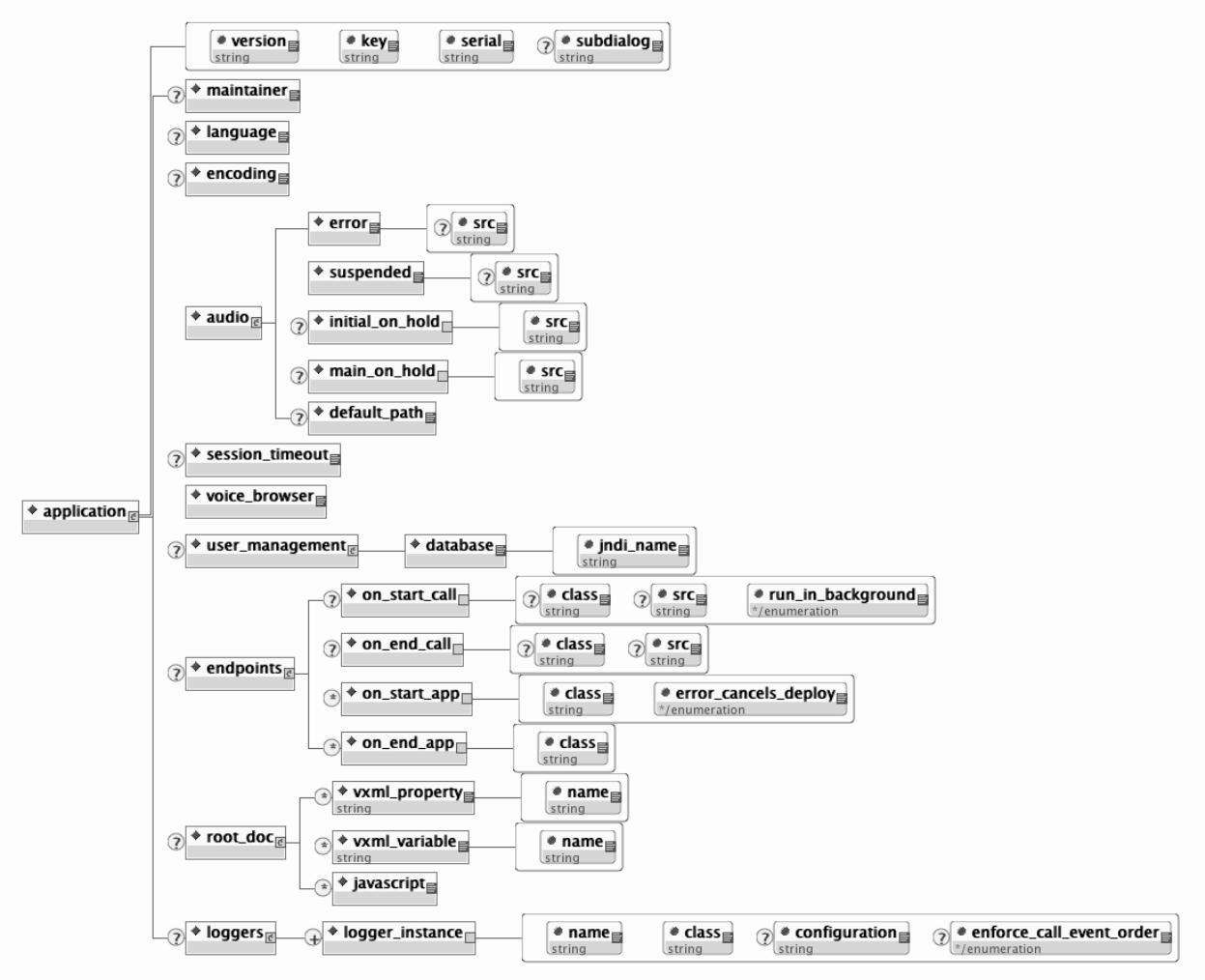


*Figure 3-2*

The tags in this XML document are:

- **application** – The root tag. The `key` and `serial` attributes are used by Call Studio and Call Services and can be safely ignored here. The `version` attribute holds the version of the file. The above diagram represents version 1.2 of the DTD. The `subdialog` attribute holds a `true` if the application is accessed as a subdialog, `false` if not.
- **maintainer** – This tag holds the e-mail address of the maintainer.
- **language** – This tag holds the language for the application. This value shows up in the VoiceXML pages produced by Call Services. The contents of this tag is formatted according to the specification for using languages in VoiceXML (e.g. "en-US").
- **encoding** – This tag holds the encoding format for the application. This value determines how the VoiceXML pages produced by Call Services are encoded. The contents of this tag is formatted according to the specification for encoding XML pages (e.g. "UTF-8").
- **audio** – This tag holds all audio files and/or TTS phrases to use for various situations.
  - **error** – This tag encapsulates the message to play when an error occurs and the application does not contain an error element.
  - **suspended** – This tag encapsulates the message to play when a caller calls an application that is suspended.
  - **initial_on_hold** – This tag encapsulates the first message a caller hears when all the Call Services ports are in use.
  - **main_on_hold** – This tag encapsulates the message repeatedly played after the initial on hold audio is played until a Call Services port is available.
  - **default_path** – This tag lists the URI path in which all pre-recorded audio for this application is located.
- **session_timeout** – This tag lists the length of time in minutes of inactivity Call Services will time out the call session.
- **voice_browser** – This tag lists the voice browser selected for this application. Note that the real name of the voice browser is used here, *not* the display name. Gateway Adapter real names can be seen by reading the folder name for that adapter in the `gateways` folder of Call Services.
- **user_management** – This tag encapsulates information concerning the user management database. The `<database>` tag's `jndi_name` attribute contains the JNDI name for the database and the tag itself contains the database to use, which is either "MySQL" or "SQLServer".
- **endpoints** – This tag encapsulates actions to perform at the endpoints of an application and a call.
  - **on_start_call**. The Java class or URI to call when a call to the application starts. The `class` attribute lists the full class name of the Java class. The `src` attribute contains the URI to call. The two attributes are mutually exclusive. The attribute `run_in_background`

is "true" if the Java class or URI is to be accessed in a separate thread by Call Services and "false" if the call is to wait for the action to complete.

- **on_end_call**. The Java class or URI to call when a call to the application ends. The `class` attribute lists the full class name of the Java class. The `src` attribute contains the URI to call. The two attributes are mutually exclusive.
- **on_start_app**. The Java class(es) to call when the application is loaded or updated. The `class` attribute lists the full class name of the Java class. The tag can appear multiple times, denoting multiple classes to run at the start of an application. The classes will be run in the order in which they appear in the document. The `error_cancels_deploy` attribute is set to "true" when an error in the class should cause the application loading to fail.
- **on_end_app**. The Java class(es) to call when the application is taken down (either as a result of the application server shutting down or due to an update). The `class` attribute lists the full class name of the Java class. The tag can appear multiple times, denoting multiple classes to run at the end of an application. The classes will be run in the order in which they appear in the document.

- **root_doc** – This tag contains tags representing the additions made to the application's root document. This tag will not appear if no language, encoding, properties or variables are added to the root document. The possible additions are:
  - **vxml_property** – This tag can appear any number of times listing the VoiceXML properties to add to the root document. Typically most voice browsers will take a property set here as an indication that it apply to the entire application. The `name` attribute lists the name of the property and the tag itself encapsulates the value.
  - **vxml_variable** – This tag can appear any number of times listing the VoiceXML variables to add to the root document. A variable set in the root document is available to all VoiceXML pages in the application. The `name` attribute lists the name of the variable and the tag itself encapsulates the value.
  - **javascript** – This tag encapsulates a Javascript function to place in the application's root document. Any number of these tags can appear to add multiple Javascript functions to the root document.
- **loggers** – This tag contains one or more `<logger_instance>` tags defining the loggers that are to listen to the events for calls for this application. The name attribute defines the logger instance name (all logger instances must have unique names). The class attribute defines the full Java class name of the logger to use. The optional configuration tag points to a configuration file for the logger instance. When the optional attribute enforce_call_event_order is "true" Call Services will ensure that the logger receive the logger events for a call in the order in which they occurred in the call.

Again, these two documents are sent as HTTP POST arguments to any URL using the XML API. The documents mirror all the functionality provided in the Java API to obtain session

information. Changing session information, such as setting session data, is done in the response XML document. Since each component has a separate response document, they are described in each component's individual chapter.

# Chapter 4: Call Start Action

Call Services can be configured to run code when a call has been received before the call flow is visited. The call start action can be implemented with either the Java API or the XML API. The call start action is a good way to create session data to be used by the rest of the application. There are two situations where session data may already exist:

- If the voice browser passed additional arguments to Call Services when the call was first received, these additional arguments are added as session data with the arguments' name/value pairs translated to the session data name and value (both as `String`s).
- If a separate Universal Edition voice application transferred to the current application, the application designer may have chosen to transfer element and session data to the destination application. This data will be converted to session data in the destination application.

The call start action is also given the ability to change the voice browser and any root document-affecting settings for the call. These changes apply to the current call only, and allows for a truly dynamic application. By allowing the voice browser to change, the application can be deployed on multiple voice browsers at once and use a simple DNIS check to output VoiceXML to the appropriate browser. Changing root document settings such as properties and language allow the call start action to control how the application appears to the caller using information it knows only at call time. Note that these changes can only be made by the call start action because it runs before Call Services has returned the first VoiceXML page and therefore can make changes that affect the outputted VoiceXML. Aside from these settings, the call start action can also change the maintainer and default audio path, though any component run within the call can do this as well.

The start of call event can be run in the background by checking the appropriate checkbox in the Call Studio application settings. If this is not done, the caller will hear silence until the call start action is complete and the call flow reaches the first VoiceXML-producing element. Answering the phone with too much silence could cause the caller to hang-up, thinking something went wrong. Latency issues are not as big a concern later in the application because audio can be played while action is executing or the application could make the caller aware that some potentially lengthy action is about to occur. Running the call start action in the background will ensure that the call flow will begin immediately.

Some notes of caution are warranted when running the call start action in the background. Firstly, ensure that elements in the call flow that attempt to access data created by the call start action do not try too quickly since it is possible the data has not been created yet. Since the call start action is run in a separate thread, there is no guarantee it will complete before the data it creates is required. The application can be architected to handle this by checking if the data exists before accessing it and if not, make the caller wait until it is created. Secondly, any errors that occur during execution of this action are placed in the error log but do not end the call (unless the application cannot run without performing the tasks in the call start action).

# Using the Java API

The call start action is built in Java by implementing the Universal Edition class `StartCallInterface` found in the `com.audium.server.proxy` package. It contains a single method named `onStartCall` that is the execution method for the call start class. This method receives a single argument, an instance of `CallStartAPI`. This class belongs to the Session API and is used to access and modify session information such as session data (See Chapter 3: Session API for more on this API). The method does not have a return value. It is expected that should an unrecoverable error occur, the call start action will throw an `AudiumException`.

# Using the XML API

As described in Chapter 3: Session API, the standard "inputs" and "settings" XML documents are sent via POST to the call start URI. Figure 4-1 shows the DTD diagram of the XML document that must be sent in response. The DTD for the start of call action response is defined in the file `CallStartResponse.dtd` found in the Call Services `dtds` folder.

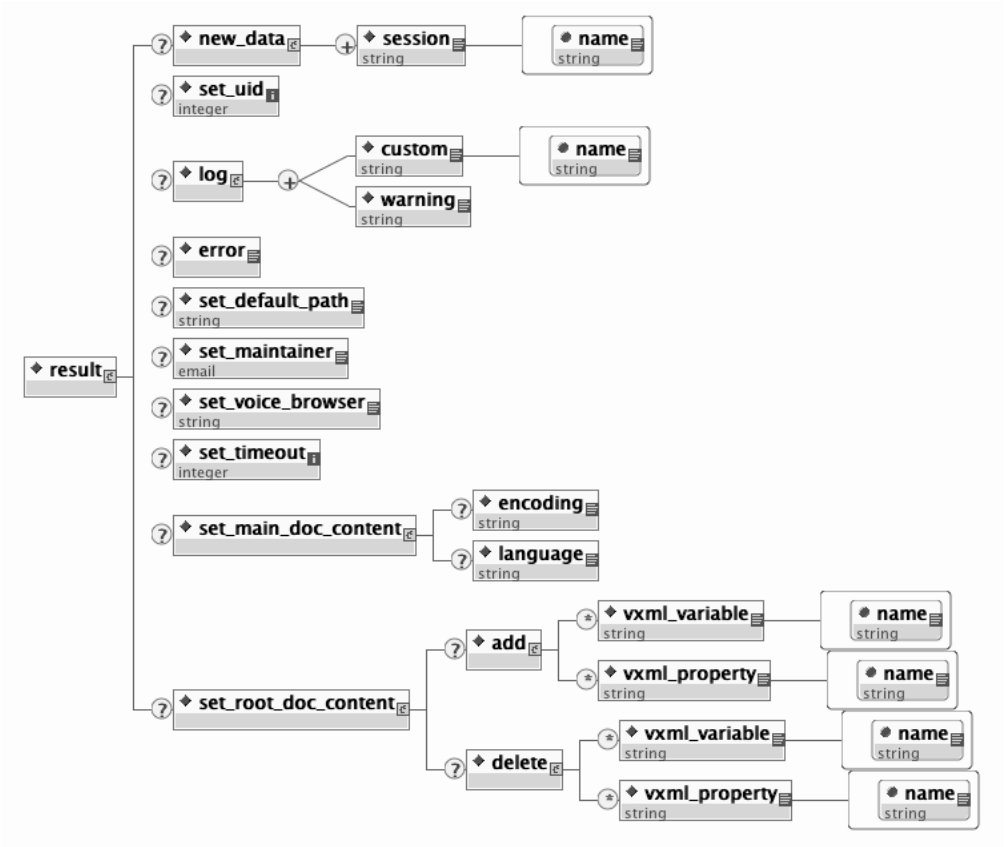


*Figure 4-1*

The tags in these XML documents are:

- **new_data** – This tag holds the session data to be created. Any number of `<session>` tags can appear, one for each session data variable to be created. Note that element data cannot be created because the call start action is not an element.
- **set_uid** – This tag is used to associate the call to a UID in the user management system. The content of the tag should be the integer UID.
- **log** – This tag is used to trigger logger events for this application. Any number of `<custom>` tags can appear, denoting the triggering of a custom event. The `name` attribute holds the name of the data, and the `<custom>` tag encapsulates the value. Any number of `<warning>` tags can appear, denoting the triggering of a warning event. The `<warning>` tag encapsulates the warning message.
- **error** – This tag reports to Call Services that an error occurred while executing the call start action. Call Services will then throw an exception whose message is contained in the `<error>` tag. This allows the XML API to throw exceptions just as the Java API does.
- **set_default_path** – This tag is used to change the default audio path.
- **set_maintainer** – This tag is used to change the maintainer e-mail address.
- **set_voice_browser** – This tag is used to change the voice browser for this particular call. Note that the real name of the voice browser must be used here, *not* the display name. Gateway Adapter real names can be seen by reading the folder name for that adapter in the `gateways` folder of Call Services.
- **set_timeout** – This tag allows the timeout length set for this session to be changed. The contents of the tab must be an integer representing the number of minutes in the timeout.
- **set_main_doc_content** – This tag allows the encoding and language settings for the application to be changed for this call. The `<language>` tag content is formatted according to the specification for using languages in VoiceXML (e.g. “en-US”). The `<encoding>` tag content is formatted according to the specification for encoding XML pages (e.g. “UTF-8”).
- **set_root_doc_content** – This tag allows for the addition and removal of VoiceXML properties and variables. The `<delete>` tag is necessary only if properties or variables were set in the application’s project pane in Call Studio and due to runtime circumstances the call start action determines they are no longer needed. The `name` attribute specifies the property or variable name and the tag contents encapsulates the value.

Note that all the tags are optional, there is no tag required except for the root `<result>` tag. Since the XML API requires a document in response, it is acceptable to return an XML document whose `<result>` tag is empty.

# Chapter 5: Call End Action

Call Services can be configured to run code once a call has ended. Unlike the call start action, the call end action can occur at any time in the call and there are several different situations that would trigger the call end action. The following lists those situations:

- The caller hangs up normally.
- The application hangs up on the caller. This includes any errors that are caught by the system that yield a hang-up, or places in the application when the call's purpose is over.
- A blind telephony transfer takes place. Blind transfers connect the caller with the party called using telephony switching equipment, removing the voice browser (and hence Call Services) from the calling context. Even though the physical phone call continues, the role of the automated system ends and so for it, the call has ended. Note that the availability of blind transfers is determined by the voice browser's functionality and network setup.
- The application performs a transfer to another Universal Edition application. This is not a telephony transfer, but the results are very similar. Since the call leaves the source application, it is considered the end of the "call" to that application.
- Call Services times out a session. This occurs only rarely, it would be seen only when some error prevented Call Services from receiving a request in the middle of a call and it waited a certain amount of time before timing out the session. This could be due to a voice browser going down or if the request coming from the voice browser is malformed and Call Services cannot determine which call that request was supposed to be for.
- The session is invalidated by a custom element. Standard and configurable elements have the ability to invalidate the session for situations where some process ends the call that would not prompt Call Services to be notified that the call ended. This functionality is described in more detail in the chapters on custom elements.

The call end action can be implemented with either the Java API or the XML API. Unlike the call start action using the XML API, the call end action does not have an option to perform it in the background. In fact, one need not worry about performing time consuming tasks in the call end action because it will not affect the performance of the call since it has ended. One must still be careful not to perform tasks that maximize CPU usage since that would aversely affect the handling of other calls.

Like the call start action, the call end action can modify the session such as creating session data or changing the default audio path, though these actions would not make sense as there is no more call flow to visit. The call end action can access everything that occurred within the call, including how the call ended (hangup, call transfer, etc.) This is useful for activities such as creating CDR records which must list everything a caller did.

A unique feature of the call end action is to optionally send back a final VoiceXML page to the voice browser. Some voice browsers will actually interpret a VoiceXML page sent back in response to a request triggered by a disconnect or hang-up event. Since the caller is no longer interacting with the IVR, this page would obviously only be useful for limited functionality that had nothing to do with interacting with the caller, such as executing `<log>` tags. Note that this final page applies only to when the caller hangs up on the application or the application hangs up on the caller.

## Using the Java API

The end of call action is built in Java by implementing the Universal Edition class `EndCallInterface` found in the `com.audium.server.proxy` package. It contains a single method named `onEndCall` that is the execution method for the call end class. This method receives a single argument, an instance of `CallEndAPI`. This class belongs to the Session API and is used to access session information such as session data (See Chapter 3: Session API for more on this API). The method does not have a return value. It is expected that should an unrecoverable error occur, the call end action will throw an `AudiumException`.

If the call end action is to return a final VoiceXML page to the voice browser, this is done by using the Voice Foundation Classes (VFCs) (See Appendix A: The Voice Foundation Classes for more on the VFCs) and accessing methods in the `CallEndAPI` Session API class.

## Using the XML API

As described in Chapter 3: Session API, the standard “inputs” and “settings” XML documents are sent via POST to the call start URI. Figure 5-1 shows the DTD diagram of the XML document that must be sent in response. The DTD for the end of call action response is defined in the file `CallEndResponse.dtd` found in the Call Services `dtds` folder.

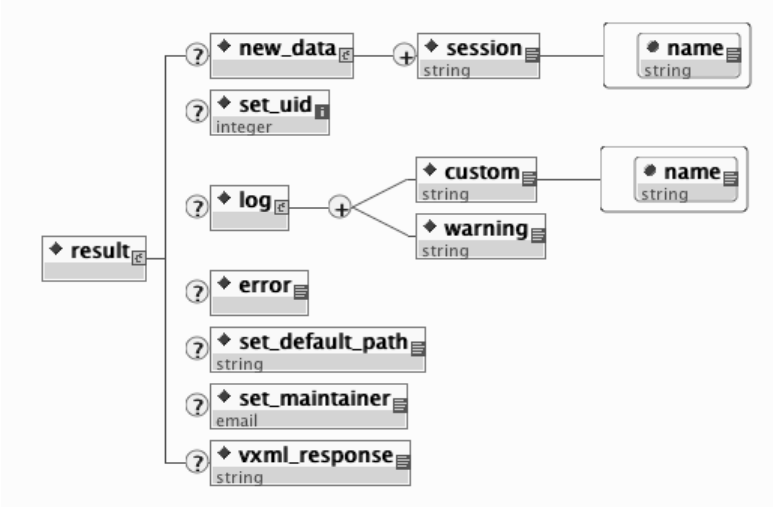


*Figure 5-1*

The tags in these XML documents are:

- **new_data** – This tag holds the session data to be created. Any number of `<session>` tags can appear, one for each session data variable to be created. Note that element data cannot be created because the call end action is not an element.
- **set_uid** – This tag is used to associate the call to a UID in the user management system. The content of the tag should be the integer UID.
- **log** – This tag is used to trigger logger events for this application. Any number of `<custom>` tags can appear, denoting the triggering of a custom event. The `name` attribute holds the name of the data, and the `<custom>` tag encapsulates the value. Any number of `<warning>` tags can appear, denoting the triggering of a warning event. The `<warning>` tag encapsulates the warning message.
- **error** – This tag reports to Call Services that an error occurred while executing the call end action. Call Services will then throw an exception whose message is contained in the `<error>` tag. This allows the XML API to throw exceptions just as the Java API does. Note that since the call has ended, there would be no adverse affect to the call itself, though an error event will be thrown.
- **set_default_path** – This tag is used to change the default audio path.
- **set_maintainer** – This tag is used to change the maintainer e-mail address.
- **vxml_response** – This optional tag encapsulates the VoiceXML page that is to be passed to the voice browser for the final response. It is expected to contain a CDATA tag that encapsulates the entire VoiceXML document as it is to be returned to the voice browser (including the first line starting with `<?xml`).The developer is responsible for ensuring the VoiceXML is correct as Call Services does no validation of the VoiceXML before returning it to the browser. Note that since the VFCs are not used to generate the VoiceXML like the Java API, the developer is responsible for ensuring the VoiceXML is compatible with the voice browser(s) being deployed to.

Note that all the tags are optional, there is no tag required except for the root `<result>` tag. Since the XML API requires a document in response, it is acceptable to return an XML document whose `<result>` tag is empty.

# Chapter 6: Dynamic Element Configurations

Configurable voice, action, and decision elements used in an application must have configurations. Usually, the configuration will be fixed, i.e., it acts the same every time a caller visits it. In this case, the configuration itself exists as an XML file stored on the system. Builder for Call Studio creates this file when the application is deployed. Programming is required when a dynamic element configuration is desired, i.e., one which is generated at runtime each time a caller visits it.

The manner in which configurations are used warrant closer examination. Configurations are used by pre-built elements in order to tell it how to function. Since configurable elements are constructed with Java, the configuration for the element must be given to it in the form of a Java class. The API provides a set of Java classes that encapsulate an entire element configuration, which are nothing more than just Java expressions of the visual Builder for Call Studio's Configuration Pane: three tabs *General*, *Settings*, and *Data* for action and decision elements and a fourth tab, *Audio*, for voice elements. When a static configuration is used, this information is stored as an XML file generated by Builder for Call Studio. Call Services converts this XML file to one of the Java configuration classes and then passed it on to the element.

A dynamic configuration is simply a way of adding an additional step in this process. Once Call Services loads the static representation of the configuration (known as the base configuration), it will pass this to the dynamic configuration Java class or URI for modification instead of passing it directly to the element. The class or URI adds to or changes the base configuration depending on the application business logic and returns a complete configuration. Call Services then passes this new configuration to the element.

## Using the Java API

Dynamic voice, action, and decision element configurations are constructed in the Java API by implementing the Java interfaces `VoiceElementInterface`, `ActionConfigInterface` and `DecisionConfigInterface` respectively, all found in the `com.audium.server.proxy` package. Note that the name of the voice element interface is not consistent with the others due to backwards compatibility concerns. Each of these interfaces contains a single method named `getConfig` that receives three arguments:

- The name of the element as a `String`.
- An instance of `ElementAPI` or `ActionAPI` (for dynamic action element configurations). These classes belong to the Session API and are used to access session information (See Chapter 3: Session API for more on this API).
- An instance of `VoiceElementConfig`, `ActionElementConfig` or `DecisionElementConfig` (found in the `com.audium.server.xml` package) that contains the base configuration for the element (or `null` if there is no base configuration).

The method must return an instance of the configuration object (`VoiceElementConfig`, `ActionElementConfig` or `DecisionElementConfig`). This can be a modified version of the object passed as input to the method or one built from scratch. It is expected that should an unrecoverable error occur, the dynamic configuration class should throw an `AudiumException`.

Due to the fact that most dynamic configurations involve only a few changes to the static configuration, obtaining a base configuration as input to the execution method saves significant coding effort since the dynamic configuration class simply needs to modify this object in order to create the final configuration object then return it.

All three configuration classes extend a common base class, `ElementConfig`. This class defines those features common to all three element configurations: settings, element and session data created, custom log content, and associating the call with a UID. `ActionElementConfig` and `DecisionElementConfig` are essentially identical, separate classes are used for design considerations and for possible future differentiation. `VoiceElementConfig`, however, expands upon the `ElementConfig` class by introducing voice element only features: local hotlinks, VoiceXML properties and audio groups. The three configuration classes allow the developer to obtain everything about a configuration as well as change or add to the configuration in any way.

In order to handle audio groups, `VoiceElementConfig` introduces inner classes that define an audio group (`AudioGroup`) and a generic audio item (`AudioItem`). Two additional inner classes define audio item types that extend the `AudioItem` class to define a Say It Smart audio item (`SayItSmart`) and a static audio item (`StaticAudio`). The `AudioGroup` class encapsulates any number of `AudioItem` objects of either type. A developer can create new audio groups separately and call a method in `VoiceElementConfig` to add the audio group to the configuration, or an existing `AudioGroup` object can be obtained, modified, then reinserted into the configuration.

Meanwhile, voice element now supports local hotlinks which add VoiceXML page-scoped hotlinks but not to the root document like what global hotlinks do. `VoiceElementConfig` introduces an inner class called LocalHotlink to wrapper the configurations for a local hotlink.

The Javadocs provide much more detail regarding these classes and their methods.

## Using the XML API

Dynamic element configurations using the XML API send four HTTP POST arguments to the URI specified:

- “name”. The name of the element whose configuration is dynamic as a string.
- “inputs”. One of the standard arguments passed to all components utilizing the XML API as described in Chapter 3: Session API.
- “settings”. One of the standard arguments passed to all components utilizing the XML API as described in Chapter 3: Session API.

- "base". The base configuration for the element represented as an XML document. If there is no base configuration, this argument is not included. There are two possible DTDs for this argument. One is used if the dynamic configuration is for a voice element and the other is if the dynamic configuration is for decision and action elements.

The response must contain the final configuration to use, which follows the same DTD as the base configuration XML document. Incidentally, this DTD is the same one used for the fixed element configuration XML files created by Builder for Call Studio.

### Decision and Action Element Configuration DTD

Figure 6-1 shows the DTD for decision and action element configurations sent in the argument "base". The DTD for decision element configurations is defined in the file `DecisionElementConfiguration.dtd` and the DTD for action element configurations is defined in the file `ActionElementConfiguration.dtd`, both in the Call Services `dtds` folder. Each are stored as separate files despite being syntactically identical in order to allow for future divergence.

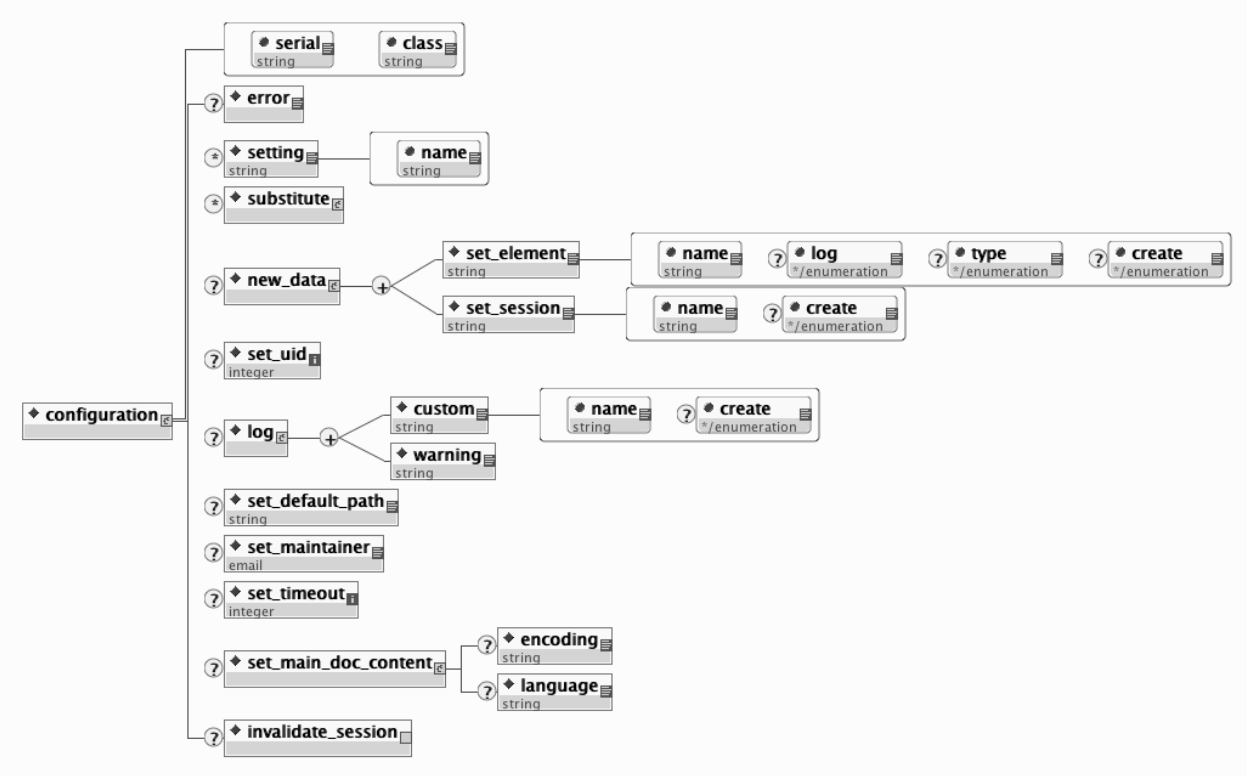


*Figure 6-1*

The tags in this XML document are:

- **configuration** – The root tag. The `class` attribute refers to the Java class defining the configurable action or decision element whose configuration is being dynamically produced.

Refer to the Element Specifications for Cisco Unified Call Services, Universal Edition and Unified Call Studio document for the full Java class names of all Universal Edition elements. The `serial` attribute is used by Call Studio and can be safely ignored here.

- **error** – This tag reports to Call Services that an error occurred while executing the dynamic configuration. Call Services will then throw an exception whose message is contained in the `<error>` tag. This allows the XML API to throw exceptions just as the Java API does.
- **setting** – This tag holds an element setting, the name appearing in the `name` attribute and the value of the setting contained within the `<setting>` tag. It is repeated for each setting included in the base configuration. No `<setting>` tags appear if the base configuration contains no settings or the element itself defines no settings.
- **substitute** – This tag holds information on substitution. Substitution is typically used in static configurations and since static and dynamic configuration XML documents share the same DTDs, it appears here. Substitution would not normally be used with dynamic configurations. The substitution tag contents are fully described in the section entitled "Substitution XML format" at the end of this chapter.
- **new_data** – This tag holds the element and session data this dynamic element configuration is to create. Any number of `<set_element>` and `<set_session>` tags can appear, one for each element and session data variable to be created. The `log` attribute of `<set_element>` sets whether the value of the variable is stored in the activity log. The optional `type` attribute is used to specify the data type of the variable and can be *string*, *int*, *float*, or *boolean*. The `create` attribute found in both tags determines when the variable is created, before the element is entered (*before_enter*), or after the element exits (*after_exit*).
- **set_uid** – This tag is used to associate the call with a UID in the user management system. The content of the tag should be the integer UID.
- **log** –This tag is used to trigger logger events when this dynamic configuration is executed. Any number of `<custom>` tags can appear, denoting the triggering of a custom event. The `name` attribute holds the name of the data, and the `<custom>` tag encapsulates the value. Any number of `<warning>` tags can appear, denoting the triggering of a warning event. The `<warning>` tag encapsulates the warning message.
- **set_default_path** – This tag is used to change the default audio path from this point onwards for this call.
- **set_maintainer** – This tag is used to change the maintainer e-mail address from this point onwards for this call.
- **set_timeout** – This tag allows the timeout length set for this session to be changed. The contents of the tab must be an integer representing the number of minutes in the timeout.
- **set_main_doc_content** – This tag allows the encoding and language settings for the application to be changed for this call. The `<language>` tag content is formatted according to the specification for using languages in VoiceXML (e.g. "en-US"). The `<encoding>` tag content is formatted according to the specification for encoding XML pages (e.g. "UTF-8").

- **invalidate_session** – This tag, if included in the XML, will prompt Call Services to invalidate the call session it retains in memory, call the end of call class or URI (if defined), and free up the Call Services port utilized by the call. The session is invalidated only after the execution method of the dynamic configuration is completed. This tag is rarely used and would be needed in a few circumstances where some external process takes the call away from Call Services such as when using a CTI system to transfer the call to an agent.

### Voice Element Configuration DTD

Figure 6-2 shows the DTD diagram for the voice element configuration XML document sent in the argument "base". The DTD is defined in the file `VoiceElementConfiguration.dtd` found in the Call Services `dtds` folder.

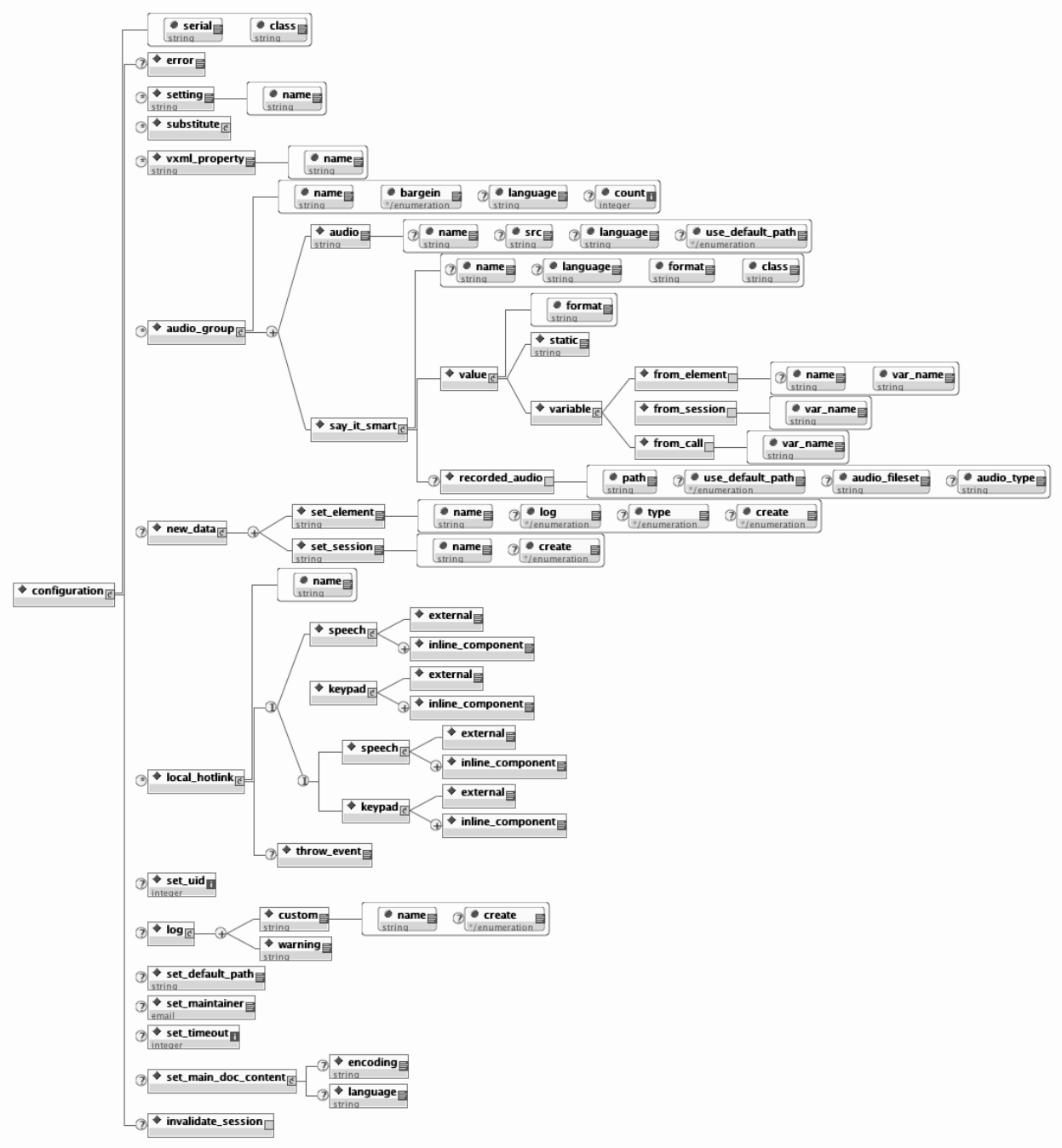


*Figure 6-2*

The tags in this XML document are:

- **configuration** – The root tag. The `class` attribute refers to the Java class defining the configurable voice element whose configuration is being dynamically produced. Refer to the Element Specifications for Cisco Unified Call Services, Universal Edition and Unified Call Studio document for the full Java class names of all Universal Edition elements. The `serial` attribute is used by Call Studio and can be safely ignored here.
- **error** – This tag reports to Call Services that an error occurred while executing the dynamic configuration. Call Services will then throw an exception whose message is contained in the `<error>` tag. This allows the XML API to throw exceptions just as the Java API does.
- **setting** – This tag holds an element setting, the name appearing in the `name` attribute and the value of the setting contained within the `<setting>` tag. It is repeated for each setting included in the base configuration. No `<setting>` tags appear if the base configuration contains no settings or the element itself defines no settings.
- **substitute** – This tag holds information on substitution. Substitution is typically used in static configurations and since static and dynamic configuration XML documents share the same DTDs, it appears here. Substitution would not normally be used with dynamic configurations. The substitution tag contents are fully described in the section entitled "Substitution XML format" at the end of this chapter.
- **vxml_property** – This tag holds a VoiceXML property, the name appearing in the `name` attribute and the value of the property contained within the `<vxml_property>` tag. It is repeated for each VoiceXML property referred to in the base configuration.
- **audio_group** – This tag holds all the audio items for a single audio group. Attributes to `<audio_group>` set its name, bargein preference and count (for those audio groups that can have counts greater than 1), and the language that it encapsulates. Each audio item is represented as a single `<audio>` or `<say_it_smart>` tag.
  - The `<audio>` tag defines a name for the audio item, the source of the audio file (optional if no audio file is being referenced), whether to use the default audio path (the `use_default_path` attribute may be *true* or *false*), and encapsulates the TTS backup message.
  - The `<say_it_smart>` tag's attributes define the name of the audio item, the output format to represent the data, and Java class name of the Say It Smart plugin. Its contents encapsulate a `<value>` tag representing either a static value or a value from a variable. The `format` attribute of `<value>` defines the input format of the data. The `<variable>` tag contains tags for obtaining the data from element data, session data or call data. The `var_name` attribute can contain the following values: *ani*, *dnis*, *iidigits*, *uui*, *start_date*, *start_time*, and *application_name*. Note that one can avoid using the `<variable>` tag by referring to a substitution string in the contents of the `<value>` tag. This also allows for the substitution of content in addition to element, session, and call data. The `<variable>` tag remains for backwards compatibility and for those not willing to use substitution.
- **new_data** – This tag holds the element and session data this dynamic element configuration is to create. Any number of `<set_element>` and `<set_session>` tags can appear, one for each

element and session data variable to be created. The `log` attribute of `<set_element>` sets whether the value of the variable is stored in the activity log. The optional `type` attribute is used to specify the data type of the variable and can be *string*, *int*, *float*, or *boolean*. The `create` attribute found in both tags determines when the variable is created, before the element is entered (*before_enter*), or after the element exits (*after_exit*).

- **Local_hotlink** – This tag is used for local hotlink configurations. The name attribute defines this local hotlink's name which must be unique in the same VoiceXML page.
  - The <speech> tag indicates whether the inline or external speech grammar was set for this hotlink.
  - The <keypad> tag indicates whether the inline or external DTMF grammar was set for this hotlink.
  - The <throw_event> tag tells if this hotlink throws an event or has an exit state.
- **set_uid** – This tag is used to associate the call with a UID in the user management system. The content of the tag should be the integer UID.
- **log** – This tag is used to trigger logger events when this dynamic configuration is executed. Any number of `<custom>` tags can appear, denoting the triggering of a custom event. The `name` attribute holds the name of the data, and the `<custom>` tag encapsulates the value. Any number of `<warning>` tags can appear, denoting the triggering of a warning event. The `<warning>` tag encapsulates the warning message.
- **set_default_path** – This tag is used to change the default audio path from this point onwards for this call.
- **set_maintainer** – This tag is used to change the maintainer e-mail address from this point onwards for this call.
- **set_timeout** – This tag allows the timeout length set for this session to be changed. The contents of the tab must be an integer representing the number of minutes in the timeout.
- **set_main_doc_content** – This tag allows the encoding and language settings for the application to be changed from this point onwards for this call. The `<language>` tag content is formatted according to the specification for using languages in VoiceXML (e.g. "en-US"). The `<encoding>` tag content is formatted according to the specification for encoding XML pages (e.g. "UTF-8").
- **invalidate_session** – This tag, if included in the XML, will prompt Call Services to invalidate the call session it retains in memory, call the end of call class or URI (if defined), and free up the Call Services port utilized by the call. The session is invalidated only after the execution method of the dynamic configuration is completed. This tag is rarely used and would be needed in a few circumstances where some external process takes the call away from Call Services such as when using a CTI system to transfer the call to an agent.

### Substitution XML Format

The DTD for element configuration XML documents contain a tag `<substitute>` that is used to define substitution. Substitution is the process of constructing a value from a combination of static and dynamic content. It is used as a way for a developer to use dynamic content in an element configuration without having to resort to a dynamic configuration. Substitution can be used throughout an element's configuration such as settings, audio, VoiceXML properties, etc. See the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more on substitution.

Since the DTDs of the documents returned by the XML API are the same as those for static element configurations produced by Builder for Call Studio, dynamic configurations may also utilize substitution. Using substitution in dynamic configurations, however, makes little sense as the dynamic configuration is produced by programming code which could just as easily set the appropriate value rather explicitly rather than assemble it using substitution. To be comprehensive, this section briefly describes the contents of the `<substitute>` tag.

A value for a setting, audio source, audio TTS or any other configuration option that supports substitution contains static content combined with integer values encapsulated by braces. When this format is detected, Call Services knows to replace (substitute) the parts encapsulated in braces with the dynamic data. For example, "`http://{0}/grammar/{1}`" as a value for a setting indicates to substitute some dynamic content for "{0}" and "{1}", where the indices are used for uniqueness (the same index can be used multiple times in the same value or in separate values if applicable).

This is where the `<substitute>` tag comes in. Each `<substitute>` tag specifies what dynamic data to substitute for a particular number surrounded by braces. The `index` attribute must be an integer that matches the number to substitute. A diagram of what it can contain is shown in Figure 6-3.

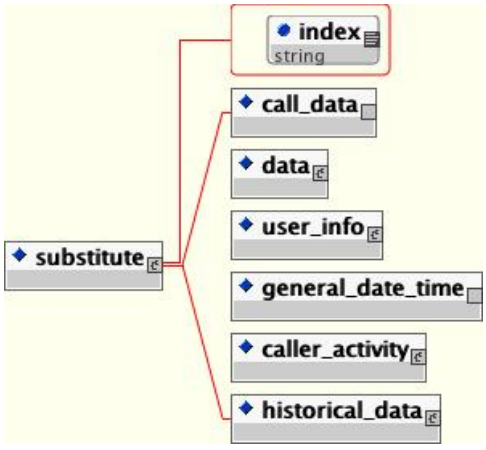


*Figure 6-3*

The content to substitute can be one of the six possible tags:

- **call_data** – Represents call information such as the ANI.
- **data** – Represents element or session data.
- **user_info** – Represents information about the user associated with the call (available only when the user management system is turned on and the call is associated with a particular UID).
- **general_date_time** – Represents the current time or the start of the call.
- **caller_activity** – Represents the activity taken by the caller in this call.
- **historical_data** – Represents past actions taken by the user associated with this call (available only when the user management system is turned on and the call is associated with a particular UID).

These tags are identical to tags of the same name used within the Universal Edition XML decision format. These tags are fully described in Chapter 2 of the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio.

For example, in the above situation where a setting has the value "`http://{0}/grammar/{1}`", the following substitute tag can represent index 0 coming from element data:

```
<substitute index="0">
      <data>
            <element name="AnElementName" variable="SomeValue"/>
      </data>
</substitute>
```

# Chapter 7: Standard Action Elements

Action elements are responsible for performing some action and returning an indication whether the action was a success. A pre-built, configurable action element has already defined the actions to take and only requires a configuration to modify its behaviors. Standard action elements, however, are defined by the developer and have no configuration since they represent actions specific to an application.

A standard action element, in addition to the functionality provided all components, is allowed to create and modify element data. It can also act as a flag if desired.

## Using the Java API

A standard action element is built in Java by extending the abstract base class `ActionElementBase` found in the `com.audium.server.voiceElement` package (this package's name is such due to backwards compatibility considerations). It contains a single abstract method named `doAction`, that acts as the execution method for the action element, and must be implemented by the developer. The method receives two arguments: the name of the action element (as a `String`) and an instance of `ActionElementData`. This class belongs to the Session API and is used to access session information (See Chapter 3: Session API for more on this API). The method does not expect anything in return because all action elements have a single exit state ("done"). It is expected that should an unrecoverable error occur, an `AudiumException` is thrown.

The `ActionElementBase` class defines many methods in addition to `doAction`. These are used for configurable action elements, which also extend the class. The only method required for standard action elements is `doAction`, as it is the only abstract method in `ActionElementBase`.

## Using the XML API

As described in Chapter 3: Session API, the standard "inputs" and "settings" XML documents are sent via POST to the standard action element URI. An additional parameter, called "name", is sent containing the name of the action element. Figure 7-1 shows the DTD diagram of the XML document that must be sent in response. The DTD for the standard action element response is defined in the file `ActionResponse.dtd` found in the Call Services `dtds` folder.

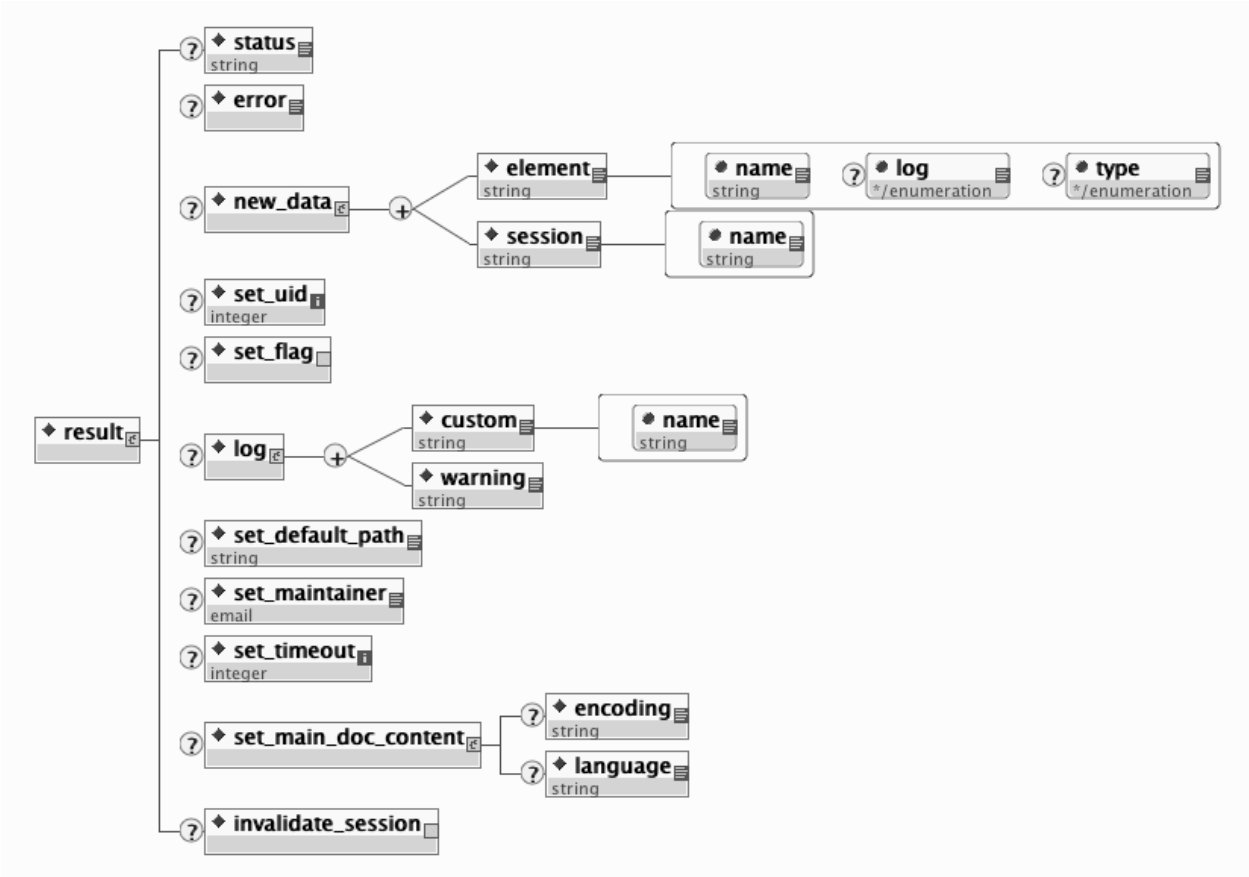


*Figure 7-1*

The elements in this XML document are:

- **status** – Since the XML API accesses a process that exists in context separate from Call Services, there is no automatic way for an error that occurs during the creation of a response to be caught and handled properly by Call Services. This tag exists to simulate that process by containing either the word "success" or a text message describing the error. When anything but "success" is returned, Call Services throws an exception using the content of `<status>` as the error message. This way, from the perspective of Call Services and the application logs, the result will be the same no matter whether the Java API or the XML API is used. See the description for the `<error>` tag below as there is some overlap in functionality.
- **error** – This tag reports to Call Services that an error occurred while executing the standard action. Call Services will then throw an exception whose message is contained in the `<error>` tag. This tag acts almost exactly like the `<status>` tag and was introduced later to allow for consistency across all components. An error listed in this tag takes precedence over an error message listed in the `<status>` tag. The `<status>` tag must still be used to indicate that the standard action element executed without error by containing the word "success".

- **new_data** – This tag holds the element and session data this standard action element is to create. Any number of `<set_element>` and `<set_session>` tags can appear, one for each element and session data variable to be created. The `log` attribute of `<set_element>` sets whether the value of the variable is stored in the activity log. The optional `type` attribute is used to specify the data type of the variable and can be *string*, *int*, *float*, or *boolean*. The `create` attribute found in both tags determines when the variable is created, before the element is entered (*before_enter*), or after the element exits (*after_exit*).
- **set_uid** – This tag is used to associate the call with a UID in the user management system. The content of the tag should be the integer UID.
- **set_flag** – This tag is used to make the action element act like a flag when visited. If it appears, a flag with the same name as the action element will be considered triggered and that fact will be noted in the activity log.
- **log** – This tag is used to trigger logger events when this standard action element is executed. Any number of `<custom>` tags can appear, denoting the triggering of a custom event. The `name` attribute holds the name of the data, and the `<custom>` tag encapsulates the value. Any number of `<warning>` tags can appear, denoting the triggering of a warning event. The `<warning>` tag encapsulates the warning message.
- **set_default_path** – This tag is used to change the default audio path from this point onwards for this call.
- **set_maintainer** – This tag is used to change the maintainer e-mail address from this point onwards for this call.
- **set_timeout** – This tag allows the timeout length set for this session to be changed. The contents of the tab must be an integer representing the number of minutes in the timeout.
- **set_main_doc_content** – This tag allows the encoding and language settings for the application to be changed from this point onwards for this call. The `<language>` tag content is formatted according to the specification for using languages in VoiceXML (e.g. “en-US”). The `<encoding>` tag content is formatted according to the specification for encoding XML pages (e.g. “UTF-8”).
- **invalidate_session** – This tag, if included in the XML, will prompt Call Services to invalidate the call session it retains in memory, call the end of call class or URI (if defined), and free up the Call Services port utilized by the call. The session is invalidated only after the execution method of the standard action element is completed. This tag is rarely used and would be needed in a few circumstances where some external process takes the call away from Call Services such as when using a CTI system to transfer the call to an agent.

# Chapter 8: Standard Decision Elements

Decision elements apply business logic to decide which exit state to return. A pre-built, configurable decision element has already defined the business logic and only requires a configuration to modify its behavior. Standard decision elements, however, are defined by the developer and have no configuration since they represent decisions specific to an application. For simple to moderately complex decisions, Universal Edition provides a means of defining decisions without programming by constructing an XML document (the Universal Edition XML decision format is described in Chapter 2 of the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio). Should this format prove insufficient, Java or XML APIs are provided to allow the developer to build the business logic programmatically.

A standard decision element, in addition to the functionality provided all components, is allowed to create and modify element data.

## Using the Java API

A standard decision element is built in Java by extending the abstract base class `DecisionElementBase` found in the `com.audium.server.voiceElement` package (this package's name is such due to backwards compatibility considerations). It contains a single abstract method named `doDecision` that acts as the execution method for the decision element, and must be implemented by the developer. The method receives two arguments: the name of the decision element (as a `String`) and an instance of `DecisionElementData`. This class belongs to the Session API and is used to access session information (See Chapter 3: Session API for more on this API). The method expects a `String` object in return containing the exit state in the exact format specified in Builder for Call Studio when the standard decision element was first defined.

The `DecisionElementBase` class defines many methods in addition to `doDecision`. These are used for configurable decision elements, which also extend the class. The only method required for generic decision elements is `doDecision`, as it is the only abstract method in `DecisionElementBase`.

## Using the XML API

As described in Chapter 3: Session API, the standard "inputs" and "settings" XML documents are sent via POST to the decision element URI. An additional parameter, called "name", is sent containing the name of the decision element. Figure 8-1 shows the DTD diagram of the XML document that must be sent in response. The DTD for the generic action element response is defined in the file `DecisionResponse.dtd` found in the Call Services `dtds` folder.

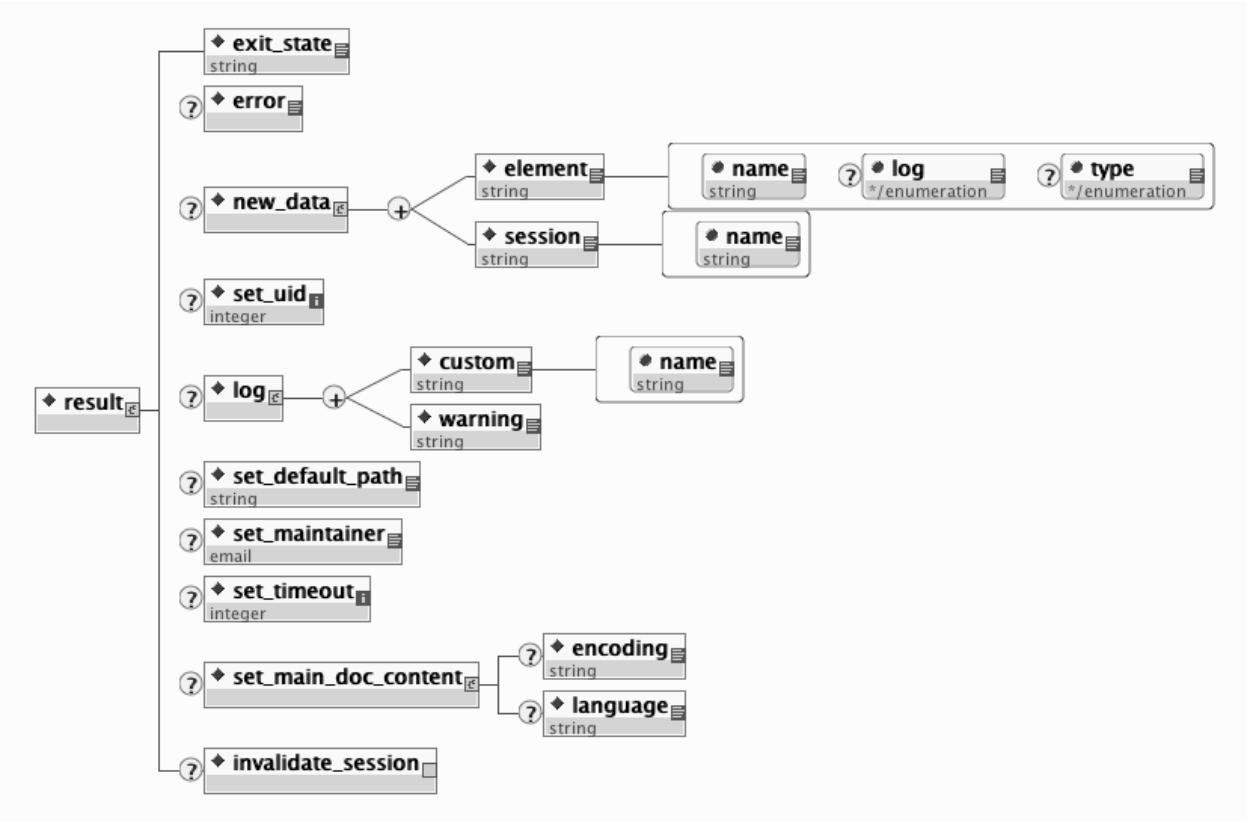


*Figure 8-1*

The elements in this XML document are:

- **exit_state** – This tag contains the string value representing the exit state in the exact format specified in Builder for Call Studio when the standard decision element was first defined.
- **error** – This tag reports to Call Services that an error occurred while executing the standard decision element. Call Services will then throw an exception whose message is contained in the `<error>` tag. This allows the XML API to throw exceptions just as the Java API does.
- **new_data** – This tag holds the element and session data this standard decision element is to create. Any number of `<set_element>` and `<set_session>` tags can appear, one for each element and session data variable to be created. The `log` attribute of `<set_element>` sets whether the value of the variable is stored in the activity log. The optional `type` attribute is used to specify the data type of the variable and can be *string*, *int*, *float*, or *boolean*. The `create` attribute found in both tags determines when the variable is created, before the element is entered (*before_enter*), or after the element exits (*after_exit*).
- **set_uid** – This tag is used to associate the call with a UID in the user management system. The content of the tag should be the integer UID.

- **set_flag** – This tag is used to make the decision element act like a flag when visited. If it appears, a flag with the same name as the decision element will be considered triggered and that fact will be noted in the activity log.
- **log** – This tag is used to store information in log files for this application when this standard decision element is executed. Any number of `<custom>` tags can appear, denoting the information to insert in the application's activity log. The `name` attribute holds the name of the data, and the `<custom>` tag encapsulates the value. Any number of `<warning>` tags can appear, denoting warnings to be placed in the application's error log. The `<warning>` tag encapsulates the warning message.
- **set_default_path** – This tag is used to change the default audio path from this point onwards for this call.
- **set_maintainer** – This tag is used to change the maintainer e-mail address from this point onwards for this call.
- **set_timeout** – This tag allows the timeout length set for this session to be changed. The contents of the tab must be an integer representing the number of minutes in the timeout.
- **set_main_doc_content** – This tag allows the encoding and language settings for the main VoiceXML documents to be changed from this point onwards for this call. The `<language>` tag content is formatted according to the specification for using languages in VoiceXML (e.g. "en-US"). The `<encoding>` tag content is formatted according to the specification for encoding XML pages (e.g. "UTF-8").
- **invalidate_session** – This tag, if included in the XML, will prompt Call Services to invalidate the call session it retains in memory, call the end of call class or URI (if defined), and free up the Call Services port utilized by the call. The session is invalidated only after the execution method of the standard decision element is completed. This tag is rarely used and would be needed in a few circumstances where some external process takes the call away from Call Services such as when using a CTI system to transfer the call to an agent.

# Chapter 9: Configurable Elements

The large array of Universal Edition Elements bundled with Universal Edition software encapsulate a lot of functionality a typical voice application requires. There are, however, situations where more customized, proprietary, or robust elements are desired. This is certainly the case with action and decision elements, which tend to be highly specialized for interfacing with proprietary backend systems or implementing custom business logic. To a lesser extent this applies for voice elements as well. Universal Edition has tried to provide pre-built voice elements that cover most of what a typical voice application requires. There is always the option of using VoiceXML insert elements to handle a situation that there are no voice elements to manage. The disadvantages of VoiceXML insert elements, however, are in their inconsistencies across various voice browsers, the size they can be before management becomes burdensome, the difficulty of interacting with backend systems, and their overall performance. Custom voice elements are the best way to achieve these goals as they are fast, use the Universal Edition Voice Foundation Classes (VFCs) that work consistently across multiple voice browsers, and are built with Java, so complex calculations and integrations are simple.

Universal Edition software was designed to be modular without compromising integration. Custom elements are integrated into both Call Services and Call Studio as easily as Universal Edition Elements are. They can be deployed for a specific application or shared across all applications and are configured in the Call Studio alongside Universal Edition Elements, including supporting dynamic configurations. With such seamless integration and effortless deployment, a developer can, over time, create entire libraries of custom elements to use for their voice applications or potentially for resale.

Due to the requirements to integrate with both Call Studio and Call Services, configurable elements can only be constructed using the Java API. This is not to be confused with standard action and decision elements, both of which can be constructed using both Java and XML APIs. It is the configuration of a configurable element that places demands that can only be met by the Java API. Building standard action and decision elements are fully explained in Chapter 7: Standard Action Elements and Chapter 8: Standard Decision Elements.

This chapter describes in detail how to create custom voice, action, and decision elements and integrate them into both Call Studio and Call Services.

## Design

Configurable elements are built by extending an abstract Java class. This base class lays out the methods used to identify the element's configuration, how it is executed, and any utility methods available to it. The developer simply implements the appropriate methods and uses the Java API provided to build the element. Within the execution method, the developer is then free to use Java in any way possible, such as creating a complex class hierarchy, accessing files or backend systems, utilizing third party libraries, even communicating with external systems via HTTP or RMI. Voice elements must extend `VoiceElementBase`, action elements `ActionElementBase`,

and decision elements `DecisionElementBase`, all found in the `com.audium.server.voiceElement` package. These three base classes all extend a common base class for configurable elements, `ElementBase`.

In order for Builder for Call Studio to identify the Java class representing the actual element (as opposed to another class lower in the class hierarchy or a standard element that also extends that class), a "marker" Java interface named `ElementInterface` must be implemented. Only those classes that implement this interface are shown in the Builder's Element Pane.

Each configurable element contains a single execution method in which the element performs its function. This method is called by Call Services when that element is visited in the call flow. One can equate this method to the element's `main()` method, it begins and ends there.

One argument to the method is an instance of a Session API class. Aside from the standard functionality available in this class such as obtaining the ANI and setting element data, this API class can be used to obtain a Java object representing the element's configuration. Call Services automatically creates this configuration object with the data entered by the application developer in the Builder or made available through a dynamic configuration. The name of the execution method and the API class passed to it differ for each element type as do the classes encapsulating the element's configuration. The execution method for decision and voice elements must return an exit state and the execution method for action elements do not return anything (since all action elements explicitly have a single exit state).

The execution method for action and decision elements can throw an `AudiumException` while voice elements can throw an `ElementException`. The developer would throw this exception if an error was encountered within the execution method that the developer wishes to end the call. A call that encountered this exception would then visit the error element (or the application-specific error message if the error element was not defined), and the error message is placed in the error log including the exception's full stack trace.

The base class also includes various methods used to define the element's configuration. These methods define everything from the element's name to its possible exit states. These methods are essential for Builder for Call Studio to visually render the element and its configuration correctly. Custom elements will be indistinguishable from Universal Edition Elements within the Builder. The developer can choose as simple or complex a configuration as desired (or even no configuration at all, though it wouldn't be very reusable).

Note that element data generated by an element will be overwritten if that same element is visited again in the call flow. For example, a variable set by a voice element handling the main menu of an application will be reset the next time the main menu is visited. The activity log, however, is a historical account of the call, so would have all values of the element data. Should the developer wish to retain all data created by all visits to the element, they must build that into the element, such as creating new variables or appending the new value to an existing variable.

## Common Methods

The methods listed below are defined in `ElementBase` and are common to all configurable elements no matter what type. All custom Universal Edition classes used by these methods are defined in the `com.audium.server.voiceElement` package. Refer to the Javadocs for more in depth explanations of these methods and the classes they utilize.

```
String getElementName()
```

This returns the display name for the element. This is the name displayed in the Element Pane of Builder for Call Studio. There are no restricted characters for the display name, though best practices recommend a short name that avoids spaces and punctuation.

```
String getDescription()
```

This returns the description of the element. The Builder displays this information in a tool tip when the cursor is placed above the element's icon in the Element Pane. There is no restriction on the size or contents of the description.

```
String getDisplayFolderName()
```

This returns the name of the folder in the Builder Element Pane in which the element resides. If `null` is returned, the element appears directly under the Elements folder (currently, only the Audio voice element appears directly under the Elements folder). To support a hierarchy of folders, the folder name can include a full path and the folder tree will automatically be generated by the Builder (i.e. "`MyElements/Financial/Banking/`" would put the element icon inside three levels of folders). Best practices recommend short folder names that avoid spaces and punctuation.

```
ExitState[] getExitStates()
```

This method defines the exit states this element can return. It is necessary in order for the Builder to properly render the exit state dropdown menu when the element is right-clicked. The method returns an array of `ExitState` classes. The `ExitState` class encapsulates the real and display name for an exit state. The display name is used only by the Builder and the real name is used everywhere else (within code or XML decisions). Note that for configurable action elements, this method need not be implemented as all action elements automatically have a single exit state named "done".

```
ElementData[] getElementData()
```

This method describes the element data generated by this element. This method returns an array of `ElementData` objects or can be `null` if the element does not create any element data. The `ElementData` object encapsulates the variable's name and its description. The description exits for future compatibility when the Builder obtains the ability to display a tool tip for element data. Note that element data does not have a display name.

At this juncture, neither the Builder nor Call Services use the information returned from this method. One of the reasons is that for some elements, what element data is created may not be known until runtime. In future versions, the Builder may utilize the results of this method to aid the application designer in choosing element data when using substitution. In the meantime, the method must be configured to return something, either `null` or at least one element data variable.

```
Setting[] getSettings()
```

This method describes the settings this element contains in its configuration. All element types have settings which allow a designer to control how the element functions when visited. The Builder uses information returned in this method to render the *Settings* tab in the Element Configuration Pane. This method returns an array of `Setting` objects, where each `Setting` object represents a single setting rendered on a separate line in the Element Configuration Pane. The setting object lists the following information about a setting:

- Its real and display name. The Builder uses the display name in the Element Configuration Pane and the real name is used everywhere else (such as when referring to the setting in the element Java code).
- The setting data type. The data type determines how the information is entered in the Builder. The following lists the different data types available:
  - **Boolean.** The boolean data type displays a dropdown menu with two options, *true* and *false*.
  - **Enumerated.** The enumerated data type displays a dropdown menu with the options defined in the enumeration.
  - **Float.** The float data type displays a text box that accepts only floating point numbers. The setting can define upper and lower limits for the entered number.
  - **Integer.** The integer data type displays a text box that accepts only integer numbers. The setting can define upper and lower limits for the entered number.
  - **String.** The string data type displays a text box that accepts any input.
  - **Textbox.** The textbox data type displays a button that when clicked produces a large text field that accepts any input. It should be used when text content tends to be long and it is desirable to give more space to enter the setting value than would normally be given with a string data type.
- Whether the setting is required or repeatable. A required setting is displayed with a red star next to it and the Builder will not allow the application to be deployed or validated if this setting is left blank. A repeatable setting is displayed with a plus sign next to it and can be given multiple values by the application designer. A setting can be both required and repeatable in which case there must be at least one value.
- Whether the setting allows substitution within it. Substitution is a mechanism for specifying the assembly of dynamic content in the Builder at build time. A setting can be configured to

allow substitution or prevent it, for example if that setting's behavior would be too unpredictable if its value were set dynamically. See the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more on substitution.

- The setting default value. When an element is dragged to the workspace for the first time, the element can specify default values for all settings. This allows the application designer to create applications very rapidly by choosing the default values and tweaking them later during the testing phase. A setting need not have a default value, especially if one cannot be predicted (such as a call transfer phone number).
- Setting dependencies. A setting can specify criteria that determines whether it appears in the Builder's Element Configuration Pane. The criteria involves setting the relationship between the setting and the other setting(s) that it depends on. One can even build complex logic expressions for determining when the setting appears. The main purpose for setting dependencies is to simplify the process of configuring the element in the Builder. By displaying only the settings appropriate for the configuration the designer desires, settings that are not applicable can be safely hidden to avoid confusion and possible conflicting information. For example, a setting for specifying the confidence value of a data capture field would not be required if the input mode of the element were set to DTMF. So one can set the confidence setting to appear only when the input mode setting is not DTMF. Many Universal Edition Elements employ dependencies to simplify their configuration and so are good examples of how dependencies are implemented.

  A setting's dependencies are defined by using the `Dependency` Java class. A single dependency instance defines any number of setting values combined with the "and" logical operator. For example, single `Dependency` object can define that a setting appears only if one setting is set to "true" and another is set to "10".

  A setting can take an array of `Dependency` objects. Each `Dependency` object in the array is considered combined with the "or" logical operator. So to make a setting appear when one setting is "true" *or* another setting is "10", two `Dependency` objects are created and placed in a 2-member array.

  Configuring dependencies for settings can be complex and involved, though once set up, they can greatly simplify the configuration of a complex element. The Javadocs for the various classes describe what each class and its member methods are used for.

## Configuration Classes

As described in the above section, each configurable element's execution method receives a Session API class that is used to obtain the element's configuration. Do not confuse this with the methods in `ElementBase` that are used to describe the configuration to Builder for Call Studio. We are referring here to a Java object that contains a full configuration the application designer entered in the Builder or that came from a dynamic element configuration Java class or via the XML API.

These Java classes, `ActionElementConfig`, `DecisionElementConfig`, and `VoiceElementConfig` are found in the `com.audium.server.xml` package and all extend the base class `ElementConfig`. These classes are identical to those used when constructing dynamic element configuration classes. Refer to Chapter 6: Dynamic Element Configurations for a good description of these configuration classes. While dynamic configuration are responsible for creating and editing a configuration object, a custom element uses these classes basically to read their information. There is little reason to edit the configuration classes within a custom element as the configuration object applies only to that particular use of the element (revisiting the element will provide it with a new configuration).

## Action Elements

A configurable action element extends the abstract Java class `ActionElementBase` found in the `com.audium.server.voiceElement` package. This class has default implementations for the abstract configuration methods inherited from the `ElementBase` class. The default implementation sets an empty configuration (null name, folder name, description, element data, settings, and one exit state named “done”). This was done because this same base class is extended to create standard action elements as well as configurable action elements. What makes a configurable action element different is the fact that it defines a specific configuration, so the custom action element must re-implement all the configuration methods rather than relying on the default implementation. Some of the default implementations, though, may be appropriate even for a custom action element. For example, the default implementation of the `getExitStates` method returns a single exit state named “done”, which applies to all action elements and so a custom action element need not implement this method.

The execution method, `doAction()`, receives an instance of the API class `ActionElementData`. This class belongs to the Session API and is used to access session information (See Chapter 3: Session API for more on this API). In addition to providing access to session information, this API class is also used to return the action element configuration that drives the functionality of the element. The `getActionElementConfig()` method in `ActionElementData` returns an `ActionElementConfig` object. Call Services takes care of obtaining the appropriate configuration and returning it in this method, whether or not the configuration is dynamic. The element need not worry about where the configuration came from.

`ActionElementConfig` is almost a direct extension of the base `ElementConfig` class. It is kept separate for future differentiation.

## Decision Elements

A configurable decision element extends the abstract Java class `DecisionElementBase` found in the `com.audium.server.voiceElement` package. This class has default implementations of the abstract configuration methods inherited from the `ElementBase` class. The default implementation sets an empty configuration (null name, folder name, description, element data, settings, and exit states). This was done because this same base class is extended to create

generic decision elements as well. What makes a configurable decision element different is the fact that it defines a specific configuration, so the custom decision element must re-implement all the configuration methods rather than relying on the default implementation. Some of the default implementations, though, may be appropriate even for a custom decision element. For example, if the decision element creates no element data, the custom decision element need not implement the `getElementData` method as the default implementation is sufficient.

The execution method, `doDecision()`, receives an instance of the API class `DecisionElementData`. This class belongs to the Session API and is used to access session information (See Chapter 3: Session API for more on this API). In addition to providing access to session information, this API class is also used to return the decision element configuration that drives the functionality of the element. The `getDecisionElementConfig()` method in `DecisionElementData` returns a `DecisionElementConfig` object. Call Services takes care of obtaining the appropriate configuration and returning it in this method, whether or not the configuration is dynamic. The element need not worry about where the configuration came from.

`DecisionElementConfig` is almost a direct extension of the base `ElementConfig` class. It is kept separate for future differentiation.

# Voice Elements

Voice elements are more complex custom elements because they are responsible for producing VoiceXML pages to send to the voice browser. The execution method for voice elements contains more arguments and the voice element class requires additional configuration methods to be implemented. Finally, while action and decision elements complete in one call of the execution method, a typical voice element requires multiple VoiceXML pages to be produced in a certain order determined at runtime. Voice elements, therefore, must have state management where other elements do not.

It is important to understand how voice elements integrate with Call Services and the voice browser to prepare the developer for constructing voice elements. Unlike a traditional static VoiceXML page or a script-generated VoiceXML page that is accessed directly from the voice browser, the system uses Call Services as an abstraction layer between the voice browser and the voice element that produces the VoiceXML pages. This abstraction layer not only allows the developer to avoid coding to a specific browser, it also saves the developer from having to deal with HTTP request and response management. Each page the voice element produces is passed through Call Services, which acts as the central access point for the voice browser. Each link for a new document specified in the VoiceXML page points back to Call Services and Call Services' internal call flow data indicates which voice element it is currently visiting. All arguments passed by the voice browser through those links are sent by Call Services to the voice element for it to manage.

Each VoiceXML page generated by a voice element begins as a "shell" page that contains the VoiceXML Call Services requires. The voice element then adds to this page any custom

VoiceXML content desired before passing it back to Call Services to send to the voice browser. Most voice elements will require multiple pages, the content of each depending on the actions of the caller. When a voice element is done producing VoiceXML pages, it returns an appropriate exit state so Call Services can visit the next element according to the call flow. As long as the proper VoiceXML is passed back through the execution method, the developer is free to do anything allowed by Java, including creating helper classes, accessing backend systems, etc.

Having a single voice element class produce multiple VoiceXML pages poses a problem. With multiple calls simultaneously accessing the element in various stages of a call, how is the voice element to know where a particular caller is within the element at any one moment in time? Universal Edition helps by providing "scratch space" for custom voice elements to store any data it wishes. Call Services maintains separate scratch data for each call and makes this data available to the voice element through the Session API. Usually the data stored in the scratch space will be the state of the element for a particular call. Each time the execution method is called, the element can check the scratch space, determine where the caller is within the element's internal call flow, and produce the appropriate VoiceXML page. Borrowing web application terminology, scratch space provides "session and state management" functionality to voice elements. Any Java class can be added to the scratch space so the developer can get as complex as desired in handling the state management. Note that when the voice element is complete, the scratch space is automatically cleared by Call Services, meaning that a voice element that is revisited in the same call will start off with empty scratch space.

Figure 9-1 shows a diagram that visualizes the above points concerning how a voice element interacts with Call Services and the voice browser.

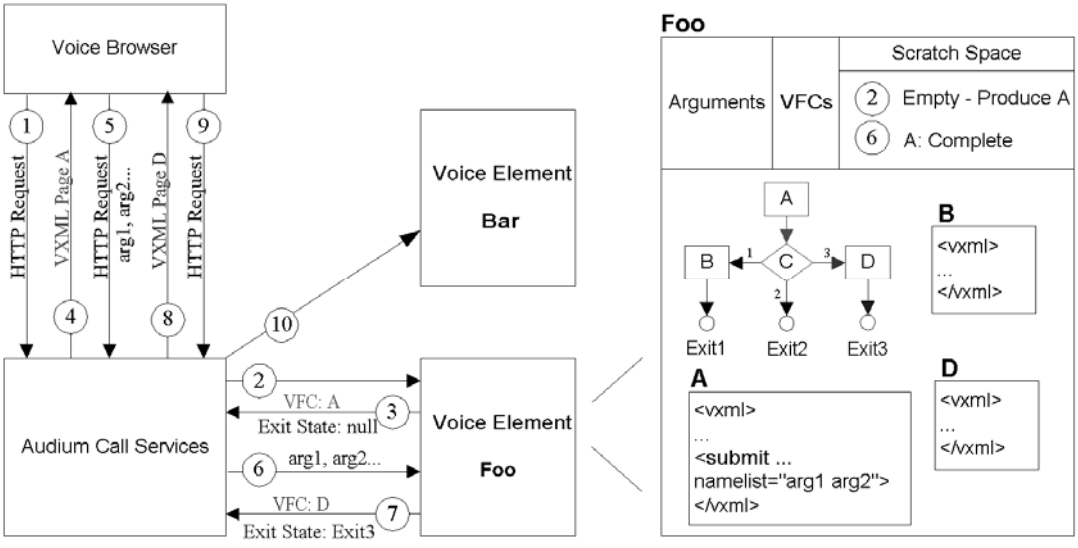


*Figure 9-1*

The diagram shows a typical exchange between the voice browser, Call Services, and several voice elements. In step 1, the voice browser makes an HTTP request for a VoiceXML page to Call Services. According to its record of the application call flow, Call Services in step 2 accesses the voice element Foo. The voice element Foo is shown on the right. The element is coded to contain within it a small call flow with three possible exit states. It can produce three separate VoiceXML pages, A, B, and C. In step 2, the scratch space is empty, indicating that the element is being visited for the first time. Foo therefore needs to produce the VoiceXML page A. Note that the A contains in it a submit that points back to Call Services and includes two arguments "arg1" and "arg2". In step 3, Foo produces the VoiceXML page by assembling VFC objects and passes those objects back to Call Services. Before exiting, it puts in the scratch space information indicating that page A was completed. Foo returns a `null` exit state, indicating to Call Services that the voice element is not done. In step 4, Call Services converts the VFC objects into a complete VoiceXML page that is sent back to the voice browser. In step 5, the voice browser has parsed the VoiceXML and makes a new request for the next VoiceXML page. The request contains the two arguments specified in the VoiceXML page A. In step 6, Call Services knows to go back to Foo because it previously returned a `null` exit state. It revisits Foo, passing along the two arguments. Since Call Services maintains the session for each call, and the scratch space is stored in the session, Foo can access the Session API to get the scratch space. Foo's scratch space indicates that A is complete. It makes a decision C based on the arguments passed to it and chooses option 3, therefore requiring page D to be produced. Foo also notes that after producing this page, it is done. In step 7, Foo returns the VFC objects containing page D and returns with the exit state "Exit3". In step 8, Call Services produces the VoiceXML page and sends it to the voice browser. In step 9, the voice browser parsed the VoiceXML page and asks for the next one. Finally, in step 10, Call Services notes that Foo returned an exit state of "Exit3" so refers to the application call flow to discover where to go once Foo returns an exit state of "Exit3". It determines that the next voice element to access is "Bar". The call continues in this manner until it ends.

### Restrictions and Recommendations

Universal Edition tries to make as few restrictions on building custom elements as possible. However, in order to ensure proper integration of custom voice elements with the rest of the system, some restrictions must be set. Additionally, there are guidelines that do not necessarily need to be followed, though they are recommended for design and tighter integration considerations.

*Restrictions*

- The voice element must produce VoiceXML using the Universal Edition Voice Foundation Classes (VFCs). The VFCs are the mechanism through which the voice element produces the VoiceXML to send to the voice browser and Call Services is tightly integrated with them. See Appendix A: The Voice Foundation Classes for a full description of the VFCs.
- The voice element cannot define its own root document. The root document is automatically generated by Call Services and is vital for many features to function such as the ability to activate hotlinks, perform logging, and activate the end of call action.

- The voice element cannot use a `<submit>` or `<goto>` to link to a VoiceXML page external to Call Services. When it is time to return to the voice element in a VoiceXML page, a URL pointing back to Call Services must be used. This is required by the design using Call Services as the abstraction layer between the voice browser and the voice element. The only exception to this rule allows for the call to continue from a Universal Edition application to an external application but only if steps are taken to properly end the Call Services session first.
- The VoiceXML `<form>` the developer wishes the browser to visit first must be named "start".
- The voice element must obtain the `VPreference` object to use in all VFC constructors by calling the `getPreference()` method in `VoiceElementData`. This `VPreference` object contains, among other things, the voice browser choice made by the application designer in Builder for Call Studio (or dynamically in a start of call action). `VPreference` is a VFC and is described in more detail in Appendix A: The Voice Foundation Classes. `VoiceElementData` is described in more detail in the following sections.

*Recommendations*

- The interaction category of the activity log is used to record the activity of callers within the VoiceXML pages produced by voice elements. Since this activity occurs on the voice browser, the only way to record this in the Call Services logs is to store the appropriate information within the VoiceXML and submit it back to Call Services. Universal Edition defines an interaction logging convention for recording caller behavior within the VoiceXML. By conforming to this convention, the developer ensures that the activity log will contain the detail expected for application testing and reporting. While interaction logging is not required in a custom voice element, not performing logging will reveal nothing about what the callers did within the element. See the section entitled "Interaction Logging" in this chapter for more. A full description of the activity log can be found in Chapter 5 of the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio.
- Try to produce VoiceXML that conforms to the VoiceXML specifications, understanding that using browser-specific functionality will prompt that element to function correctly only on that browser.
- Throw an `ElementException` where appropriate so an error logger event can be thrown. Exceptions should indicate the voice element encountered a situation which prevents it from doing its assigned task. Non-fatal exceptions should be caught within the voice element (and possibly linked to exit states) and warnings placed in the error log.
- Do not end the call in the voice element manually using the `<exit>` or `<disconnect>` tags. Instead, exit the voice element with a specific exit state that can then be linked to a hang-up element in the call flow. If it must be done, use `<disconnect>` so that Call Services can detect the hang-up and execute the on end call action. Using `<exit>` would cause Call Services to not know the calls had ended, using up a Call Services port until it times out on its own. This is obviously not a desirable situation.

- A voice element can add any Java object to scratch space via the `setScratchData()` method in `VoiceElementData`. Scratch data is used for voice element internal uses only since the scratch space is cleared when the voice element ends, even if it is revisited in the same call. Additionally, the data in the scratch space is stored in the session managed by Call Services. To minimize performance issues, placing large Java objects in the scratch space is not suggested.

### VoiceElementBase Methods

A voice element extends the abstract Java class `VoiceElementBase` found in the `com.audium.server.voiceElement` package. Like other elements, it shares methods obtained from the base class `ElementBase` described in the previous sections. `VoiceElementBase`, however, adds many additional methods that apply to voice elements only. Its execution method is also more complex.

*Execution Method*

```
String addXmlBody(VMain vxml, Hashtable params,
                  VoiceElementData data) throws VException, ElementException
```

This is the execution method for voice elements. The arguments to the method are:

- **vxml**. The `VMain` object is the container VFC object in which all VoiceXML content is added (by adding other VFC objects to it). The desired VoiceXML page to produce must be assembled using the VFCs and added to this object. If no VFCs are added, an incomplete VoiceXML page will be produced, causing the voice browser to encounter an error. Simply add all `VForm` objects to this object. Call Services will take care of the rest. Note that the form you wish to be visited first must be named *start*. ***This is very important!***
- **params**. The `params` object is a `Hashtable` that contains all HTTP arguments passed by the voice browser through Call Services. For example, if the voice element produces a VoiceXML page with a variable named *dataToCollect* that is then included in a submit argument list, the `params Hashtable` will contain an entry with the key "dataToCollect" and the value as a `String`. To access it, the developer would write:

  ```
  String data = (String) params.get("dataToCollect");
  ```

- **data**. The `VoiceElementData` object belongs to the Session API and is used to access session information (See Chapter 3: Session API for more on this API). Aside from the standard functionality, `VoiceElementData` provides all data required by the voice element such as ways to access the scratch space, obtain the `VPreference` object used to instantiate the VFCs, and obtain the voice element's configuration object, `VoiceElementConfig`.

The `String` return value of the method must refer to the real name of the voice element's exit state. The real name of the exit state must match one of the names given in the `getExitStates()` configuration method. Since voice elements can span multiple VoiceXML pages, returning `null` indicates that the voice element is not done and the execution method should be visited again when the voice browser sends its next request to Call Services. Only

when the method returns a proper exit state will Call Services follow the application call flow and proceed to the next element. The method throws an `ElementException` that is used to indicate an error occurred within the execution method that prevented the voice element from doing its assigned task. The error message will be logged in the application's error log and the error element (if applicable) will be visited. This method additionally throws a `VException`, which is thrown by incorrectly configured VFCs and does not need to be thrown explicitly by the element execution method itself.

*Utility Methods*

```
String getSubmitURL()
```

One of the restrictions listed for creating a voice element is to use a Universal Edition-specified URL when submitting back to Call Services. This method returns the URL to use. The developer would use this URL in a `<submit>` adding any arguments desired.

```
VAction getSubmitVAction(String args, VPreference pref)
```

This is a convenience method which provides more than `getSubmitURL()` does by returning a new `VAction` object containing the entire submit to Call Services along with any arguments passed as input (as a space-delimited list). The `VPreference` object is required to instantiate the `VAction` object and can be obtained by calling the `getPreference()` method of the `VoiceElementData` object. The `VAction` object returned by the `getSubmitVAction()` method is just like any other, the developer can add additional actions to it as desired.

```
VAction getSubmitVAction(VAction existing, String args)
```

This convenience method does the same as the above method except it adds the submit to an existing `VAction` object passed as input. There is no `VPreference` object required because no new VFC objects are instantiated in this method. This method may be more convenient if the developer wants to perform some actions (such as the declaration or assigning variables) *before* the submit occurs.

See Appendix A: The Voice Foundation Classes for a full description of the VFCs.

```
VoiceElementResult createSubVoiceElement(VoiceElementBase mainVoiceElement,
                        VMain vxml, Hashtable params, VoiceElementData data)
```

There are times when one wishes to create a voice element that acts a combination of a group of existing voice elements, something like a "super-element". While one can simply build this element from scratch, it would be desirable to somehow leverage the work already done with existing voice elements. The advantage would be that the super-element code need not contain any VFC code, it would only act as the container for sub-elements within it. This is possible utilizing the `createSubVoiceElement()` method.

This method is intended to be called from an instance of a sub-element, not the super-element. The super-element first creates an instance of the sub-element and then calls this method from

that instance. This executes a sub-element within the super-element using the same context. The sub-element executes normally, it reads from a configuration object and creates VFC classes. The difference is that the super-element can take what the sub-element produced and modify it or add to it.

The arguments to the `createSubVoiceElement()` method are required in order to provide the correct context for the sub-element to execute within. A voice element requires the correct context in order for it to be able to read from a configuration object, use the appropriate VFC objects, and have access to the system. The first argument to the `createSubVoiceElement()` method must be an instance of the super-element (which will be "`this`"). The last three arguments are simply the arguments the super-element receives in it's `addXmlBody()` method.

The `VoiceElementResult` object returned is a small data structure class containing the exit state of the sub-element (if any), and whether the sub-element produced any VoiceXML. This last value is important because the only time a voice element is allowed to produce no VoiceXML is when it is visited for the last time. It returns an exit state and no VoiceXML so Call Services knows to visit the next voice element immediately rather than producing a VoiceXML page first and visiting the element after the browser makes the next request. If the sub-element does not return any VoiceXML, the super-element must either add its own VoiceXML content directly, visit another sub-element, or exit with an appropriate exit state. After the `createSubVoiceElement()` method executes, the `VMain` object will contain a complete VoiceXML page. If the `VoiceElementResult` object indicates that the sub-element returned no VoiceXML, `VMain` will contain an incomplete VoiceXML page.

Building super-elements in this manner is a tricky process. The developer must be aware of the following:

- Since the sub-element executes in the super-element's context, it shares the same configuration. The consequence of this is that the super-element must ensure that its configuration contains all the settings and audio groups expected by the sub-element. If it does not, the super-element must modify its configuration object *before* calling the `createSubVoiceElement()` method or the sub-element may throw an exception. This issue is compounded when the super-element encapsulates multiple sub-elements that have settings or audio groups with the same real name. The super-element's configuration would need to define separate settings for each sub-element's settings and then rename them appropriately in order for the sub-element to understand it.

- The scratch space for a super-element is *shared* with all sub-elements. The consequence for this is that any scratch data stored by the super-element must not conflict with the scratch data stored by the sub-element. Many times, though, voice elements do not publicize the names of the scratch data used. A good bet would be to name the super-element scratch data uniquely so there will not be any conflicts. Another consequence is that the super-element must be responsible for clearing the scratch data when an sub-element is complete. Call

Services automatically clears the scratch data, but only for the elements it is aware of, which in this case is only the super-element.

- The element data namespace is shared across the super-element and all sub-elements. Again, Call Services uses a separate namespace for each element it is aware of, which means the super-element. The super-element must ensure that element data created by one sub-element does not overwrite the element data created by another sub-element.

- The super-element must handle all the "internal call flow" between sub-elements. Essentially, the super-element must perform the tasks Call Services normally performs to handle the call flow between sub-elements. This means keeping track of state, managing the exit states of sub-elements, and knowing when to leave the super-element with its own exit states.

Even though creating super-elements can be a tedious and potentially error-prone process, it may still be preferable over creating a new voice element from scratch.

*Configuration Methods*

In addition to the existing configuration methods defined in `ElementBase`, two additional methods are required for a voice element class to implement. These methods deal with audio, a feature unique to voice element configurations. The values returned by these methods are used by Builder for Call Studio to render the contents of the Audio tab of a voice element's configuration.

```
HashMap getAudioGroups()
```

Unlike element settings, which are defined in a simple one-dimensional array, audio groups are combined into sets. This is done to facilitate organization and ease of use when configuring the voice element in the Builder. When right-clicking on the Audio Groups folder in the Element Configuration Pane, the dropdown menu lists all the audio group sets. The designer can then choose an audio group within a set by selecting the appropriate name from the submenu. This method is used to return the audio groups for the element and the sets they belong to.

A single audio group is contained within a single `AudioGroup` instance (`AudioGroup` is found in the `com.audium.server.voiceElement` package). The object encapsulates the audio group's real name, display name, and description. The display name is shown within the dropdown menu in the Builder, the real name is used for all other situations. The audio group description is displayed as a tool tip when the cursor points to the audio group in the Element Configuration Pane.

The `AudioGroup` class also defines setting dependencies. As with element settings, the appearance of audio groups can also depend on the values of certain settings. Again, this exists to simplify configuring a complex voice element in the Builder. For example, an audio group that introduces a confirmation menu would not be necessary if a setting determining if a confirmation should exist is turned off. See the previous section describing the `getSettings()` method for a

full description of how dependencies work. Configuring audio groups to depend on settings works the same way as configuring settings to depend on other settings.

The audio groups are arranged in sets by storing them in a `HashMap` Java collection. The keys of the `HashMap` are the names of the audio group sets and the values of the `HashMap` are arrays of `AudioGroup` instances that belong to their set. The array represents the audio groups to display within that set. The order specified in the array is the order they appear in the Builder.

```
String[] getAudioGroupDisplayOrder()
```

This method is provided for the developer to determine the order in which the sets of audio groups appear in the Builder. The `String` array is a list of set names in the order in which they should appear in the dropdown menu. The values must match those used as keys to the `HashMap` returned by the `getAudioGroups()` method.

### Interaction Logging

As listed in one of the recommended guidelines, interaction logging is a Universal Edition-defined mechanism for voice elements to record to the Call Services logs the actions of a caller when they are interacting with the voice browser. Many voice browsers have the ability to record detailed logs of a phone call and a caller's interaction with a VoiceXML page. These logs, however, are stored on the voice browser, which may be inaccessible or at least difficult to access. Additionally, logs on the Call Services side and the browser side would need to be cross-referenced in order to determine what happened in a particular call. It would be desirable to store all pertinent information in one place, which is what the interaction logging attempts to do. Interaction logging is stored in the application's activity log, which already stores other information such as the ANI and DNIS, what elements were visited in the call with what exit states, element data, etc. While it does not have the fine level of detail that a browser log can provide, interaction logging is sufficient for an administrator to determine what happened in a call or a designer to calculate call statistics to aid in improving the application.

Since VoiceXML is used to tell the voice browser how to interact with the caller, it must also be used to keep track of the caller's activity. The mechanism Universal Edition uses to do this is to create a single VoiceXML variable in which all the interaction logging data is stored. As new data becomes available, it is appended to the variable using a convention to delineate the data. This variable is automatically defined in the Call Services-generated root document, all the voice element developer needs to do is append content to it. The variable is named `audium_vxmlLog`, though a voice element developer should not use this name directly in their code. Instead, they should use the Java constant `VXML_LOG_VARIABLE_NAME`, defined in `VoiceElementBase` to refer to the variable. The reason for this is that the VoiceXML variable name is subject to change while the Java constant name is not. The Java constant will always contain the name of this variable so the developer need not recompile their code if the VoiceXML variable name changes.

Once a VoiceXML page is visited and the `audium_vxmlLog` variable is filled with content, it must be passed back to Call Services for parsing to place in the activity log. This means that

every submit back to Call Services must include this variable as an argument. If the voice element uses the `getSubmitURL()` method to obtain the submit URL, `audium_vxmlLog` must be explicitly added to the argument list in order to log correctly. Another advantage of using the `getSubmitVAction()` methods are that they already add this variable to the submit.

The convention used to for interaction logging appends the action name, description, and timestamp using the following format:

"`|||ACTION$$$VALUE^^^ELAPSED`"

where:

- ACTION is the name of the action. There are currently seven different actions that can be logged (the names are exactly as listed, all lowercase):
  - **audio_group.** This is used to indicate that the caller heard an audio group play. The value is the name of the audio group.
  - **inputmode.** This is used to report how the caller entered their data, whether by voice or by DTMF keypresses. The value is stored in the inputmode VoiceXML shadow variable.
  - **utterance.** This is used to report the utterance as recorded by the speech recognition engine (available using a VoiceXML shadow variable). The value is the actual utterance.
  - **interpretation.** This is used to report the interpretation as recorded by the speech recognition engine. The value is the actual interpretation.
  - **confidence.** This is used to report the confidence as recorded by the speech recognition engine (available using a VoiceXML shadow variable). The value is the confidence value.
  - **nomatch.** This is used to indicate the caller entered the wrong information, incurring a nomatch event. The value is the count of the nomatch event.
  - **noinput.** This is used to indicate the caller entered nothing, incurring a noinput event. The value is the count of the noinput event.
- VALUE is the value (description) to put in the log.
- ELAPSED is the number of milliseconds since the VoiceXML page was entered. This is required in order to keep an accurate timestamp in the activity log. Luckily, the Call Services-generated root document provides a Javascript function named `application.getElapsedTime(START_TIME)` that returns the number of milliseconds elapsed since the time specified in `START_TIME`. A VoiceXML variable is declared in the root document that holds the time the VoiceXML page was entered and should be passed as input to this method. The variable name is `audium_element_start_time_millisecs`, though just as with `audium_vxmlLog`, a Java constant defined in `VoiceElementBase` named `ELEMENT_START_TIME_MILLISECS` should be used to refer to this variable.

Interaction logging is best explained though several examples. The first example wishes to add to the interaction log the fact that the *xyz* audio group was played. The VoiceXML necessary to produce this logging as per the specified convention would be:

```
<assign name="audium_vxmlLog" expr="audium_vxmlLog+'|||audio_group$$$xyz^^^' +
application.getElapsedTime(audium_element_start_time_millisecs)"/>
```

Note that the `expr` attribute of `<assign>` is used because the value is actually an expression that concatenates various strings together. First, the `audium_vxmlLog` variable must be listed because we are appending new data to it. We append the string listing the audio group action and the name of the audio group, all contained within single quotes because this is a string literal. The final part is the Javascript, which cannot be within single quotes.

To do this within a voice element, one would have to use the `VAction` object (since it handles the `<assign>` tag) like this:

```
VAction log = VAction.getNew(pref, VAction.ASSIGN,
                             VXML_LOG_VARIABLE_NAME,
                             VXML_LOG_VARIABLE_NAME + "+'|||audio_group" +
                             "$$$xyz^^^' + application.getElapsedTime(" +
                             ELEMENT_START_TIME_MILLISECS + ")",
                             VAction.WITHOUT_QUOTES);
```

Note that the `audium_vxmlLog` and `audium_element_start_time_millisecs` variables are not mentioned by name, the `VXML_LOG_VARIABLE_NAME` and `ELEMENT_START_TIME_MILLISECS` Java constants are used instead. Where Java constants are used, they must appear outside the double quotes defining the string. Also note that `pref` is expected to be a valid `VPreference` object.

In a more complex example, the utterance of a field named *xyz* is to be appended to the log. The utterance is determined by a VoiceXML shadow variable. The VoiceXML would look like:

```
 <assign name="audium_vxmlLog" expr="audium_vxmlLog + '|||utterance$$$' +
xyz.$utterance + '^^^' + application.getElapsedTime(
audium_element_start_time_millisecs)"/>
```

and the `VAction` object would be configured like:

```
VAction log = VAction.getNew(pref, VAction.ASSIGN,
                             VXML_LOG_VARIABLE_NAME,
                             VXML_LOG_VARIABLE_NAME + "+'|||utterance" +
                             "$$$' + xyz.$utterance + '^^^' + " +
                             "application.getElapsedTime(" +
                             ELEMENT_START_TIME_MILLISECS + ")",
                             VAction.WITHOUT_QUOTES);
```

See Chapter 5 in the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more detail about the different logs Call Services records and the data that can appear in the logs.

# Chapter 10: Application Start Classes

Application start classes are unlike most components in that they are not related to a call session. Application start classes are instead associated with a particular application and are executed when the application itself is initialized or updated. Currently, only the Java API can construct code to run when an application starts.

An application can define in its application settings any number of application start classes. Call Services will execute the classes sequentially in the order they appear in the application's settings. By conforming to this order, a developer can create an application start class that stores information that can then be referenced by subsequent start of application classes.

The application settings can specify that an error on a particular application start class will cancel the application's deployment. This is an optional attribute and by default the system will not allow an error in one of these classes to stop the application deployment. If set, an error encountered in the class will stop the application from being deployed, an error message will appear in the application server console, and an error event will be thrown to be logged by any error loggers. This attribute is provided should an application require its application start class to run without error for calls the application to succeed.

There are four situations where the application start class is run:

- The application server is launched. Call Services is configured to begin initializing applications once it is loaded by the application server. This process triggers each applications' start classes to be run.
- The Call Services web application is restarted. Most application servers provide the ability to restart just a certain web application running within it rather than restarting the entire application server. The act of restarting the web application that defines the Call Services will prompt it to start the application loading process just like an application server restart.
- An application is deployed after the Call Services has started. Using the deployment administration scripts, an application can be deployed while the system is actively handling calls to other applications. When the application is loaded, the application start classes will be run.
- An application is updated. The process of updating an application prompts Call Services to create a new instance of the application in memory, while keeping the old instance in memory long enough for all existing callers to complete their calls. The new application must initialize itself, including calling all application start classes.

An application start class only has access to the Global API, which allows for the creation of global and application data. It does not have access to the Session API because it is not run within a call session and is associated only with an application, not a call.

The main purpose for an application start class would be to prepare information that would then be used by call-specific components, especially if the setup process takes a long time to run. For example, if an application has elements that are written to access a backend database or mainframe system, they could initiate connections to that system each time they need to. This, however, would incur overhead in initiating the connection each time. An alternative would be to write an application start class that will open up the connection to the database or mainframe and perform all necessary setup. The class could take as long as necessary to run because the application is being initialized and is not actively taking calls. The application start class could then store the connection in application data that the elements could then access when needed. This solution incurs overhead at the best time, during application initialization, and eliminates it at the worst time, during a call.

The application start class action is built by implementing the Universal Edition class `StartApplicationInterface` found in the `com.audium.server.proxy` package. It contains a single method named `onStartApplication` that is the execution method for the application start class. This method receives a single argument, an instance of `ApplicationStartAPI`. This class belongs to the Global API and is used to access and create application data and global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on application and global data). The method does not have a return value. It is expected that should an unrecoverable error occur, the application start class will throw an `AudiumException`.

# Chapter 11: Application End Classes

Application end classes are unlike most components in that they are not related to a call session. Application end classes are instead associated with a particular application and are executed when the application itself is taken down or updated. Currently, only the Java API can construct code to run when an application starts.

An application can define in its application settings any number of application end classes. Call Services will execute the classes sequentially in the order they appear in the application's settings. By conforming to this order, a developer can create an application end class that stores information that can then be referenced by subsequent end of application classes.

Unlike application start classes, an error in one application end class does not cancel the application release, once started the application will be released no matter what occurs. Should an error occur in an application end class, an error event will be thrown for any error loggers to report.

There are four situations where the application end class is run:

- The application server is shut down. Call Services is configured to shut down all of its own operations as well as shut down each individual application by running their application end classes.
- The Call Services web application is restarted. Most application servers provide the ability to restart just a certain web application running within it rather than restarting the entire application server. The act of restarting the web application that defines Call Services will prompt it to initiate the application unloading process, including calling its application end classes just like an application server restart.
- An application is released after the Call Services has started. Using the release administration scripts, an application can be released while the system is actively handling calls to other applications. Call Services will first run all application end classes for the application and then release it from memory.
- An application is updated. The process of updating an application prompts Call Services to create a new instance of the application in memory, while keeping the old instance in memory long enough for all existing callers to complete their calls. Once all calls have completed with the old instance, that instance's application end classes are run.

An application end class only has access to the Global API, which allows for the creation of global and application data. It does not have access to the Session API because it is not run within a call session and is associated only with an application, not a call.

The main purpose for an application end class would be to perform cleanup operations for an application. Typically this would involve closing database connections or files that were opened by an application start class or by code run within calls to the application. Application end

classes have the ability to create application or global data, though the concept of creating data right before the application is released would seem pointless. The one situation where this would be useful would be to set data that a subsequent application end class could use. For example, an application start class could open up a database connection and store it in global data so that all applications deployed on Call Services could utilize the connction. This connection would need to be closed, but if there are multiple applications with multiple application end classes, the desire would be to close the connection by the last application to be released, in case the application end classes need to use the connection. Each application start classes could increment an application count value stored in global data that the application end classes would decrement. The application end class that yielded a zero would know that it was the last application released and so close the database connection.

The application end class action is built by implementing the Universal Edition class `EndApplicationInterface` found in the `com.audium.server.proxy` package. It contains a single method named `onEndApplication` that is the execution method for the application end class. This method receives a single argument, an instance of `ApplicationEndAPI`. This class belongs to the Global API and is used to access and create application data and global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on application and global data). The method does not have a return value. It is expected that should an unrecoverable error occur, the application end class will throw an `AudiumException`.

# Chapter 12: Say It Smart Plugins

Similar to the ability for a developer to create custom elements, a developer can create their own Say It Smart plugins. A developer can produce plugins that handle brand new Say It Smart types as well as plugins that extend the functionality of existing Say it Smart plugins. Due to the integration requirements for Say It Smart plugins, they can be built only by using the Java API.

Custom Say It Smart plugins are integrated into both Call Services and Call Studio as easily as Universal Edition Say It Smart plugins are. They can be deployed for a specific application or shared across all applications and are configured in the Studio in the same manner Universal Edition Say It Smart plugins are. With such seamless integration and effortless deployment, a developer can, over time, create entire libraries of custom Say It Smart plugins to use for their voice applications or potentially for resale.

This chapter describes in detail how to create custom Say It Smart plugins and integrate them into both Call Studio and Call Services.

## Design

Say It Smart plugins were designed to be very simple to build, extend, and deploy. Much of the design mirrors that of custom configurable elements, though Say It Smart plugins are simpler and set far fewer restrictions. Their sole purpose is to take input representing formatted data and convert it into a list of pre-recorded audio files with TTS backups and pauses if desired.

Similar to configurable elements, a Say It Smart plugin is constructed by creating a Java class that extends an abstract base class, `SayItSmartBase`. The base class defines abstract methods that must be implemented by the plugin to describe how Builder for Call Studio displays the plugin. A plugin that is to appear in the Builder must implement a Java marker interface named `SayItSmartPlugin`. Unlike elements, though, Say It Smart plugins have two execution methods, one for converting data to a set of audio files with TTS backups, and the other for converting the data using TTS only. These execution methods may throw a `SayItSmartException` that is used to indicate the inability of the plugin to convert the data passed to it.

The configuration of a Say It Smart plugin involves four options. The first is the Say It Smart type (such as phone number or date). Each type must be defined in a separate plugin class. The second option is the chosen input format. Input formats list how to expect the input data to arrive (such as a date with just the month and year or a date with the month, day and year). A plugin defines all the input formats it supports. The next option is the chosen output format. Output formats list how to render the converted data (such as reading back a time where 12:00AM is read back as “noon” as opposed to “12 A M”). Output formats are dependent on input formats, so when the input format changes, the list of output formats available changes accordingly. The plugin defines these dependencies using a configuration method. The final option is the fileset. Filesets determine what group of audio files are used to render the same data (such as one that reads back a better-sounding number by requiring more audio files). Filesets are dependent on

output formats, so when the output format changes, the list of filesets available changes accordingly. The plugin defines these dependencies using a configuration method. Note that a fileset deals with audio files so does not apply when the Say It Smart value is rendered in TTS only.

As usually occurs with components configured in Builder for Call Studio, each type, input format, output format, and fileset has a real name, display name, and description. The display name is shown in the Builder in dropdown menus. The real name is used everywhere else. This design allows the Say It Smart plugin developer to use a display name that visually represents the appropriate information but choose a real name that is small, easy to remember, and will not change. As long as the real name stays the same, the developer can change the display name of any component without affecting backwards compatibility. At this point, the descriptions are not displayed in the Builder and are there for future compatibility.

All Java classes related to Say It Smart plugins are found in the `com.audium.server.sayitsmart` package.

Note that unlike elements, Say It Smart plugin classes are instantiated as needed. This means that the developer is free to use static, member, and local variables as they would expect. It is still recommended to avoid using static variables in Say It Smart plugin classes unless they are static final because static variables will be reset whenever the application is updated.

## Execution Methods

```
SayItSmartContent convertToFiles(Object data, String inputFormat,
                                 String outputFormat, String fileset)
```

This is the execution method for converting data to a list of audio files with TTS backups. The first argument is the data to convert. This data can be any Java object and it is up to the plugin to cast to the appropriate type (usually depending on what the input format is). Most of the time, though, it will be considered a `String`. The next three arguments list the real names of the input format, output format, and fileset specified in the voice element configuration (either statically in Builder for Call Studio or dynamically).

The method returns an instance of `SayItSmartContent`. This class encapsulates any number of audio filenames, their TTS backups, and pauses to insert in the playback. The execution method must create an instance of this class, add the desired content to it, and return it. Call Services then reads this content to generate the VoiceXML for the Say It Smart audio item in the audio group. Note that the path and file type options available in the Say It Smart configuration in the Builder are not passed here as they are added *automatically* by Call Services once the plugin has converted the data. The plugin should produce just the audio file names without any paths or extensions.

Note that while the option exists, the plugin need not include TTS backups for the audio files. They are used as a backup in case the audio file is not found or is corrupted. Not including a

transcript (by making it `null`) would mean that if the audio file could not be found, the application would prematurely end with an error.

The developer can throw a `SayItSmartException` in this method indicating the data passed to the plugin could not be converted using the specified configuration. This would most likely end the call prematurely so the exception should be thrown only when the Say It Smart plugin cannot do what it is supposed to do and is unable to recover.

`convertToFiles()` is abstract, meaning every Say It Smart plugin must implement this method.

```
SayItSmartContent convertToTTS(Object data, String inputFormat,
                               String outputFormat, String fileset)
```

This is the execution method for converting data to a TTS string. The arguments are identical in this method as the `convertToFiles()` method. Note that even though the fileset option technically does not apply here, it is included in case the plugin does alter the TTS output based on fileset information.

The method must still return a `SayItSmartContent` object since the plugin may still decide to play back the TTS content with pauses at certain points. In this case, the filename value would be set to `null`.

Note that `convertToTTS()` is not abstract, `SayItSmartBase` actually provides a default implementation of this method. The default implementation simply calls the `converToFiles()` method, combines the TTS transcripts it receives and returns a new `SayItSmartContent` object with just those transcripts and any pauses. The plugin need only provide its own implementation of this method if the desired behavior is different.

The developer can throw a `SayItSmartException` in this method indicating the data passed to the plugin could not be converted using the specified configuration. This would most likely end the call prematurely so the exception should be thrown only when the Say It Smart plugin cannot do what it is supposed to do and is unable to recover.

## Configuration Methods

```
SayItSmartDisplay getDisplayInformation()
```

This method is used to specify display information for Builder for Call Studio to render this plugin's configuration. The `SayItSmartDisplay` object lists the plugin type, input formats, output format, and filesets. Each type, input format, output format, and fileset must have a real name, display name, and description. The relationship between the input formats, output formats, and filesets are defined by the methods listed below.

This method throws a `SayItSmartException` if the `SayItSmartDisplay` object is not configured correctly.

`getDisplayInformation()` is abstract, meaning every Say It Smart plugin must implement this method.

`SayItSmartDependency getFormatDependencies()`

As described in the design section, a plugin's output formats are dependent on the input formats. This method defines those relationships. The `SayItSmartDependency` object is used to specify which output formats listed in the `getDisplayInformation()` method apply to which input formats.

The `SayItSmartDependency` class is defined generically with the concepts of "parents" and "children". For this method, the parents are input formats and the children are output formats. The developer simply uses the `addParent()` methods to add a new input format and the output formats that depend on it. The `addChild()` and `addChildren()` methods are used to add additional output formats that are dependent on the specified parent. This is done until all input formats are listed and all the output formats that depend on it are listed. Remember that the names to use here are the *real names*, the display names are used only by the Builder. All the input formats and output formats defined in the `getDisplayInformation()` method must be mapped here.

This method throws a `SayItSmartException` if the `SayItSmartDependency` object is not configured correctly.

`getFormatDependencies()` is abstract, meaning every Say It Smart plugin must implement this method.

`SayItSmartDependency getFilesetDependencies()`

This method is identical to the `getFormatDependencies()` method except it defines which filesets are dependent on which output formats. This method also returns a `SayItSmartDependency` object, except this time the "parent" is an output format, and the "child" is a fileset. Again, the real names of the output formats and filesets must be used here. All the output formats and filesets defined in the `getDisplayInformation()` method must be mapped here.

This method throws a `SayItSmartException` if the `SayItSmartDependency` object is not configured correctly.

`getFilesetDependencies()` is abstract, meaning every Say It Smart plugin must implement this method.

## Utility Methods

These utility methods are provided to aid Say It Smart plugin developers. Their use is optional.

```
void validateArguments(Object data, String inputFormat,
                       String outputFormat, String fileset)
```

This method validates each argument and throws a `SayItSmartException` if there is an error with one of them. The exception is thrown if `data` is `null`, or `inputFormat`, `outputFormat`, or `fileset` does not correspond to the ones defined in the `getDisplayInformation()` method. Additionally, an exception is thrown if `outputFormat` does not depend on `inputFormat` or `fileset` does not depend on the `outputFormat`. The error message lists all the appropriate options available.

All Universal Edition Say It Smart plugins call this method in the first line of their execution methods.

```
String createError(String header, String[] options, String footer)
```

This utility method is a shortcut for creating an error message when some value does not match one of an array of different possible values. The error conveys that and lists all the values the data can be. Many Universal Edition Say It Smart plugins provide an array of `String`s containing formats that they will accept input data to arrive in and this method is used when the input data does not arrive in a supported format. The output string starts with “SayItSmart Error – PLUGIN” where PLUGIN is the name of the plugin type as returned by the `getDsplayInformation()` method. The header then appears, followed by a comma-delimited list of correct values (with the last option following an “and”) followed by the footer.

This is best explained with an example. A Say It Smart plugin named "Foo" has the input formats "a", "b", "c", and "d". If the input format passed was "e", this method can be called to create the error message. Calling the message like so:

```
createError("Only the input formats supported are: ", new String[] {"a", "b",
"c", "d"}, " Please use a supported format.").
```

will produce the following error message:

```
SayItSmart Error - Foo: Only the input formats supported are: "a", "b", "c",
and "d". Please use a supported format.
```

```
String[] getFilesetContents(String fileset)
```

This utility method returns an array of `String`s containing the filenames required for the fileset passed as input. This method can be used to determine what audio files need to be recorded to fully support a fileset for a Say It Smart plugin. At this point, this method exists only for informational purposes, it is not accessed by either Builder for Call Studio or Call Services.

# Chapter 13: Loggers

The mechanism used by Call Services to record information about global administrations, errors caused by sub-systems, activities taken by callers to deployed applications or administrators is by using loggers. Loggers collect this information and can do any number of tasks with it, from aggregating it for reporting purposes, to sending that information to external systems for managing, to simply storing the information in log files. A developer can produce a logger to supplement or replace the functionality the loggers included with Call Services provide. Due to the complexity of integrating with Call Services, loggers can be built only by using the Java API.

This chapter describes in detail how to create custom loggers and integrate them with Call Services. For more detail on the loggers included with Call Services, refer to the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 5: Call Services Logging.

## Call Services Logging Design

Before discussing the design of an individual logger, it is warranted to introduce the design of the logging mechanism within Call Services. Knowledge of this design will help the logger developer create loggers that work harmoniously with the system.

The mechanism by which information is passed to a logger is through an event object. This object will encapsulate information about what just occurred, including a timestamp. Event objects are created by Call Services in many different situations that belong to three levels: related to global activities, related to an application and related to a call session. The event object will contain all the information accessible to the logger for the particular event as well as information about the environment. For global level events, the environment varies. Some global events such as global request error event, it provides information about the requested URI, HTTP parameters and headers whereas for others such as global warning event which is used to inform user about a undesirable situation, environment is not required. For application-level events such as an administration event, the environment consists of application data and global data (not call data since this event is not affiliated with a call). For call-level events such as a call start event, the environment consists of information about the call such as the ANI, element and session data, default audio path, etc. Since the purpose of a logger is to report information, loggers are limited to obtaining environment information and cannot change any value. Loggers may still need to store session-related information for its purposes so to accommodate this Call Services provides loggers “scratch” data that is stored in the session and will be available to the logger only for those events associated with the session.

Call Services comes with three built-in global loggers for administration, call and errors. Custom global loggers that utilize Global logger APIS can also be added. Each global logger is configurable through an xml configuration file or by defaults.

An application designer can define any number of loggers to use in an application. The designer defines logger instances with unique names and can pass configurations to them. They can even define multiple instances of the same logger class with a different configuration for each. Call Services will then create a separate instance of the logger class for each instance referenced in the application's settings. The instances are created when the individual applications are initialized and are maintained for the lifetime of the application. All events for the application, both application-level events as well as call-level events, are handled by this single instance. Note that an instance variable in the logger class will allow it to maintain information that spans calls.

A logger is expected to "register" the events it wishes to act on. This is done on logger initialization. When Call Services loads, it initializes all the global loggers and the loggers referenced for an application the records which events each will act on. When a situation occurs that would constitute an event, Call Services checks to see if any loggers will act on the event and if so, will create the event object and pass it to the loggers. This registration mechanism allows Call Services to save the overhead in creating an event if no loggers will act on it. Additionally, should there be multiple loggers acting on an event, only one event object is created and passed to all of them.

In order to ensure that no call be held up due to logging activities, the entire Call Services logging mechanism is fully multi-threaded. The only logging-related activity that an HTTP request thread provided by the application server performs is creating an event object and adding it to a queue. It does not actually handle the logging of that event and once it has added the event to the queue, it continues with the call. A separate, constantly running asynchronous process, called the Logger Manager, will process the events in the queue. This allows the logging process to act independently from the process of handling a call and so will not directly affect the performance of the system.

In order to ensure that no logger be held up due to the activities of another logger (or the same logger) while handling an event, a second layer of threads are used. While the Logger Manager handles the queue, when it is time for a logger to be given the event to handle, this is itself done in a separate thread. The Logger Manager therefore is responsible only for managing the queue of events and spawning threads for loggers to handle them. This ensures that a logger that takes a long time to handle an event does not hold up the logging for the same or other applications, it only hold up the thread in which it is running. To efficiently use created thread, a thread pool has been introduced in this context. To avoid creating too many threads when under load, the maximum number of allowable threads in the pool can be configured in the global configuration file named `global_config.xml` found in the `conf` directory of Call Services by editing the contents of the `<maximum_thread_pool_size>` tag. For a thread to be reused after it is done with the current task, the `<keep_alive_time> tag from the same configuration file can be set.` When all the allowable threads are taken, Call Services will not process the queue until a thread becomes available from the pool. Note that one of the consequences here is that the longer the events remain in the queue, the less "real-time" the logging will occur. Additionally, if the maximum thread pool size is made too low to handle a given call volume, the queue can

become very large and could eventually cause issues with memory and spiking CPU. Typically, though, a logger handles an event in a very short period of time, allowing a small number of threads to handle the events created by many times that number of simultaneous callers.

There are times when the true asynchronous nature of the logging design works against the developer. The tasks done by a logger can take a variable amount of time to complete so there is no guarantee when a call event will be handled. This is by design, and for a logger that simply records events that are then sorted by timestamp later, this is not a problem. A logger that requires more context, though, could encounter issues. For example, if a logger needed to note when a call was received so that an event that occurred later on in the call could be handled correctly, problems could be encountered because there would be no guarantee that the events would be handled in the same order they occurred within the call. To remedy this situation while keeping the design unfettered, it is possible to specify that Call Services pass a logger instance events in the same order they occurred in a call. With this option on, the logger developer can be assured that the events for a call would not be handled out of order. In fact, the Activity Logger included with Call Services has this requirement. The penalty for this requirement, however, is a loss of some of the true asynchronous nature of the system as there will now be situations where events that are ready to be handled must wait for a previous event to be handled by the logger. If a logger hung while handling one event, the queue would forever contain the events that occurred after it in the call, and that call session would not be fully logged.

Some of the conclusions that can be deduced from the Call Services logging design can be summarized in some best practices:

- A logger developer need not worry about the time taken by the logger to handle an event as it will have no bearing on the performance of the call. With that said, the developer must also be aware of the expected call volume and ensure that the logger not take so long as to use up the event threads faster than they can be handled.

- Loggers work under a multi-threaded environment and the logger developer must understand how to code with this in mind. A single logger class can be handling events for many calls and so it must manage internal and external resources in a synchronized manner to prevent non-deterministic behavior.

- When possible, design the logger so that it does not rely on events within a call being passed to it in the order in which they occurred in the call. Doing so will maximize performance due to being able to handle events whenever they occur. Should the logger be unable to do so, require that the enforce call event order option be turned on for the logger.

# Logger Design

Similar to configurable elements, a logger is constructed by creating a Java class that extends an abstract base class, GlobalLoggerBase or `ApplicationLoggerBase which in turn extends from LoggerBase class`. The base classes define abstract methods that must be implemented

by the logger. Global loggers can be categorized into three types: administration, call and errors. Each type handles different global events. An application logger must implement a Java marker interface named `LoggerPlugin` to allow Builder for Call Studio to recognize it as a valid logger. Loggers have methods for initialization and destruction as well as an execution method to call when an event is to be handled. These methods may throw an `EventException` to indicate an error in the logger.

All Java classes related to loggers are found in the `com.audium.server.logger` package while the logger event classes are found in the `com.audium.server.logger.events` package.

## Global Logger Methods

```
void initialize(File configFile, LoggerGlobalAPI api)
```

This method is called by Call Services when a new logger instance is created. This occurs in two different situations: the application server starts up, Call Services web application is restarted.

The global logger designer has optionally included a reference to a configuration file for the logger which is passed here as a `File` object. If no configuration file reference was specified in the conf directory, this object will be `null`.

The method also receives a `LoggerGlobalAPI` object, which is used to access the GlobalAPI. The GlobalAPI provides access to global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on global data).

Aside from initializing the logger, this method is also responsible for configuring which events the logger instance is to handle. This is done by setting the value of the member variable `eventsToListenFor`. This variable is a `HashSet` that must contain all the event IDs that the logger listens for. Call Services accesses this `HashSet` to determine if a new event should be sent to the logger. When an event occurs, `eventsToListenFor` is accessed to see if the event ID can be found. If so, the logger will be notified of the event. The IDs for the events are defined in the interface `IEventIDs. GlobalLoggerBase` implements this interface so the event IDs are available directly within the logger class.

```
void destroy(LoggerGlobalAPI api)
```

This method is called in two different situations: the application server is shut down, Call Services web application is restarted.

The method also receives a `LoggerGlobalAPI` object, which is used to access the GlobalAPI. The GlobalAPI provides access to global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on global data).

The purpose of this method is to give the logger the opportunity to perform clean up operations. Typically, this would involve closing database connections or files that were opened by the logger in its initialize method or while handling calls. A logger that does not have that requirement must still implement the destroy method but can define an empty implementation.

```
void log(GlobalEvent event)
```

This method is the execution method for the logger and is called by Call Services when a new event has occurred that the logger must handle. This method is called only if Call Services has found the event's ID in the `eventsToListenFor HashSet`.

Only one object is passed to this method, a Global`Event`. This object encapsulates all the information about the event and provides access to other environment information depending on the type of event. `GlobalEvent` is a base class for all global level events and the logger will typically check for the event type and cast to the appropriate event to get its information. All event classes are found in the `com.audium.server.logger.events` package.

Figure 13-2 shows the class hierarchy for all events defined for both global and application levels:

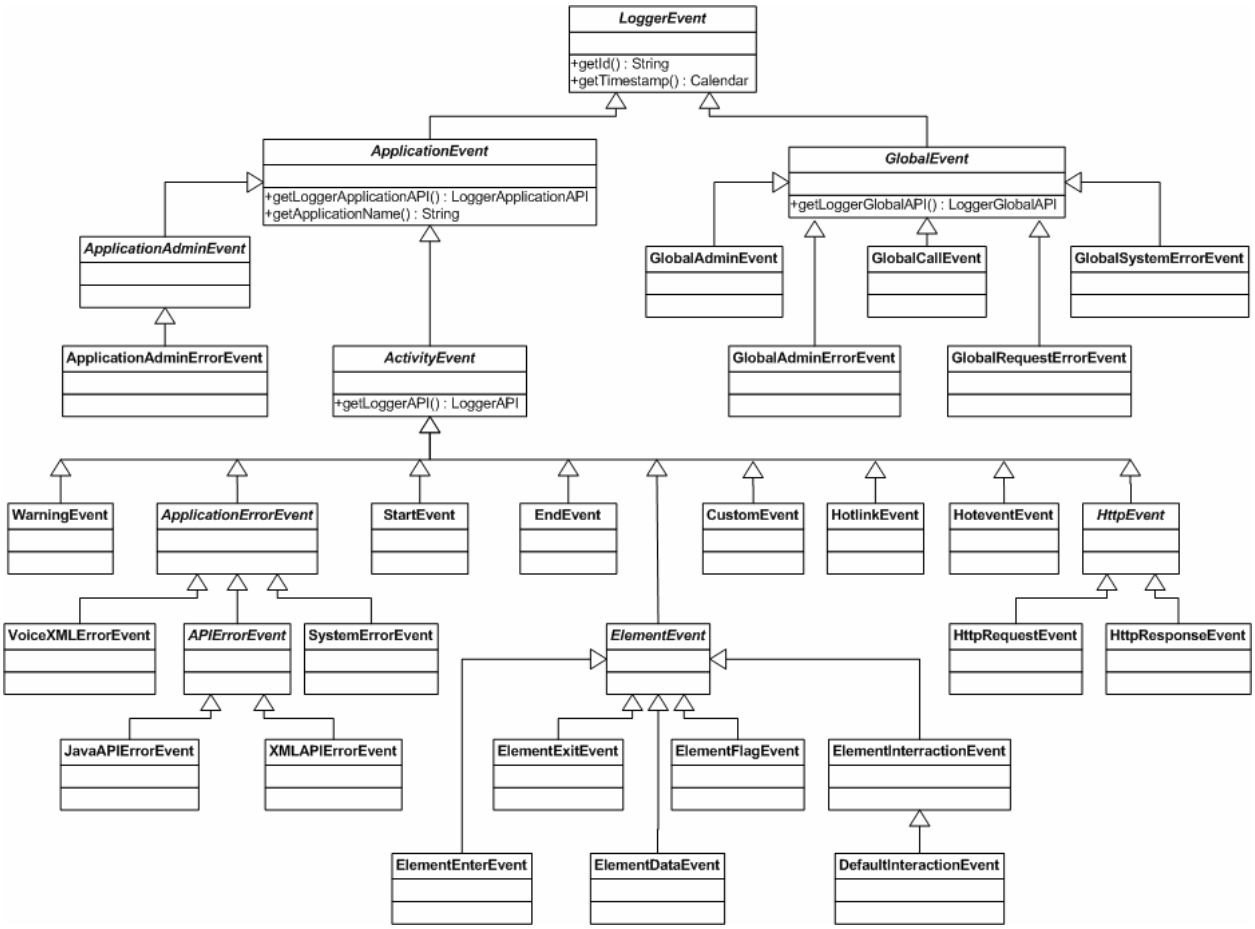

*Figure 13-1*

Notes on events:

- `GlobalEvent and ApplicationEvent` both extend the generic event class `LoggerEvent`. In this structure, new event types in the future can be added without affecting the existing class hierarchy.
- All events have an ID, obtained by calling the `getID` method.
- Global events are structured in a straightforward way. Most of the events such as global system error event associate with a message to describe event details.

```
void doPreLogActivity(GlobalEvent event)
void doPostLogActivity(GlobalEvent event)
```

These two methods can be optionally overridden to perform any activity desired before and after the `log` method is called. By default these methods are defined in the GlobalLoggerBase class to do nothing.

## Application Logger Methods

```
void initialize(File configFile, LoggerApplicationAPI api)
```

This method is called by Call Services when a new logger instance is created. This occurs in four different situations: the application server starts up, Call Services web application is restarted, the application the logger instance belongs to is deployed after the application server had started up, and the application is updated. These situations are the same as those for when the application start class is called (see Chapter 10: Application Start Classes for more on these situations).

The application designer has optionally included a reference to a configuration file for the logger which is passed here as a `File` object. If no configuration file reference was specified in the application settings, this object will be `null`.

The method also receives a `LoggerApplicationAPI` object, which is used to access the GlobalAPI. The GlobalAPI provides access to application data and global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on application and global data).

Aside from initializing the logger, this method is also responsible for configuring which events the logger instance is to handle. This is done by setting the value of the member variable `eventsToListenFor`. This variable is a `HashSet` that must contain all the event IDs that the logger listens for. Call Services accesses this `HashSet` to determine if a new event should be sent to the logger. When an event occurs, `eventsToListenFor` is accessed to see if the event ID can

be found. If so, the logger will be notified of the event. The IDs for the events are defined in the interface `IEventIDs. ApplicationLoggerBase` implements this interface so the event IDs are available directly within the logger class.

```
void destroy(LoggerApplicationAPI api)
```

This method is called by Call Services when an application is released. This occurs in four different situations: the application server is shut down, Call Services web application is restarted, the application the logger instance belongs to is released, and the application is updated. These situations are the same as those for when the application end class is called (see Chapter 11: Application End Classes for more on these situations).

The method also receives a `LoggerApplicationAPI` object, which is used to access the GlobalAPI. The GlobalAPI provides access to application data and global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on application and global data).

The purpose of this method is to give the logger the opportunity to perform clean up operations. Typically, this would involve closing database connections or files that were opened by the logger in its initialize method or while handling calls. A logger that does not have that requirement must still implement the destroy method but can define an empty implementation.

```
void log(ApplicationEvent event)
```

This method is the execution method for the logger and is called by Call Services when a new event has occurred that the logger must handle. This method is called only if Call Services has found the event's ID in the `eventsToListenFor HashSet`.

Only one object is passed to this method, an `ApplicationEvent`. This object encapsulates all the information about the event and provides access to other environment information depending on the type of event. `ApplicationEvent` is a base class for all application-level events and the logger will typically check for the event type and cast to the appropriate event to get its information. All event classes are found in the `com.audium.server.logger.events` package.

For information about event class structure, please refer to Figure 13-2.

Notes on application events:

- All application events have access to the Global API to get application and global data (see the User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio Chapter 2 in the section entitled Variables for more on application and global data).
- Application events are defined at two levels: call-level, and application- level.
    - Call-level events extend from `ActivityEvent` and are associated with a particular call. As such, these events provide information about the call environment through the method `getLoggerAPI()`. The resulting `LoggerAPI` object is from the Session API (See Chapter

3: Session API for more) and provides read-only access to session information such as element and session data.

  - Application-level events are associated only with an application and not a call session. Currently, the only application-level events are the application administration event (`ApplicationAdminEvent`) that reports on administration activity performed on the application and the application administration error event (`ApplicationAdminErrorEvent`) that reports any errors encountered while performing administration activities.

- The events extending `ActivityEvent` are all the events that can occur in a call: a new call, a call ending, an element being entered, an element exiting, an element storing data, a flag element being triggered, an element interacting with the caller, a hotevent being activated, a hotlink being activated, a custom event caused by calling the `addToLog` method of the Session API or entering data in an element's configuration pane in Builder for Call Studio, a warning event, an error event, and an HTTP event signifying a new HTTP request made to Call Services.
- Error events are defined in a hierarchy to define different types of errors. This allows a logger developer to act on certain types of errors or to do different tasks based on the error type.
- Currently, when a voice element returns interaction data, Call Services sends loggers a `DefaultInteractionEvent`, which extends from `ElementInteractionEvent`. This designs will allow for different interaction content in the future without affecting the existing class hierarchy.

```
void doPreLogActivity(ApplicationEvent event)
void doPostLogActivity(ApplicationEvent event)
```

These two methods can be optionally overridden to perform any activity desired before and after the `log` method is called. By default these methods are defined in the ApplicationLoggerBase class to do nothing.

# Utility Methods

These utility methods provide important information to the logger.

```
String getLogFileDirectory()
```

This method returns the full path of the directory Call Services has provided this logger to store its logs, if necessary. On startup, Call Services creates a folder for each logger instance whose name is the logger instance name within the global log folder and the logs folder of the application, for global and application loggers respectively. This folder is for exclusive use of the logger referenced in the logger instance should it require it. Should the logger create log files, it should use this directory, unless the logger's configuration specified to log somewhere else.

Should the logger not use files, such as if it logs directly to a database, this method and the folder it references can be ignored.

```
String getApplicationName()
```

This method is only for application-level. It returns the name of the application that the logger instance belongs to.

```
String enforceCallEventOrder()
```

This method is only for application-level. It returns true if the logger instance has been configured to enforce the call event order, false otherwise. This is useful if the logger wishes to throw an error if the logger instance was not configured a certain way.

```
HashSet getEventsListeningFor()
```

This method returns the HashSet containing the event IDs that the logger is handling. This method is called by Call Services to determine if to send a new event to the logger. This methods simply returns the protected member variable eventsToListenFor defined in LoggerBase.

```
String getLoggerInstanceName()
```

This method returns the name of the logger instance as defined by logger designer or the application designer for the application.

```
int getLoggerType()
```

This method returns the type of the logger, and is for future use. Currently, it returns `APPLICATION_LOGGER or GLOBAL_LOGGER`.

# Chapter 14: Hotevents

Hotevents can only be produced through the Java API because they involve the use of the Universal Edition Voice Foundation Classes (VFCs), which are Java-only (see Appendix A: The Voice Foundation Classes for more on the VFCs).

When a new hotevent is added to the workspace in Builder for Call Studio, the full name of a Java class must be entered in the hotevent dialog box. This class, when executed, is expected to produce the VoiceXML to run when the event is triggered. The VoiceXML generated by this class is placed in the root document automatically generated by Call Services. Since the root document is cached by the voice browser, this class is executed only once per call, it is *not* called when the event is triggered and therefore does not have access to the Session API to obtain dynamic session information like the ANI, element and session data.

Following the standard design of the Java API, the hotevent class must implement a Java interface named `HoteventInterface` found in the `com.audium.server.proxy` package. The interface defines a single method `addEventBody` that is called when the call's root document is being generated. The method receives two VFC classes as arguments, a `VPreference` object and a `VEvent` object. The VoiceXML code to execute when the hotevent is triggered must be added to the `VEvent` object and the `VPreference` object is used to instantiate the VFC classes defining that VoiceXML. The method does not need to return anything as all content is encapsulated within the `VEvent` object passed by reference to the method.

# Chapter 15: On Error Notification

The error notification process can only be implemented using Java because when an error occurs, one desires the most reliable method for reporting that error. There is no guarantee an HTTP request to a URI could even be generated, a response received and the XML parsed without incurring another error.

The on error notification class is built in Java by implementing the class `GlobalErrorInterface` found in the `com.audium.server.proxy` package. It contains a single method named `doError` that acts as the execution method for the class. The method receives nine arguments containing information on the status of the application and Call Services at the time the error occurred. No API classes are passed to this method because accessing them may cause additional errors due to their complexity. Any of the arguments may be `null` if the data cannot be determined or the error is such that it is not related to a specific application.

The arguments are: the Call Services session ID (as a `String`), the name of the application (as a `String`), the ANI (as a `String`), the DNIS (as a `String`), the IIDIGITS (as a `String`), the UUI (as a `String`), an `ArrayList` of `String`s listing the elements visited in the call up to the time the error occurred, an `ArrayList` of the `String`s listing the exit states for each of the elements, and a `HashMap` containing the session data created up to the time the error occurred (the key of the `HashMap` is the name of the session data, and the value is the session data value).

The on error notification class must be deployed in the `common` directory of Call Services since classes placed there are shared across applications.

To configure Call Services to use this class if an error occurs, a file named `global_config.xml` found in the `conf` directory of Call Services must be used. This XML file contains a tag named `<error_class>` that should encapsulate the full Java name of this class (package name included). The changes will take effect only the next time the Java application server on which Call Services is installed is restarted.

# Chapter 16: Application Management API

Call Services now comes with application management facilities to provide more visibility into the runtime environment and better, more flexible control over the platform together with its components and services. As a result, a comprehensive OA&M feature set is necessitated to operate, administrate and manage the health of the platform and to provide statistics and performance measurement.

This chapter describes in detail how to the management server system has been designed and how to create custom management support based on the API provided with the Call Services platform.

## Design

All the OA&M features on Call Services are built with JMX management standards. Applications and components on Call Services are instrumented using Managed Beans (MBeans). MBeans expose their management interfaces, composed of attributes, operations and event notifications, through a JMX agent for remote management and monitoring.

Managed resources are categorized at levels of application, global configuration and command, and platform. Each level may facilitate three typical JMX managements: lookup and modify configuration, collect and avail application statistics, notify of state changes and erroneous conditions.

At the application level, administrators can operate the following:

- Get/set default audio path
- Get/set suspended audio
- Get/set session timeout
- Get gateway adapter
- Methods derived from loggers
- Application release
- Application resume
- Runtime status
- Application update
- Suspend/resume application
- Retrieve application administration history
- Get application data
- Set and remove application data
- Remove all application data
- Get application data names

The global level manages Call Services as a web application. It allows:

- Read logger event queue size
- Read/write maximum logger thread pool size
- Read/write minimum logger thread pool size
- Read/write logger thread keep alive time
- Read/write session invalidation delay
- Get global status
- Deploy all new applications. This command assumes all the deployable applications have been updated to the AudiumHome. All this command does is to deploy these applications.
- List all new applications. This command lists the names of all new applications (i.e., those which have yet to be deployed).
- Deploy new application. This command deploys the specified application. To retrieve a list of new application names, use “List all new applications” (above) first.
- Flush all old applications
- Update all applications
- Update common classes
- Suspend/resume Call Services
- Retrieve global administration history
- Get/set global data
- Get all global data names
- Remove all global data

At the same level, the management system also provides information regarding metrics collection:

- The total number of calls since Call Services starts up.
- Maximum and average number of concurrent calls, the timestamp when it reaches the maximum calls.
- Maximum and average response time.
- Number of calls that time out.
- Number of calls that encounter errors.
- Transfer/zero-out rate. The number of calls transfering to an operator or live agent.
- Abandon rate. The number of calls that end as hang up.
- Call completion rate. The number of calls that are completed as expected through the callflow.
- Maximum logger event queue size.
- Maximum loggers thread count.

The platform level consists of information for Call Services as a product:

- Product Name
- Product Version
- Installation Key
- Licensed Ports

- License Expiration Date
- Licensed GW Adapters

Call Services currently comes with built-in management beans that provide with the access to the runtime information at these three levels. The built-in beans implement a Java API set that is defined for management interfaces. Developers can create custom management beans that use the API. Figure Figure 16-1depicts the relationship among the management API: com.audium.server.management and the built-in and custom beans.

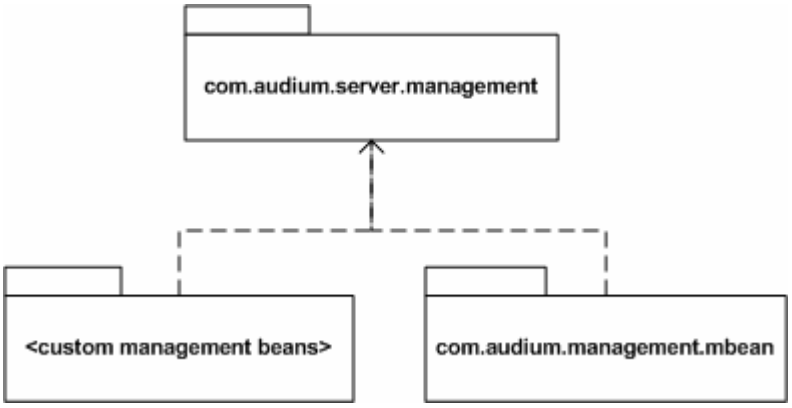


*Figure 16-1*

When Call Services starts, all the beans that are deployed to AudiumHome/Management would be loaded and registered to the JMX server. Depends on the base class it extends, each bean is grouped in VoiceApplication, Global or Info under the domain “Cisco Unified Call Services Application Management API”. For an application-scoped bean that either extends AbstractApplicationCommand, AbstractApplicationConfig or AbstractApplicationData, a bean instance will be created for each application. For instance, if bean A extends AbstractApplicationCommand and there are currently two applications deployed: appA and appB, then two instances of bean A class will be created, one for appA, the other for appB.

For a logger that has been deployed with an application, a management bean will be dynamically generated and registered for it. Currently there is no extra requirement for a logger to become manageable. However, the following operations are excluded from logger beans due to their irrelevance to the manageability: log(), init(), equals(), destroy(), initialize(), hasCode(), getClass(), wait(), notify(), notifyAll(), toString().

Custom beans that directly implement AudiumManagementBeanInterface but does not extend any of the following abstract classes will be loaded but not registered when Call Services starts up:

- AbstractApplicationCommand,
- AbstractApplicationConfig,

- AbstractApplicationData,
- AbstractGlobalCommand,
- AbstractGlobalConfig,
- AbstractGlobalData
- AbstractCallServicesInfo.

This type of beans should register itself to the management server and they do not have access to the information provided by Call Services. Call Services only loads these beans.

For errors or exceptions that happen in a custom bean, it is recommended to handle by use of java.util.logging.Logger to a logging subsystem or com.audium.logger.global.error.GlobalErrorLogger which logs to AudiumHome/logs/GlobalErrorLogger directory.

## Management Bean Samples

As discussed in the design section, Call Services management system provides a public API set that can be used for management bean development. Here is how a typical bean can be created based on the API.

1) Creating the bean interface where it defines the attributes and operations it will manage. For example,

```
public interface ApplicationConfigMBean {

      public String getDefaultAudioPath();

      public void setDefaultAudioPath(String path);

      public String getSuspendedAudioFile();

      public void setSuspendedAudioFile(String file);

}
```

2) Creating the bean class that implements the interface and extends the predefined abstract class.

```
public class ApplicationConfig extends AbstractApplicationConfig
implements ApplicationConfigMBean{

      public String getDefaultAudioPath() {

            return super.getDefaultAudioPath();

      }
```

```
        public void setDefaultAudioPath(String path) {
              super.setDefaultAudioPath(path);
        }
        public String getSuspendedAudioFile() {
              return super.getSuspendedAudioFile();
        }
        public void setSuspendedAudioFile(String file) {
              super.setSuspendedAudioFile(file);
        }
    }
```

3) By extending the predefined abstract class, a custom bean gets access to the information provided by Call Services. For example, a bean class that extends AbstractGlobalData can call getAllDataNames(), removeAllData(), getData(), etc.. These methods are made accessible when the bean gets loaded.

# Application Management Interfaces

When writing custom management beans, the public APIs can be used to have access to the runtime information that will be provided by Call Services. We now discuss each of these interfaces and abstract classes in details. Figure Figure 16-2 demonstrates the static class structure which corresponds to the three levels of the aforementioned information structure. The built-in management beans are also included in the diagram to show the relationship between the APIs and the beans.

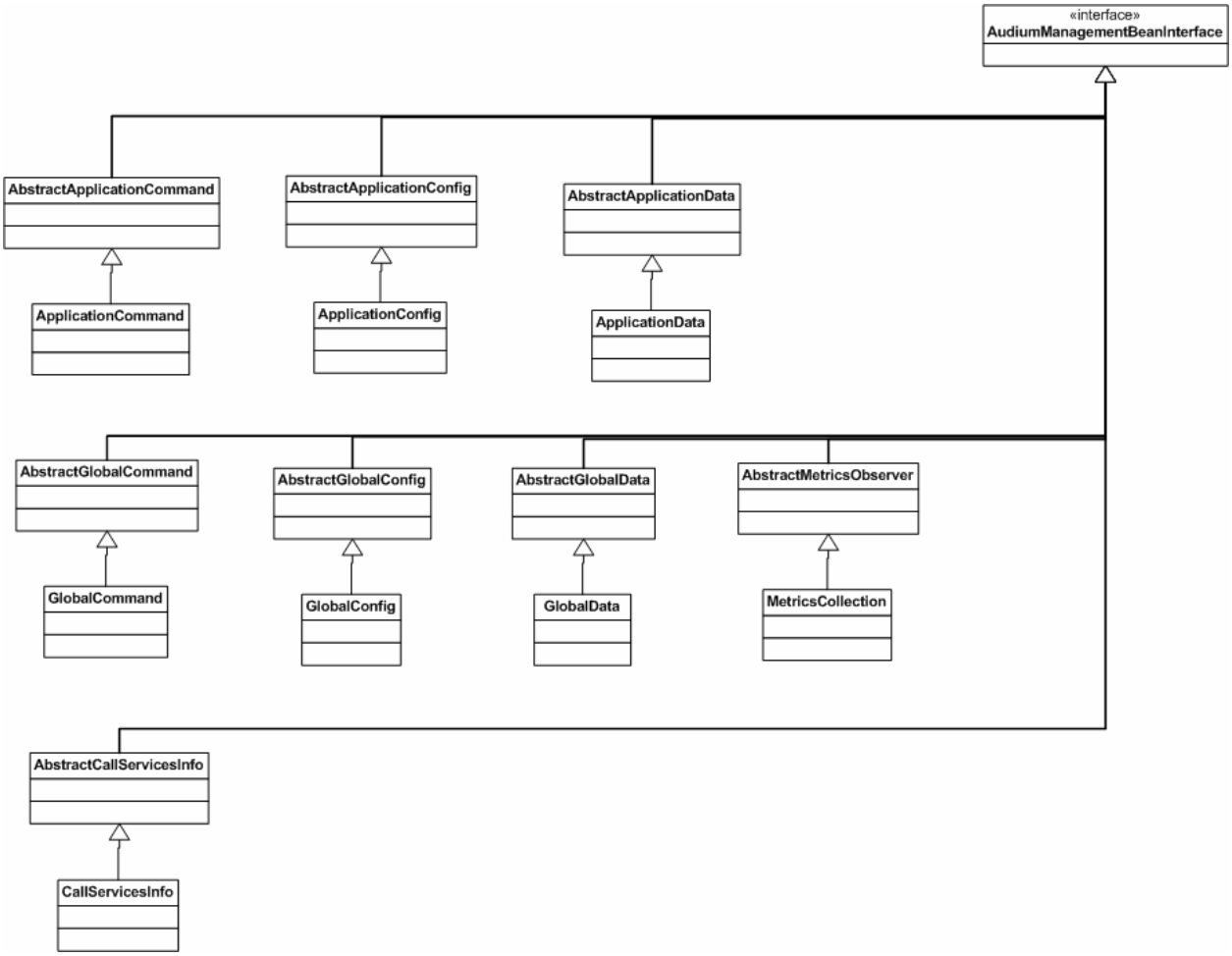


*Figure 16-2*

`AudiumManagementBeanInterface`

This is a mark interface for any JMX MBeans to load at the server startup. All the management beans including ones created by customers must implement this interface or its subinterfaces.

`AbstractApplicationCommand`

This abstract class encapsulates all the application commands. The JMX beans that extend this class will have access to all these operations.

`AbstractApplicationConfig`

This class can be used to set and get application configuration attributes.

`AbstractApplicationData`

MBeans that extend this class will have access to application data and can manipulate these data by adding or removing them. Application data is typically used to store application-specific information that does not change on a per call basis and therefore is available to all calls.

`AbstractGlobalCommand`

This abstract class encapsulates all the global commands. The JMX beans that extend this class will have access to all these operations.

`AbstractGlobalConfig`

This class can be used to set and get global configuration attributes.

`AbstractGlobalData`

Subclasses of this class will have access to global data and can manipulate these data by adding or removing them. Global data is typically used to store static information that needs to be available to all components, no matter which application they reside in.

`AbstractMetricsCollection`

This class can be used to access all the metrics information at the global level.

`AbstractMetricsObserver`

This is a marker class. When Call Services starts up, subclasses of this class will be added as listeners to application activity events, logging management events and global events. Although a subclass will be notified when any of these events happens, it only needs to react to the interested events. Events are supposed to be handled by overwriting the update() method.

# Appendix A: The Voice Foundation Classes

The Universal Edition Voice Foundation Classes are a Java API for generating VoiceXML. Any custom component wishing to produce VoiceXML must use the VFCs because their main purpose is to act as an abstraction layer between VoiceXML and the component. The VFCs handle the vagaries of VoiceXML and especially the differences in the VoiceXML interpreted by various voice browsers. This allows the developer to simply focus on the functionality desired without worrying about the details of writing VoiceXML or the quirks of their chosen voice browser. The VFCs are primarily used to construct voice elements, though hotevents and on call end classes use the VFCs as well.

## VFC Design

The high level design of the VFCs is to simulate standard VoiceXML in Java. The behavior of these classes directly match the VoiceXML specifications (both versions 1 and 2). This, however, acts only as a basis from which supporting a particular voice browser begins, since no two browsers have exactly the same compliance. The software provides voice browser compatibility by extending these base VFCs to create a layer that produces the VoiceXML compatible with a particular voice browser. Most of the functionality is still defined in the base VFC classes and only the browser-specific functionality needs to be included in the subclasses. The classes for a particular voice browser are encapsulated in a separate plugin or driver, called a Gateway Adapter. Installing a new Gateway Adapter will add support for a new voice browser and a Universal Edition application can be deployed on a new browser by simply selecting the Gateway Adapter to use.

The design of the base VFCs follows roughly the design of VoiceXML, utilizing similar concepts and naming, so prior knowledge of VoiceXML is beneficial for understanding the VFC design. The VFCs allow full compatibility with VoiceXML in that anything you can do in VoiceXML you can do in the VFCs, including using proprietary tags and/or attributes introduced by supported browsers. Many times, a single VoiceXML tag maps to a single VFC that is similarly named. The class `VForm`, for example, deals with VoiceXML `<form>` tags and the class `VField` with `<field>` tags. Some tags, however, have been combined into a single VFC for ease of use. For example, the `VAction` class encapsulates tags such as `<var>` and `<assign>` to `<break>` and `<submit>`. As a result, there are fewer VFCs than VoiceXML tags. The VFCs also help the developer by producing some VoiceXML automatically. The developer will quickly find that using the VFCs is very much like coding in VoiceXML, except in Java.

There are a few concepts that need to be described before delving into the individual VFCs. First, each VFC class extends a common base class, `VRoot`. The purpose for this is similar to having all Java classes extend `Object`, it is a way to help define common functionality of the VFCs as well as being able to identify if a Java class is a VFC.

The second concept involves the hierarchy of the VFCs. There are, in fact, several layers of abstraction in the VFCs that separate not only differences between various voice browsers but

also versions of the VoiceXML standard. There are separate VFCs for VoiceXML version 1.0 and version 2.0, and the similarities are encapsulated in a common base class. Figure A-1 shows this graphically with the `VForm` class. The main `VForm` class extends the `VRoot` class and is itself extended by `VFormV1` and `VFormV2`, representing VoiceXML 1.0 and 2.0 compliance. Luckily, there are only a few differences between these versions, so the developer can still do most coding to the base `VForm` class. The Gateway Adapters introduce VFCs that extend `VFormV1` or `VFormV2` depending on whether the voice browser it supports is compatible with VoiceXML 1.0 or 2.0.

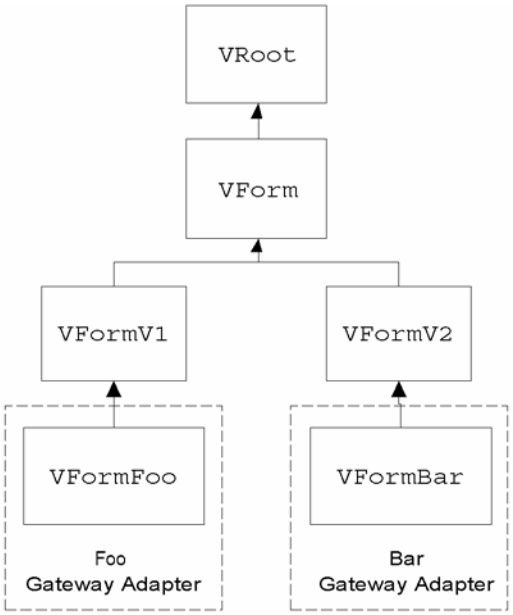


*Figure A-1*

The last concept is that VFC objects are not instantiated using the `new` keyword. A static factory method named `getNew` is used instead. The reason for this is related to the abstraction of the voice browser differences. As mentioned previously, a developer need only code using the base VFC classes. At runtime, the factory methods used to instantiate VFC classes actually returns the appropriate voice browser-specific VFC (for example, `VFormFoo` in the example in Figure A-1). But since the developer treats the return object as the base VFC, that object is downcasted. This is the heart of the VFC abstraction design. Since all VFC derivative classes extend their base VFCs, a developer need only code to the base VFCs and that automatically makes their code compatible with any voice browser represented by a Gateway Adapter.

In order to identify which voice browser VFC to return, every factory method must include as its first argument an instance of `VPreference`. `VPreference`, while a VFC class, does not match to a VoiceXML tag, it is used instead to hold preferences made by the user in Universal Edition Builder for Call Studio for the application, such as the voice browser and default audio path. By passing this object to all factory methods, the appropriate object can be returned. The `VPreference` instance is automatically created for the developer and made available through the Session API passed to voice elements, hotevents, or call end classes.

The following Java code demonstrates the concepts described above:

```
VPreference pref = ved.getPreference();
VForm form = VForm.getNew(pref, "start");
```

Here, the `VPreference` object is obtained from the `VoiceElementData` object passed as input to a voice element. It is used to create a `VForm` object. Assuming the application is using the voice browser Foo, that choice is reflected in the `VPreference` object and therefore the `getNew` factory method returns a `VFormFoo` object, which is automatically downcasted to a `VForm` object. The developer then uses the form object as desired.

This ability to treat all objects returned as a root VFC object is not available when the developer wishes to use functionality that exists either in a particular version of VoiceXML or a particular voice browser. The developer must understand that doing so would prevent their code from functioning on all voice browsers. In this case, the developer simply treats the return of the factory method as a class higher in the class hierarchy (`VFormV2` or `VFormFoo` in the example in Figure A-1).

The following Java code demonstrates this:

```
VGrammar myGrammar = VGrammar.getNew(pref);
((VGrammarV2) myGrammar).setMaxage(1000);
```

The `setMaxage` method exists only in the `VGrammarV2` class since this is a feature that exists only in VoiceXML 2.0. To call this method, one must first upcast the previously downcasted object back to `VGrammarV2`. If this is not done, an exception will be thrown indicating that `VGrammar` does not have a method named `setMaxage`. Also note that if the user in the Builder chose a voice browser that was compatible with VoiceXML 1.0 only, a runtime exception would be thrown when this code is encountered because that browser would be unable to understand VoiceXML referring to maxage.

# VFC Classes

The following lists all the VFC classes (with full package names) and briefly explains what they are responsible for. The Javadocs for the VFCs provide significantly more detail about the classes, their methods, and how they are used.

- **com.audium.core.vfc.util.VMain.** This object is the container for a complete VoiceXML document. It includes methods for managing information about the page such as the meta tags, the doc type, and the value to put in the `<vxml>` tag's `xml:lang` attribute. It includes methods for adding document-scope data such as links, variables, and VoiceXML properties. `VForm` objects are added to this object to create the VoiceXML page. Call Services uses the `VMain` object to handle the printing of the VoiceXML page. Voice elements receive an instantiated `VMain` object as input and the developer need only worry about filling the object with the appropriate content.

- **com.audium.core.vfc.form.VForm.** This class is a container for all the content in a VoiceXML page not handled by the `VMain` class. It is a direct mapping of the `<form>` tag, though it also produces other form-level tags such as `<var>` or `<filled>`.
- **com.audium.core.vfc.list.VList.** This class is used to encapsulate a list of items that can be deployed as either a traverse list or a streaming list. A traverse list presents a menu after an item is presented that allows the caller to move forwards and backwards through the list. A streaming list is one where all the items are played one after the other. This VFC class does not reflect any VoiceXML tags, it was produced by Universal Edition to facilitate the creation of lists within VoiceXML. The class outputs a set of forms that implement the list.
- **Form Items.** `VForm` classes encapsulate most of the content of a VoiceXML page, and each form has any number of form items added to it. These form items span the range of capturing input from the caller to performing a telephony transfer. Each form item has a different purpose, though many form items share features in common. The VFCs relate the classes that handle each form item by creating a hierarchy starting with the simplest form items, with features common to all, to more complex form items that add features through each successive class extension. Figure A-2 shows this class hierarchy and a description of each branch is listed below.

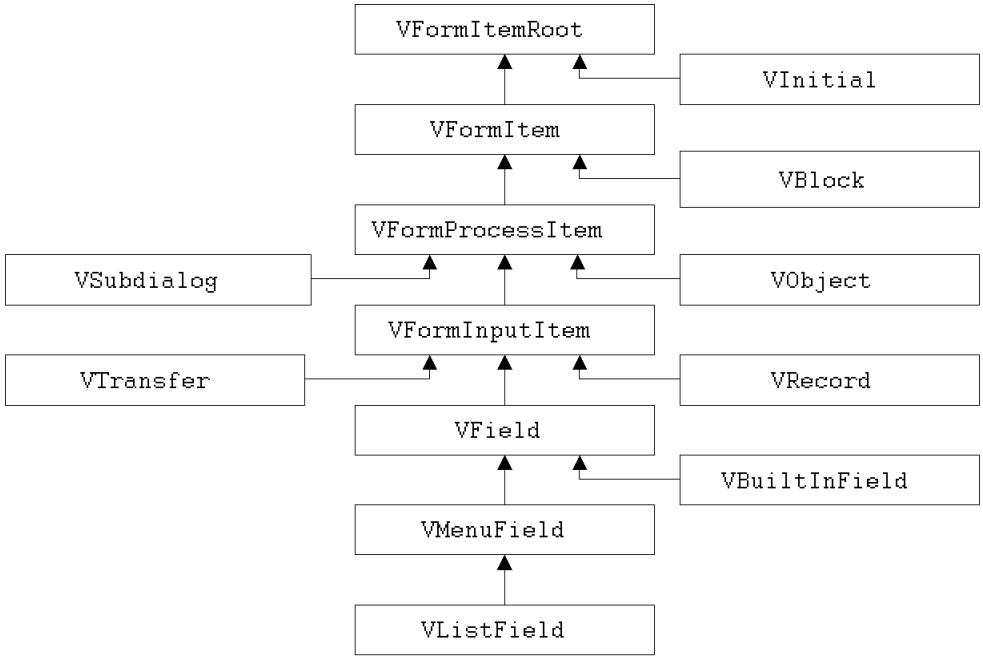


*Figure A-2*

- **com.audium.core.vfc.form.VFormItemRoot.** This class is the base class for all form items. It defines the ability to include audio, which every form item shares.
  - **com.audium.core.vfc.form.VInitial.** This class is used when performing mixed initiative data capture. Mixed initiative data capture is a way of capturing multiple inputs in one utterance, such as a person's first and last names together rather than having to prompt for each individually. It is a direct mapping of the `<initial>` tag.
- **com.audium.core.vfc.form.VFormItem.** This class defines a standard form item. It defines the ability to perform actions within the form item (some form items do this within a `<filled>` tag).
  - **com.audium.core.vfc.form.VBlock.** This class deals with producing a block in which any action and/or audio can be placed. It is a direct mapping of the `<block>` tag.
- **com.audium.core.vfc.form.VFormProcessItem.** This class defines form items that process data from the caller or an external source. Process form items define the ability to catch and handle VoiceXML events and refer to VoiceXML properties.
  - **com.audium.core.vfc.form.VSubdialog.** This class is used to make a call to a VoiceXML subdialog. A subdialog acts very much like a function call in VoiceXML, performing some encapsulated task and then returning to the calling context. It is a direct mapping of the `<subdialog>` tag.
  - **com.audium.core.vfc.form.VObject.** This class is used to produce VoiceXML that calls an external data object. It is a direct mapping of the `<object>` tag.
- **com.audium.core.vfc.form.VFormInputItem.** This class defines form items that take input from the caller. Input form items define a grammar to capture the data.
  - **com.audium.core.vfc.call.VTransfer.** This class deals with performing a telephony transfer. It is a direct mapping of the `<transfer>` tag. Note that the reason this is considered an input form item is because theoretically according to the VoiceXML specification, a grammar can be active within a call transfer. This, though, is rarely used or supported.
  - **com.audium.core.vfc.audio.VRecord.** This class deals with performing a recording of the caller's voice. It is a direct mapping of the `<record>` tag.
- **com.audium.core.vfc.form.VField.** This class defines field form items, which deal with capturing utterances from the caller and converting them into information. Fields define the ability to specify utterance links.
  - **com.audium.core.vfc.form.VBuiltInField.** This class deals with producing fields that capture data specified by grammars built into the voice browser. Any voice browser supporting VoiceXML is required to support data capture of numbers, dates, times, currency values, phone numbers, digit-by-digit values, and boolean values (yes / no). The class produces `<field>` tags as well as other field-related tags such as `<prompt>` and `<filled>`.

  - **com.audium.core.vfc.form.VMenuField.** This class is used when a menu is desired in the VoiceXML document. The class produces `<field>` tags with `<option>` tags defining each menu option. The `<menu>` and `<choice>` tags in VoiceXML are just shortcuts for this and cannot be produced with the VFCs.
  - **com.audium.core.vfc.form.VListField.** This class is a special kind of menu that is used by the `VList` class when it is configured to act as a traverse list. The menu is pre-built to support options to go forwards and backwards.

- **com.audium.core.vfc.util.VAction**. This VFC class encapsulates multiple VoiceXML tags that represent taking certain actions. All the VoiceXML tags produced by this class have the same parent tags and so can be used in the same locations. Combining these tags into one class reduces the complexity of the VFCs since special handlers are not needed for each tag. The following lists the actions that the `VAction` tag encapsulates and the corresponding VoiceXML tag: variable declarations (`<var>`), variable assignment (`<assign>`), gotos (`<goto>`), HTTP submits (`<submit>`), clearing forms and fields (`<clear>`), scripts (`<script>`), logging (`<log>`), throwing events (`<throw>`), reprompting (`<reprompt>`), returning from subdialogs (`<return>`), disconnects (`<disconnect>`) and exits (`<exit>`).
- **com.audium.core.vfc.audio.VAudio.** This class deals with audio, both TTS and through audio files. A single `VAudio` object can contain any number of audio items (so an entire voice element audio group can be encapsulated in one `VAudio` object). The class also deals with playing back a recording, managing bargein, adding pauses to the playback. Note that SSML (Speech Synthesis Markup Language) that is entered by the application designer in Builder for Call Studio is handled correctly by this class.
- **com.audium.core.vfc.util.VEvent.** This class handles VoiceXML events and what to do when they occur. Events may be user-triggered such as nomatch or noinput events, or custom events thrown by the developer or voice browser. Hotevents are basically `VEvent` classes that Call Services adds to the VoiceXML root document. It is a direct mapping of the `<catch>` tag. Note that the `<noinput>`, `<nomatch>`, and `<help>` tags are all shortcuts for variations of the `<catch>` tag so are not produced by the VFCs.
- **com.audium.core.vfc.util.VGrammar.** This class deals with specifying either an inline or external DTMF or speech grammar. It is a direct mapping of the `<grammar>` tag.
- **com.audium.core.vfc.util.VIfGroup.** This class deals with producing an if statement within VoiceXML. It is a direct mapping of the `<if>`, `<elseif>`, and `<else>` tags.
- **com.audium.core.vfc.util.VLink.** This class deals with creating an utterance-activated link within the VoiceXML page. It is a direct mapping of the `<link>` tag.
- **com.audium.core.vfc.util.VProperty.** This class deals with including VoiceXML properties in the VoiceXML page. It is a direct mapping of the `<property>` tag.
- **com.audium.core.vfc.VException.** This exception class is thrown when a VFC class encounters an error.

- **Utility Classes**. These classes are used by the VFCs to aid in the organization of data they require. The following lists those classes:
  - **com.audium.core.vfc.util.VoiceInput.** This class is used to encapsulate how input is to be expected from the caller. It can encapsulate voice only input, DTMF only input, or both. It is also used to specify what data to look for, which can be a single or multiple keywords or keypresses. This class is typically used with menus and forms.
  - **com.audium.core.vfc.util.IfCondition.** This class is used to specify an expression to put inside an if statement. It handles standard numerical and string operations and can support expressions contains "ands" (&&) and "ors" (||).
  - **com.audium.core.vfc.form.UsedInFilled.** This class is a Java interface that is used to identify all the VFCs that can be used inside a `<filled>` tag. It is used simply as a marker for those VFCs.

# Appendix B: The Java 5 Migration

From Call Services 6.0 on, the entire Call Services code base has been compiled with Java 5. All the existing application interfaces remain unchanged in the migration. Custom classes such as custom elements that have been compiled with Java 2/JDK 1.4.x will continue to work on the new Call Services platform.