User Guide for Cisco Unified CVP VXML Server and Cisco Unified Call Studio

Release 10.0(1)

December 12, 2013

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

User Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio

Contents

Chapter 1: Introduction

Welcome to Cisco Unified Call Services, Universal Edition (Call Services) and Cisco Unified Call Studio (Call Studio), the most robust platform for building exciting, dynamic VoiceXML-based voice applications. This platform:

- Allows users to build complex voice applications without requiring extensive knowledge of Java and VoiceXML.
- Includes an easy, graphical interface for building voice applications and simplifies the tasks of building custom components that easily plug into the software's modular architecture.
- Provides the fastest, most error-free process for building professional, dynamic voice applications.

This user guide introduces the process of building voice applications utilizing the various components of Universal Edition software. Its primary focus is to explain the concepts required to get the most out of one Universal Edition component, Call Services, while introducing the others. It will refer to additional documentation to fully describe other components. The reader just getting started with Universal Edition software should read at least the first few chapters to get an idea of the environment in which Universal Edition software revolves and some of the design of the Universal Edition platform.

VoiceXML Overview

Since its introduction in 2000, VoiceXML has quickly become the standard technology for deploying automated phone systems. To understand VoiceXML's quick acceptance by enterprises, carriers and technology vendors, a brief overview of the traditional technologies used to develop interactive voice response systems is given.

Limitations of Traditional IVR Technologies

Despite investing millions of dollars in Interactive Voice Response (IVR) systems, many organizations know that the applications responsible for handling automated customer service do not fulfill their business requirements. Organizations need their IVR to be as flexible and dynamic as the rest of their enterprise applications, but proprietary, one-size-fits-all solutions cannot easily support regular modifications or new corporate initiatives. Additionally, most of these IVR solutions are not speech enabled and upgrading to speech recognition on a traditional IVR platform is difficult and costly.

Heightened customer expectations for fast, quality service and a consistent experience across phone and web contact channels are putting pressure on businesses to implement a higher quality IVR solution. However, due to their proprietary nature, traditional IVR systems do not allow the choice and flexibility necessary to meet the increasing demands of high expectation customers. While the limitations of a traditional IVR pose considerable challenges for many organizations, some smart businesses have found a solution by implementing the flexible and powerful new standard in IVR technology: VoiceXML.

VoiceXML: Simplifying IVR Development

VoiceXML is a programming language that was created to simplify the development of IVR systems and other voice applications. Based on the Worldwide Web Consortium's (W3C's) Extensible Markup Language (XML), VoiceXML was established as a standard in 1999 by the VoiceXML Forum, an industry organization founded by AT&T, IBM, Lucent and Motorola. Today, many hundreds of companies support VoiceXML and use it to develop applications.

By utilizing the same networking infrastructure, HTTP communications, and markup language programming model, VoiceXML leverages an enterprise's existing investment in technology as well as the skills of many of its application developers and administrators. VoiceXML has features to control audio output, audio input, presentation logic, call flow, telephony connections, and event handling for errors. It serves as a standard for the development of powerful speech-driven interactive applications accessible from any phone.

Key Business Benefits of VoiceXML

A VoiceXML-based IVR provides unparalleled freedom of choice when creating, deploying, and maintaining automated customer service applications. By capitalizing on the standards-based nature of VoiceXML, organizations are reaping a number of benefits including:

- **Unparalleled portability** – VoiceXML eliminates the need to purchase a proprietary, special purpose platform to provide automated customer service. The standards-based nature of VoiceXML allows IVR applications to run on any VoiceXML platform, eliminating vendor lock-in. A VoiceXML based IVR offers businesses choice in application providers and allows movement of applications between platforms with minimal effort.
- **Flexible application development and deployment** – VoiceXML enables freedom of choice in IVR application creation and modification. Since it is similar to HTML, development of IVR applications with VoiceXML is simple, straightforward and does not require specialized knowledge of proprietary telephony systems. Also, VoiceXML is widely available to the development community so enterprises can choose between many competing vendors to find an application that meets their business needs. Increased application choice also means that businesses are not tied to the timeframe of a single application provider and can modify their IVR based on their own organizational priorities.
- **Extensive integration capability** – IVR applications written in VoiceXML can integrate with and utilize existing business applications and data, extending the capabilities of core business systems already in use. In fact, a VoiceXML-based IVR can integrate with any enterprise application that supports standard communication and data access protocols. By leveraging the capabilities of existing legacy and web systems to deliver better voice services, organizations can treat their IVR like their enterprise applications and fulfill business demands with an integrated customer facing solution.

  By taking advantage of the increased number of choices offered by a VoiceXML-based IVR, businesses can easily deliver the flexible, dynamic customer service that their organizations and customers demand. The wide array of options available allows businesses to maximize existing resources to deliver better service at lower cost.
- **Reduced total cost of ownership** – The freedom of choice offered by a VoiceXML-based IVR reduces the total cost of ownership in several key areas:
  - **Speech capability is standard** – The architecture of VoiceXML directly supports integration with speech recognition, making implementing a VoiceXML-based IVR a cost effective alternative to retrofitting a traditional IVR for speech. Extensive industry research indicates that incorporating speech into an IVR solution increases call completion, lowering the average cost per call.
  - **Lower hardware and maintenance costs** – VoiceXML applications run on commonly available hardware and software, enabling businesses to save money by using equipment that they already own instead of purchasing special purpose hardware. Additionally, businesses can use the same team that handles existing enterprise maintenance to maintain IVR applications written in VoiceXML.
  - **Affordable scaling** – In a VoiceXML-based IVR model, application logic resides on a web/application server and is separate from telephony equipment. Businesses can avoid unneeded capital investment by purchasing capacity for regular day-to-day needs and outsourcing seasonal demand to a network provider.

- **Applications for every budget** – Competition between VoiceXML application developers provides a variety of IVR solutions for budgets of all sizes. Businesses only pay for needed application features as an open marketplace offers a larger number of competing applications at varying price points.

How VoiceXML Works

Designed to leverage Web infrastructure, VoiceXML is analogous to HTML, which is a standard for creating Web sites. Like HTML, the development of voice applications using VoiceXML is simple, straightforward and therefore does not require specialized knowledge of proprietary telephony systems. Since the intricacies of developing voice applications are hidden from developers, they can focus on business logic and call flow design rather than complex platform and infrastructure details.

With VoiceXML, callers interact with the voice application over the phone using a voice browser. The voice browser is analogous to a graphical Web browser, such as Microsoft's Internet Explorer. Instead of interpreting HTML as a web browser does, the voice browser interprets VoiceXML and allows callers to access information and services using their voice and a telephone.

Figure 1: VoiceXML Platform Architecture

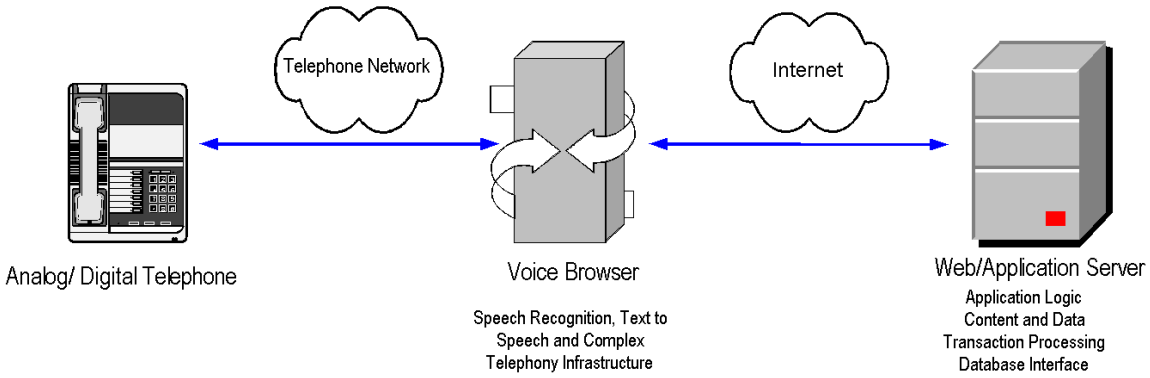


Figure 1-1

As indicated in Figure 1-1, the primary components of the VoiceXML platform architecture are the telephone, voice browser and application server. The voice browser, a platform that interprets VoiceXML, manages the dialog between the application and the caller by sending requests to the application server. Based on data, content and business logic, the application server creates a VoiceXML document dynamically or uses a static VoiceXML document that it sends back to the voice browser as a response.

Challenges with VoiceXML Development

Despite the robustness and broad acceptance of VoiceXML as the new standard for voice applications, there are a number of challenges that developers face when deploying complicated systems, including:

- **Requirement for dynamic VoiceXML** – Many applications require the ability to dynamically insert content or to base business logic on data available only at runtime. In these cases, the VoiceXML must be dynamically generated. For example, an application that plays a "Good Morning / Afternoon / Evening" prompt depending on the time of day requires VoiceXML to be dynamically generated.
- **Voice Paradigm versus Web paradigm** – There are many systems designed to manage dynamic web content or to automatically convert web content to other formats (such as for wireless phones). These systems, however, are not adequate for voice applications due to the fundamental difference between a voice application and a Web application. A web page is a two-dimensional, visual interface while a phone call is a one-dimensional, linear process. Converting web content to voice content often yields voice applications with lackluster user interfaces.
- **Browser compatibility** – Due to ambiguities and constant improvements in the VoiceXML specification, no two commercially available browsers accomplish various functions in exactly the same way. Developers must understand the variations between browsers when coding VoiceXML to ensure compatibility.
- **Stateless nature of VoiceXML** – Like HTML, VoiceXML is a stateless mark-up language. For applications that require the maintenance of data across a session, e.g., account or transactional information, or phone call, pure VoiceXML does not suffice.
- **Complicated coding** – Despite VoiceXML's promise to simplify voice application development, the process of coding an application with dozens or hundreds of possible interactions with a caller can become quite complex.
- **Limited back-end integration** – Enterprise applications rarely operate in a vacuum. VoiceXML does not natively support robust data access and external system integration.
- **OAM&P requirements** – Operators of large-scale voice applications have significant requirements for administration, management, logging and (sometimes) provisioning. VoiceXML does not natively support most of these functions.
- **Reusability** – The larger a Web or voice application becomes, the more critical reusability becomes. This is even more pronounced in dynamic applications. VoiceXML simply provides the interface for a voice application; it does not encapsulate common application functionality into configurable, reusable building blocks.

The Universal Edition Solution

To address the challenges, Universal Edition provides a complete solution for rapidly conceiving, creating and deploying dynamic VoiceXML 2.0 compliant applications. In order to understand how to use Universal Edition to build dynamic voice applications, one must understand the components of the system and how they work. This section presents a high-level overview of all the components of Universal Edition software.

Universal Edition consists of three main components, Call Studio, Call Services and Elements. Each of these components is discussed in further detail in the remainder of this section.

Call Studio

Call Studio is a development platform for the creation of voice applications. Call Studio provides a framework on which a whole host of Universal Edition and third-party tools will appear with a robust, consistent interface for voice application designers and developers to use. Call Studio will provide a true control panel for developing all aspects of a voice application, each function implemented as a plug-in to the greater Call Studio platform.

The most important plug-in for Call Studio is Builder for Call Studio (or the Builder for short), the component that provides a drag-and-drop graphical user interface (GUI) for the rapid creation of advanced voice applications. Builder for Call Studio provides:

- **Intuitive interface** – Using a process similar to flowcharting software, the application developer can use Builder for Call Studio to create an application, define its call flow, and configure it to the exact specifications required.
- **Design and build at the same time** – Builder for Call Studio acts as a design tool as well as a building tool, allowing the developer to rapidly try different application call flows and then test them out immediately.
- **No technical details required** – Builder for Call Studio requires little to no technical knowledge of Java, VoiceXML, or other markup languages. For the first time, the bulk of a voice application can be designed and built by voice application design specialists, not technical specialists.
- **Rapid application development** – By using Builder for Call Studio, developers can dramatically shorten deployment times. Application development time is reduced by as much as 90% over the generation and management of flat VoiceXML files.

Call Studio documentation resides primarily within Call Studio itself by accessing the Help menu. However, this guide includes a brief introduction to Call Studio in the section entitled Call Studio Introduction later in this chapter.

Call Services

Call Services is a powerful J2EE- and J2SE-compliant run-time engine that dynamically drives the caller experience. Call Services provides:

- **Robust back-end integration** – Call Services runs in a J2SE and J2EE framework, giving the developer access to the full litany of middleware and data adapters currently available for those environments. Additionally, the Java application server provides a robust, extensible environment for system integration and data access and manipulation.
- **Session management** – Call and user data are maintained by Call Services so that information captured from the caller (or environment data such as the caller's number or the dialed number) can be easily accessed during the call for use in business rules.
- **Dynamic applications** – Content and application logic are determined at runtime based on rules ranging from simple to the most complex business rules. Almost anything about an application can be determined at runtime.
- **System Management** – Call Services provides a full suite of administration tools, from managing individual voice applications without affecting users calling into them, to configurable logging of caller activity for analytical purposes.
- **User Management** – Call Services includes a lightweight customer data management system for applications where more robust data are not already available. The user management system allows dynamic applications to personalize the call experience depending on the caller.

The capabilities of Call Services listed above are discussed in further detail in Chapter 3: Administration, Chapter 3: User Management and Chapter 5: Call Services Logging.

Elements

The Elements are a collection of pre-built, fully tested building blocks to speed application development.

- **Browser compatibility** – Universal Edition's library of Voice Elements produce VoiceXML supporting the industry's leading voice browsers. They output dynamically generated VoiceXML 2.0 compliant code that has been thoroughly tested with each browser.
- **Reusable functionality** –Elements encapsulate commonly found parts of a voice application, from capturing and validating a credit card to interfacing with a database. Elements greatly reduce the complexity of voice applications by managing low-level details.
- **Configurable content** –Elements can be significantly configured by the developer to tailor their output specifically to address the needs of the voice application. Pre-built configurations utilizing proven dialog design techniques are provided to further speed the development of professional grade voice applications.

In Universal Edition, there are five different building block types, or *elements*, that are used to construct any voice application: voice elements, VoiceXML insert elements, decision elements, action elements, and flag elements. Call Services combines these elements with three additional concepts: hotlinks, hotevents, and application transfers, to represent a voice application.

The building blocks that make up an application are referred to as elements. In Universal Edition, elements are defined as:

Element	*A distinct component of a voice application call flow whose actions affect the experience of the caller.*

Many elements in Universal Edition share several characteristics such as the maintenance of element data and session data, the concept of an exit state, and customizability.

Element and Session Data

Much like variables in programming, elements in a voice application share data with each other. Some elements capture data and require storage for this data. Other elements act upon the data or modify it. These variables are the mechanism for elements to communicate with each other. The data comes in two forms: element data and session data.

- **Element data** are variables that exist only within the element itself, can be accessed by other elements, but can only be changed by the element that created them.
- **Session data** are variables that can be created and changed by any element as well as some other non-element components.

Exit States

Each element in an application's call flow can be considered a “black box” that accepts an input and performs an action. There may be multiple results to the actions taken by the element. In order to retain the modularity of the system, the consequences of these results are external to the element. Like a flowchart, each action result is linked to another element by the application designer. The results are called exit states. Each element must have at least one exit state and frequently has many. The use of multiple exit states creates a “branched” call flow.

Customizability

Most elements require some manner of customization to perform specific tasks in a complex voice application. Customization is accomplished through three different mechanisms supported by Universal Edition: a fixed configuration for the element, a Java API to dynamically configure pre-built elements or to define new ones, and an API accessed via XML-data delivered over http to do the same.

- The **fixed configuration** approach provides a static file containing the element configuration so that each time the element is visited in the call flow it acts the same. Even in dynamic voice applications, not every component need be dynamic; many parts actually do not need to change.
- The **Java API** approach is used for dynamic customization and is a high performance solution because all actions are run by compiled Java code. The one drawback to this approach is that it requires developers to have at least some Java knowledge, though the Java required for interfacing with the API is basic.
- The **XML-over-HTTP** (or XML API for short) approach affords developers the ability to utilize any programming language for the customization of elements. The only requirement is the use of a system that can return XML based on an HTTP request made by Call Services. The advantages of this approach include: a larger array of programming language choices, the ability to physically isolate business logic and data from the voice presentation layer and the use of XML, which is commonly used and easy to learn. The main disadvantage of this approach is the potential for HTTP connection problems, such as slow or lost connections. Additionally, the performance of this approach does not typically perform as well as compiled Java because XML must be parsed at runtime in both Call Services and the external system.

Voice Elements

Almost all voice applications must utilize a number of dialogs with the caller, playing audio files, interpreting speech utterances, capturing data entered by the user, etc. The more these dialogs can be contained in discrete components, the more they can be reused in a single application or across multiple applications. These dialog components are encapsulated in v*oice elements*.

Voice Element	*A reusable, VoiceXML-producing dialog with a fixed or dynamically produced configuration.*

Voice elements are used to assemble the VoiceXML sent to the voice browser. Each voice element constitutes a discrete section of a call, such as making a recording, capturing a number, transferring a call, etc. These pre-built components can then be reused throughout the call flow wherever needed.

Voice elements are built using the Universal Edition Voice Foundation Classes (VFCs), which produce VoiceXML compatible with multiple voice browsers (see the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more on the VFCs and constructing custom voice elements).

Voice elements are complete dialogs in that they can encompass just a single action or an entire interaction with the caller. Depending on its function, a voice element can contain almost as much dialog as a small application. However, because of the pre-built nature of voice elements, application designers do not need to worry about their complexity. Each voice element is simply a “black box” which can be treated as a single object. As a result, by combining many voice elements, a complex call flow can be reduced significantly.

Each voice element defines the exit states it can return and the designer must map each exit state to another call flow component to handle all its consequences. To fully configure voice elements, developers must specify values for four components: settings, VoiceXML properties, audio groups, and variables.

- **Settings** are used to store information that affects how the voice element performs. For example, a setting describes what phone number to transfer to or the length of audio input recording. A voice element can have many or few settings, depending on its complexity and its level of customization.
- **VoiceXML properties** are equivalent to the properties outlined in the VoiceXML specification, and are used to modify voice element behavior by directly inserting data into the VoiceXML that each element produces. For example, the length of time the voice element waits before encountering a noinput event can be changed by setting a VoiceXML property. Available properties correspond directly to those listed in the VoiceXML specification and voice

browser specification. It is up to the designer to understand the consequences of modifying these properties.

- **Audio Groups** – Nearly all voice elements involve the use of audio assets, whether in the form of pre-recorded audio files or text-to-speech (TTS) phrases. An audio group encapsulates the audio that the application plays when reaching a certain point in the voice element call flow. For example, an audio group might perform the function of asking a question, giving an answer, playing an error message, etc. An audio group may contain any number of audio items. Audio items are defined as pre-recorded audio files, TTS phrases, or information that conforms to a specified format to be read to the user (such as a date or currency value). Each audio item in an audio group is played in the order they appear in the audio group.
- **Variables**, as described in the previous section, allow voice elements to set or use element or session data. Many voice elements use element data to store information captured from a caller, though voice element configurations can also define additional variables.

Finally, a voice element's configuration can be either fixed or dynamic.

- **Fixed configurations** are XML files containing the desired settings, VoiceXML properties, audio groups, and variables that are then loaded by Call Services. The same configuration is applied each time the voice element is called.
- The configuration of some voice elements can only be determined at runtime. In these cases a **dynamic configuration** is used. As described previously, the Java API and XML API can be used to create dynamic configurations.

For a complete list of the voice elements included in Universal Edition, refer to the Element Specifications for Cisco Unified Call Services, Universal Edition and Unified Call Studio document.

VoiceXML Insert Elements

There are certain situations in a voice application where a developer may wish to include pre-written VoiceXML into their Universal Edition application. The developer may desire fine-level control over a specific voice function at the VoiceXML tag level without having to get involved with constructing a custom configurable element in Java. Additionally, the developer may wish to integrate VoiceXML content that has already be created and tested into a Universal Edition application. These situations are handled by a *VoiceXML insert element*.

VoiceXML Insert Element	*A custom element built in VoiceXML providing direct control of lower-level voice dialog at the price of decreased flexibility.*

VoiceXML insert elements contain VoiceXML code that the developer makes available as the content of a VoiceXML `<subdialog>`. The content can be in the form of static VoiceXML files, JSP templates, or even dynamically generated by a separate application server. A framework is

provided to allow seamless integration of VoiceXML insert elements with the rest of the call flow.

The use of VoiceXML insert elements has its consequences such as the loss of being able to seamlessly switch between different voice browsers, some greater processing overhead involved with integration with the rest of the call flow, as well as the added complexity of dealing with VoiceXML itself rather than creating an application with easy to use configurable elements.

VoiceXML insert elements can have as many exit states as the developer requires, with a minimum of one.

Decision Elements

Even the simplest voice applications require some level of decision making throughout the call flow. These "crossroads" are encapsulated in *decision elements*.

Decision Element	*Encapsulates business logic that make decisions with at least two exit states.*

A decision element is like a traffic cop, redirecting the flow of callers according to built in business rules. Examples of business rules include decisions such as whether to play an ad to a caller, which of five different payment plans should be offered to the caller, or whether to transfer a caller to an agent or hang up.

The results of a decision element are represented as exit states. Although many decisions are boolean in nature, (e.g. "has the caller registered?", "is the caller new to the application?"), decision elements can have as many exit states as desired, as long as at least two are specified.

The configuration for a configurable decision contains two components: settings and variables. Additionally, the Java class that defines the configurable decision sets the exit states it can return and the designer must map each exit state to another call flow component to handle all its consequences.

Action Elements

Many voice applications require actions to occur "behind the scenes" at some point in the call. In these cases, the action does not produce VoiceXML (and thus has no audible effect on the call) or perform some action that branches the call flow (like a decision). Instead the action makes a calculation, interfaces with a backend system such as a database or legacy system, stores data to a file or notifies an outside system of a specific event. All of these processes are built into *action elements*.

Action Element	*Encapsulates business logic that performs tasks not affecting the call flow (i.e., has only one exit state).*

An action element can be thought of as a way to insert custom code directly in the call flow of a voice application. A few examples of action elements could be one which retrieves and stores the current stock market price. Another example might be a mortgage rate calculator that stores the rate after using information entered by the caller. A standard Universal Edition installation bundles some pre-built action elements to simplify commonly needed tasks such as sending e-mails and accessing databases.

Since action elements do not affect the call flow, they will always have a single exit state.

The configuration for a configurable action contains two components: settings and variables.

Web Services Element

Web services are a common way for any kind of application to communicate with externally hosted servers to retrieve information or send notification events in a standard manner. Voice applications that wish to access a web service can use the *Web Services* element to do so.

Web Services Element	*A special action element used to interface with a web service.*

The Web Services element is an action element so has the same features: it does not affect the call flow and has a single exit state. The Web Services element, however, has a more complex configuration than a standard action element. Call Studio renders this configuration with its own special interface.

One unique feature of the Web Service element is its ability to configure itself at design time. This is done by loading a Web Services Description Language (WSDL) file. A WSDL file is an XML file that defines the operations supported by the web services server. It is necessary in order to define the inputs required by the service that must be entered by the designer and the outputs returned by the service that can then be stored for use later in the application.

For much more detailed information about how to use the Web Services element, refer to the Call Studio online help.

Flag Elements

One tool an application designer requires is a mechanism where the activities of callers can be analyzed to determine which part of the application is the most popular, creates confusion, or otherwise is difficult to find. To do these analyses, the developer would require knowledge on whether a caller (or how many callers) reached a certain point in the application call flow. This check may also be done within the call itself, changing its behavior dynamically if a caller visited a part of the application previously. To do this, the developer would use f*lag elements*.

Flag Element	*Records when a caller reached a certain point in the call flow.*

Flag elements can be seen as “beacons”, which are triggered when a caller visits a part of the call flow. The application designer can place these flag elements in parts of the call flow that need to be tracked. When the flag is tripped, the application log is updated so that post-call analysis can determine which calls reached that flag. The flag trigger is also stored within the call data so an application can make decisions based on flags triggered by the caller.

Flag elements have a single exit state and do not affect the call flow whatsoever.

Hotlinks

Many voice applications utilize an utterance or key press that when entered by the caller results in the application following a certain path in the call flow. In Universal Edition, these actions are referred to as *hotlinks*.

Hotlink	*An utterance and / or key press that immediately brings the call to a specific part of the call flow or throws an event.*

Hotlinks are not elements in that they do not generate VoiceXML or execute any custom code. Instead, a hotlink acts as a pointer (or link) to direct the call somewhere or throw a VoiceXML event when the right word or key press is detected.

There are two hotlink types: **global hotlinks** and **local hotlinks**. Global hotlinks are activated when the utterance/keypress is detected anywhere in the application. An application can define any number of global hotlinks. An example of a global hotlink is the utterance "operator" (and / or pressing “0”) that transfers callers to a live representative wherever they are in the application.

Local hotlinks are activated only when the utterance or keypress is detected while the caller is within the voice element in which the local hotlink is defined, i.e. that hotlink is “local” to the voice element. Local hotlinks allow the application designer to catch certain utterances or keypresses and handle them in a manner different from how the voice element would handle it. A voice element can define any number of local hotlinks. An example is listening for the utterance “I don’t know” while in a voice element that expects numeric input. Without the hotlink, the element would encounter a no match event because it’s unable to interpret the utterance to a number.

Hotevents

While hotlinks are caller utterances that trigger an action, there are times when the occurrence of a VoiceXML event is expected to trigger an action. The event can be user-triggered (such as a noinput event), asynchronous (which would be thrown by the voice browser), or developer-defined (such as a hotlink that throws an event). In each case, the developer may wish to play audio, store data, or move to another part of the call flow when the event is triggered. In Universal Edition, these are referred to as *hotevents*.

Hotevent	*A global event that when caught, executes developer-specified actions.*

Like hotlinks, hotevents can act as pointers to direct the call somewhere. They may also specify VoiceXML to execute when the event is triggered. An application can utilize any number of hotevents, each activated by a different event.

Note that a hotevent is triggered by a VoiceXML event, not a server-side event such as a Java exception or an error such as a database being down.

Unlike hotlinks, hotevents are all global, there is no such thing as a local hotevent.

Application Reuse

There are many scenarios where a set of smaller applications works better than a single monolithic application. The desire to split up applications into smaller parts centers on reuse – encapsulating a single function in an application and then using it in multiple applications can save time and effort. Additionally updating a single application is much simpler than updating multiple applications with the same change. Call Services provides two different ways to foster application reuse, each with its own unique features.

Application Transfers

There may be instances where a caller in one application wants to visit or "transfer to" another standalone application. This is accomplished with an *application transfer*.

Application Transfer	*A transfer from one voice application to another running on the same instance of Call Services, simulating a new phone call.*

Application transfers do not require telephony routing; they are a server-side simulation of a new call to another application running on the same instance of Call Services. The caller is not aware that they are visiting a new application, but Call Services treats it as if it were a separate call with separate logging, administration, etc. Data captured in the source application can be sent to the destination application (even Java objects) to avoid asking for the same information multiple times in a phone call.

A situation that could utilize application transfers would be a voice portal whose main menu dispatches the caller to various independent applications depending on the caller's choice.

An application transfer is meant to satisfy the need for one independent, standalone application wishes to move the call to another independent standalone application that can also take calls directly. Since an application transfer is used to progress a call from one application to another, it has no exit states.

Subdialogs

There are instances where an application is less independent and really encapsulates some function that multiple applications wish to share. This can be achieved by using a *subdialog*.

Subdialog	*A visit to another Call Services application or other voice application defined in a VoiceXML subdialog context that acts as a voice "service".*

Unlike application transfers that are separate but independent applications, subdialogs are "sub-applications" that an application can visit to handle some reusable functionality and then return back to the source application. It can also take as input application data (though not Java objects) and can also return data for use in the source application. Subdialogs also do not have the restriction that they be deployed on the same instance of Call Services, they can be hosted anywhere accessible via a URL and does not even need to be a Call Services application at all.

The Call Services subdialog is similar to the VoiceXML Insert element but without the requirement to understand VoiceXML. VoiceXML Insert elements are also much more integrated with the rest of the application to be considered an element alternative where a subdialog truly sends control to the subdialog application. For example, hotlinks and hotevents in the source application do not work in the subdialog application where they do in a VoiceXML Insert element.

A situation that could utilize a subdialog would be a third party that develops a sophisticated voice-based authentication system that other applications can use to validate callers. That company exposes their service as a VoiceXML subdialog that takes specific inputs and returns information on the identity of the caller. Any application that wishes to use the service will then use the subdialog element to visit this application.

In order to utilize a subdialog, several special elements are needed in the source and subdialog applications. Visiting a subdialog from an source application requires that it use a *Subdialog Invoke* element.

Subdialog Invoke	*An element used by an application to initiate a visit to a subdialog.*

The Subdialog Invoke element will be treated by the application as an element but will be the gateway to the subdialog. This element handles the inputs and outputs of the subdialog application. While the subdialog application is handling the call, the source application is dormant waiting for the subdialog to return. The Subdialog Invoke element has a single exit state that is followed when the subdialog application returns.

If a Call Services application is to act *as* the subdialog, it uses two different elements: the *Subdialog Start* and the *Subdialog Return* elements.

Subdialog Start	*An element used by a Call Services subdialog application at the start of the call flow to import all variables passed by the source application.*
Subdialog Return	*An element used by a Call Services subdialog application when the subdialog is complete to return data to the source application.*

These elements must be used as the “endpoints” of a subdialog application. The Subdialog Start must be the first element in the application from which the rest of the call flow emerges. The Subdialog Return must be the final element in the call flow (to be used instead of the Hang Up element). An application that does not use these elements can only handle calls made directly to it and cannot be visited by another application as a subdialog.

Call Studio Introduction

Call Studio is a platform for creating, managing, and deploying sophisticated voice applications. Call Studio is developed using the Eclipse framework, though no knowledge of Eclipse is necessary to work with Call Studio. Call Studio acts as a container in which features—called plug-ins—are encapsulated. It includes plug-ins for voice application development, Java programming, and many other features provided by Eclipse.

This section provides a brief introduction on how to license Call Studio, its preferences, creating a new project, and how to access online help. Refer to the online help for much more detailed information on Call Studio.

Licensing

Trial Period

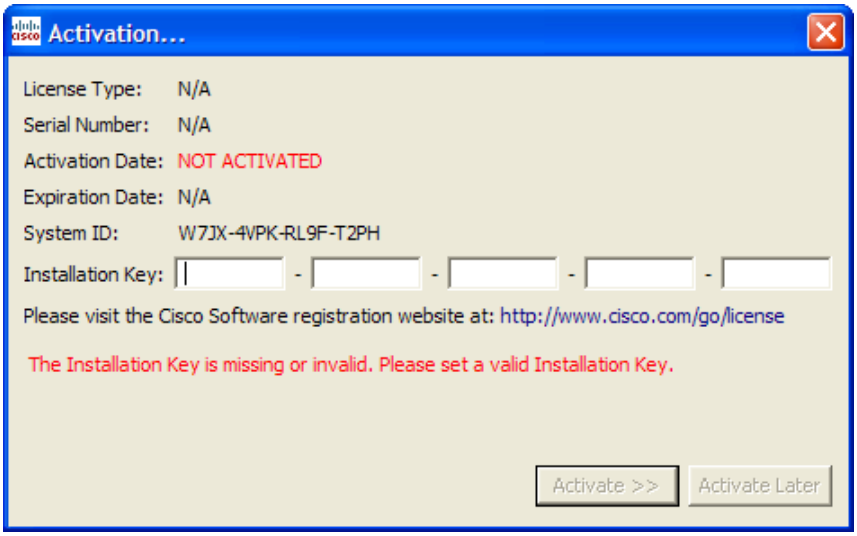
Call Studio can be used for a trial period of 30 days without activation.

Applying a License

When Call Studio is installed and launched for the first time, the software will display an activation dialog. Additionally, each time Call Studio is started without an active license, it will display this dialog.

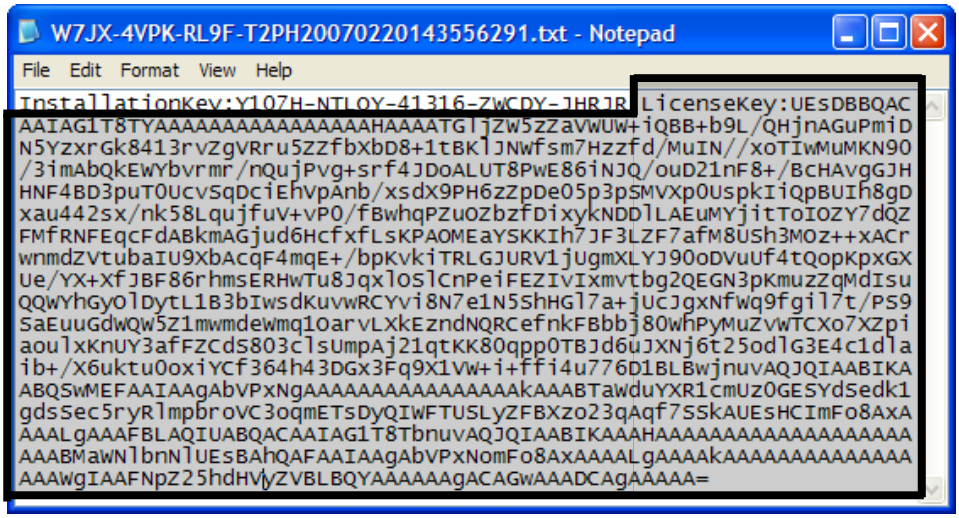
To apply a license to Call Studio:

1. When Call Studio is launched for the first time, it displays an “Activation…” dialog:



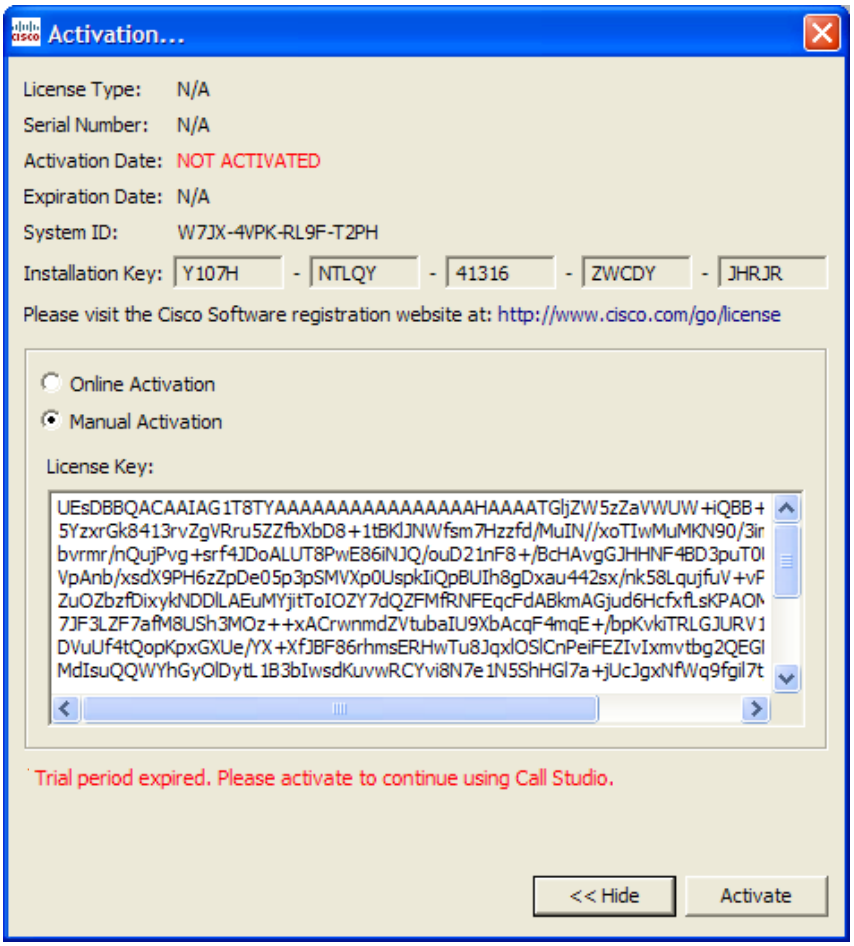
2. Open the license text file that should be applied to this installation; if needed, first visit http://www.cisco.com/go/license to obtain one. The installation key is the first piece of information found in this file, after the label "InstallationKey:" (see highlighted section of image below). Ensure that the installation key is in the format XXXXX-XXXXX-XXXXX-XXXXX-XXXXX (5 groups of 5 characters). Other formats are for other products.

3. The remaining information in the license text file is the license key text. The license key text is composed of all characters after the label "LicenseKey:" (see highlighted section of image below).

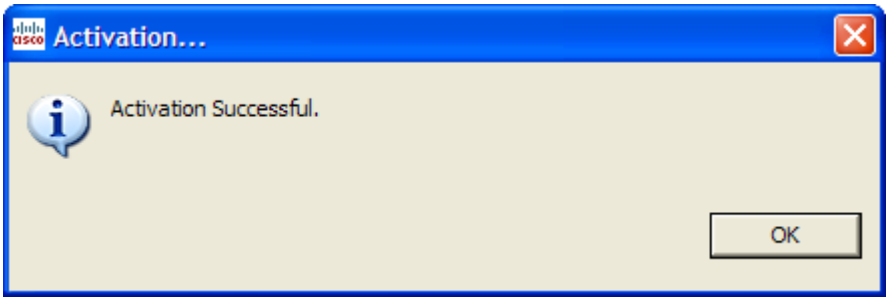


4. Enter the installation key (from step 2) into the matching fields in the "Activation…" dialog.
5. Click the "Activate >>" button.
6. Choose the "Manual Activation" radio button.

7. Paste the license key text (from step 3) into the "License Key:" text area. The dialog should now resemble the image below.



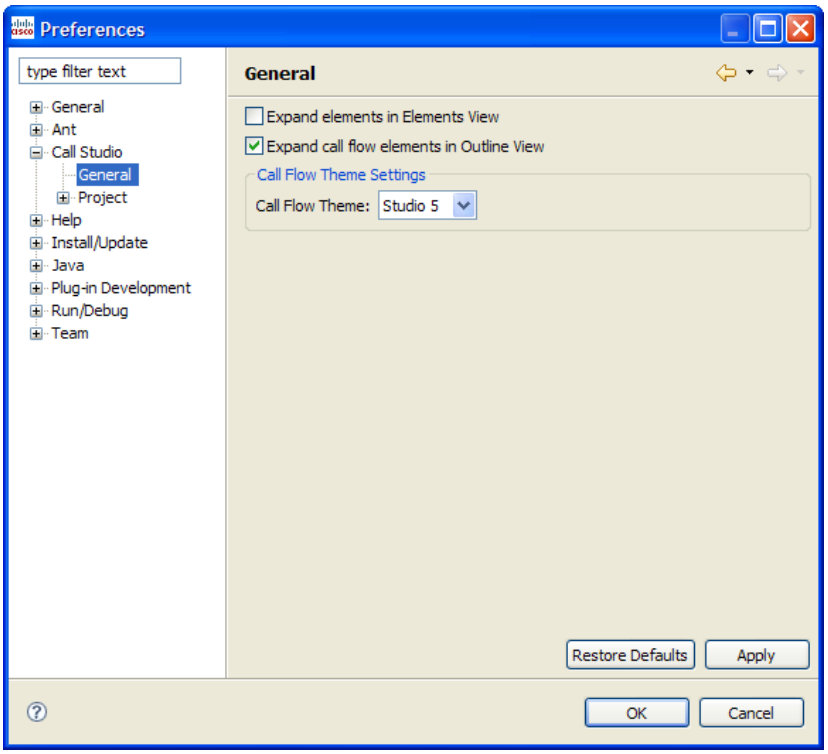
8. Click the "Activate" button, and the "Activation Successful" notification should be displayed. Click "OK" and Call Studio will continue loading. It is now licensed.



Preferences

The Preferences for Call Studio can be set by choosing **Window->Preferences** from the menu bar. Most of the settings listed here apply to the Eclipse platform; however, those listed under **Call Studio** are specifically intended for Call Studio (see below). If modifications are made to

any of these settings, it is recommended that Call Studio be restarted so that the new settings can take effect.



- **Expand elements in Elements View.** This setting controls whether elements in the Elements view appear fully expanded or collapsed (the default).
- **Expand call flow elements in Outline View.** This setting controls whether call flow elements in the Outline view appear fully expanded (the default) or collapsed.
- **Call Flow Theme.** This setting controls the look and feel (i.e., theme) of elements in the Callflow Editor.

Builder for Call Studio

The Builder for Call Studio is a graphical user interface for creating and managing voice applications for deployment on Call Services. Call Services is the runtime framework for Universal Edition voice applications.

A complete dynamic voice application can be constructed within the Builder, including call flow and audio elements. The philosophy behind the Builder is to provide an intuitive, easy-to-use tool for building complex voice applications.

The conception and design of voice applications make use of flowcharts to represent the application call flow. Because flowcharts outline actions are utilized, not the processes behind these actions, they are an effective tool for representing the overall logic of the call flow. The flowcharting process is useful for mapping all the permutations of a call to ensure that all

possible outcomes are handled appropriately. The schematic nature of flowcharts also make it easier for call flow designers to see where callers can get lost or stuck as well as how the call flow can be improved.

The Builder works as a flowcharting program tailored specifically for building voice applications. Most of the familiar features of a flowcharting tool are present in the Builder, with a palette of shapes that can be dragged and dropped onto a workspace and labeled, lines connecting those shapes, multiple pages, and more.

Project Introduction

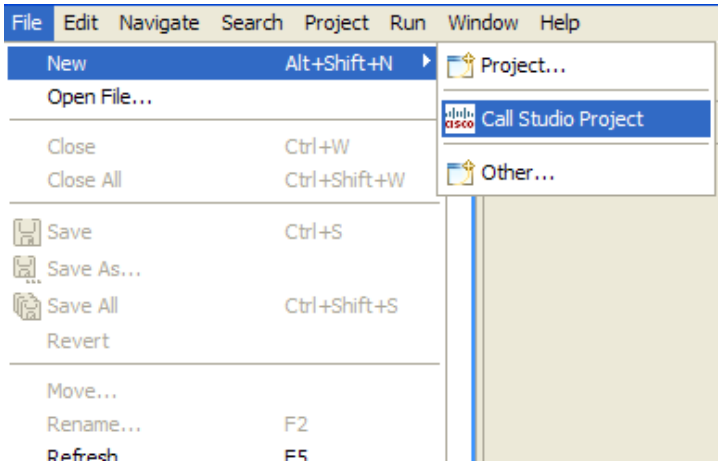
A Call Studio project contains all the resources required to build and deploy voice applications that will run on Call Services.

The `callflow` folder contains the xml files which make up that application's call flow and element configurations. The `app.callflow` file will open the Callflow Editor and graphically represent the call flow based on the information found in these files. Every time the application is saved, these files are updated.

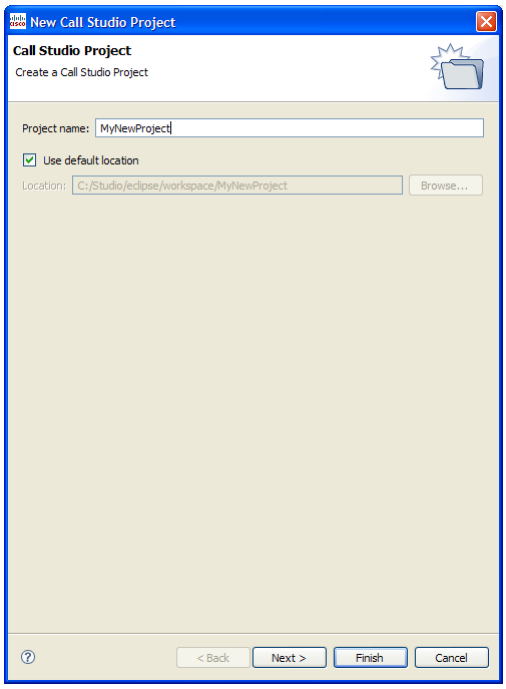
The `deploy` folder contains any extra resources required for deployment; for example, local custom elements and Say It Smart™ plug-ins.

Creating a Call Studio Project

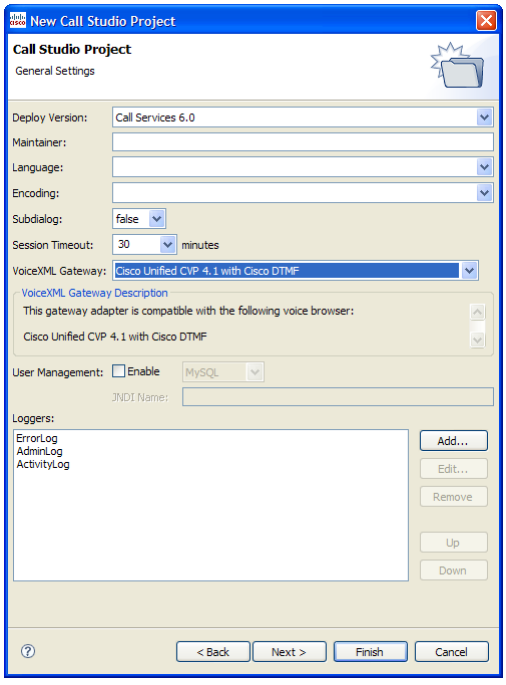
1. To create a Call Studio project, choose **File -> New -> Call Studio Project**.



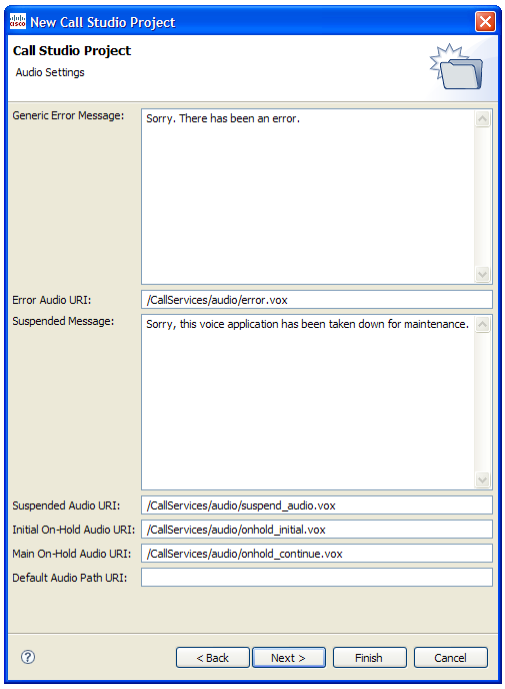
2. Enter a name for the new Call Studio project and select **Next**. Leave **Use default** checked to create the new project in the default workspace directory.



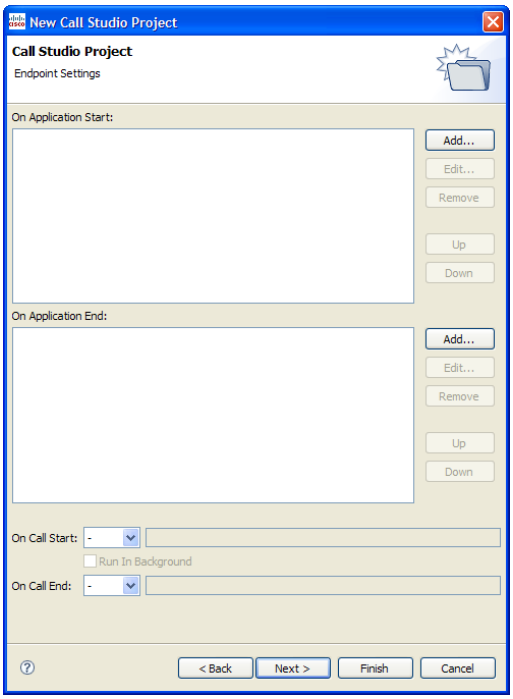
3. Enter the General Settings for the new Call Studio Project and select **Next**. General Settings can always be changed later.



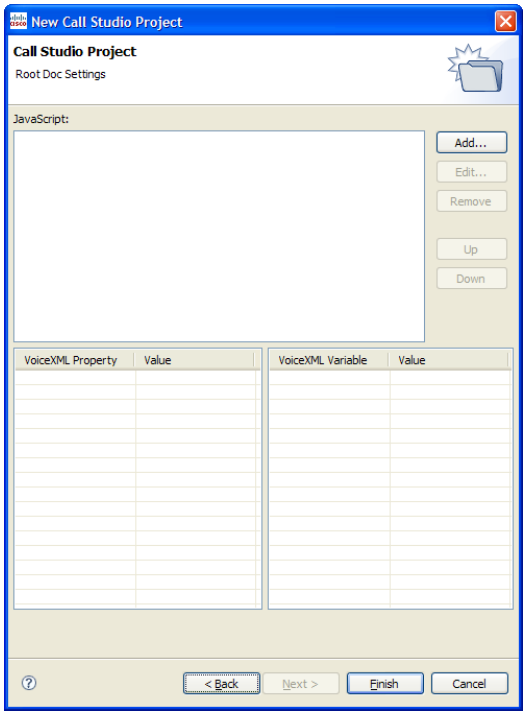
4. Enter the Audio Settings for the new Call Studio Project and select **Next**. Audio Settings can always be changed later.



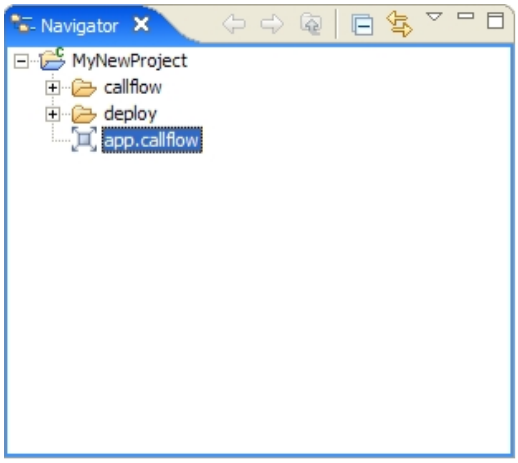
5. Enter the Endpoint Settings for the new Call Studio Project and select **Next**. Endpoint Settings can always be changed later.



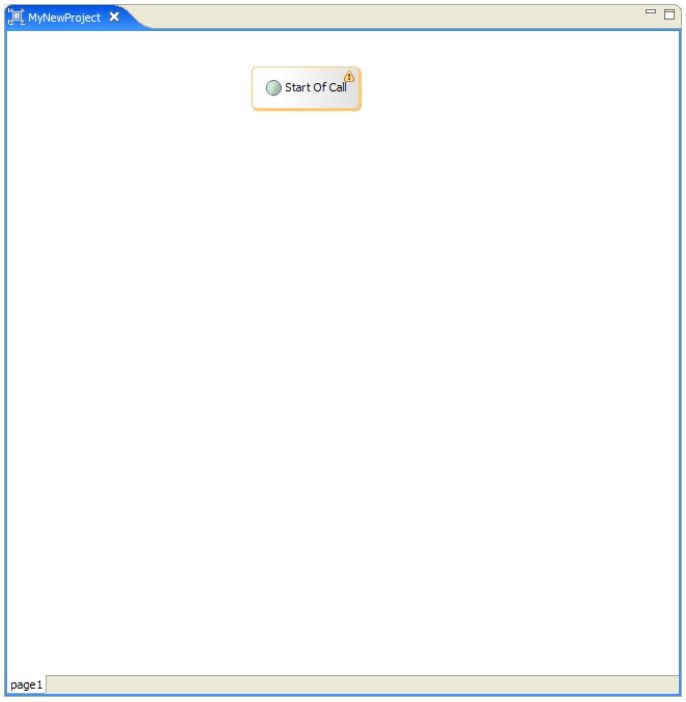
6. Enter the Root Document Settings for the new Call Studio Project and select **Finish**. Root Document Settings can always be changed later.



The new Call Studio Project will appear in the Navigator view.

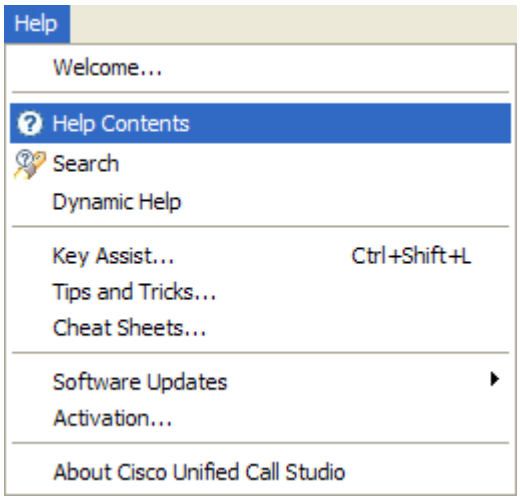


The new application's call flow will automatically open in the Callflow Editor.



Online help

Detailed descriptions of all Call Studio features, element types, and functionalities can be found in Call Studio's online help. This comprehensive online help can be accessed via the **Help -> Help Contents** menu option:



Chapter 2: Universal Edition Components in Detail

Some components of Call Services require detailed explanations on how to use them properly, especially when their functionality requires or is extended by programming. While it is certainly possible to create a voice application entirely dependent on fixed data, most dynamic applications will require some programming work.

It is important for the non-developer user to be aware of these components and the functions they serve. The application designer will need to understand in what situations various components are needed so that a comprehensive specification can be given to a developer responsible for building these components.

This chapter describes these components in more detail, explaining typical situations where they would be used. It also describes the Universal Edition concepts utilized in order to develop and use the components. The Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio describes the components that require programming from the developer's standpoint, explaining the process of constructing and deploying them. One can think of the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio as a comprehensive description of what this chapter introduces.

Components

The components discussed in this chapter are:

- **Built With Programming**. These components require some programming effort.
  - **Call-Specific**. These components are built to be used within individual calls.
    - **Custom Configurable Elements.** A developer may wish to create their own reusable, configurable elements to supplement the elements Universal Edition provides.
    - **Standard Action and Decision Elements.** For situations where unique, application-specific functionality is needed, thereby not requiring the flexibility and complexity of configurable elements.
    - **Dynamic Element Configurations.** For situations where the configuration for a configurable element can only be determined at runtime.
    - **Start and End of Call Action.** To perform tasks before each call begins and/or after each call ends.
    - **Hotevents.** To specify the VoiceXML to execute when a certain VoiceXML event occurs.
    - **Say It Smart Plugins.** To play back additional formatted data or to extend existing Say It Smart behavior.

- **Call Services-Specific**. These components are built to run on Call Services as a whole and do not apply to a specific call.
  - **Start and End of Application Actions**. To perform tasks when a Call Services application is loaded and/or shuts down.
  - **Loggers**. Plugins designed to listen to events that occur within calls to an application and log or report them.
  - **On Error Notification.** To perform tasks if an error causes the phone call to end prematurely.
- **Built Without Programming**. These components do not require high-level programming effort to construct.
  - **XML Decisions.** Universal Edition provides an XML format for writing simple decisions without programming. The exact XML format is detailed in this chapter.
  - **VoiceXML Insert Elements.** This element is used in situations where the developer wishes to incorporate custom VoiceXML content into a Universal Edition application. The guidelines for building a VoiceXML insert element are given in this chapter.

Variables

Universal Edition offers variables as a mechanism for components to share data with each other, in four forms: global data, application data, session data and element data.

Global Data

A global data variable is just that, it is globally accessible and modifiable from all calls to all applications. Global data is given a single namespace within Call Services that is shared across all calls to all applications. If a component changes global data, that change is immediately available to all calls for all applications. Global data can hold any data, including a Java object. The lifetime of global data is the lifetime of Call Services. Global data would be reset if the application server were to be restarted or the Call Services web application archive (WAR) were to be restarted.

Global data is typically used to store static information that needs to be available to all components, no matter which application they reside in. For example, the holiday schedule of a company that applies to all applications for that company.

Application Data

An application data variable is accessible and modifiable from all calls to a particular application. Application data variables from one application cannot be seen by components in another application. Each application is given its own namespace to store application data. If a component changes application data, that change is immediately available to all other calls to the

application. Application data can hold any data, including a Java object. The lifetime of application data is the lifetime of the application. Application data would be reset if the application were updated and would be deleted if the application were released.

Application data is typically used to store application-specific information that does not change on a per call basis and is to be available to all calls. For example, the location of a database to use for the application.

Session Data

Session data variables are accessible and modifiable from a single call session. Session data variables in one call cannot be accessed by components handling another call. Each session has its own session data namespace - session data set by one component will overwrite existing session data that has the same name. Session data can hold any data, including a Java data structure. The lifetime of session data is the lifetime of the session or the call. When the call ends, the session data is deleted.

Any component accessed within a call session, including elements, can create, modify and delete session data. Session data can even be created automatically by the system in two ways:

- If the voice browser passes additional arguments to Call Services when the call is first received, these additional arguments will be added as session data with the arguments' name/value pairs translated to the session data name and value (both as `Strings`). For example, if the voice browser calls the URL:

  ```
  http://myserver.com/CallServices/Server?audium_application=MyApp&SomeData=1234
  ```

  this will create session data named "SomeData" with a value of "1234" in every call session of the application "MyApp" that starts via this URL.
- If a Universal Edition voice application performs an application transfer to another application and the developer has chosen to pass data from the source application to the destination application, then this data will appear as session data in the destination application (the data is renamed before it is passed to the destination application). Please refer to the Call Studio documentation for more information on application transfers.

Element Data

Element data variables are accessible from a single call session and modifiable from a single element within that call session. As the name suggests, element data can only be created by elements (excluding start and end of call events, the global error handler, hotevents, and XML decisions). Dynamic configurations are technically part of an element since they are responsible for configuring an element, so they can also create element data. Only the element that created an element data variable can modify or delete it, though it can be read by all other components. Due to the fact that the variable belongs to the element, the variable namespace is contained within

the element, meaning two elements can define element data with the same name without interfering with each other. To uniquely identify an element data variable, both the name of the element and the name of the variable must be used. Like session data, the lifetime of session data is the lifetime of the session or the call. When the call ends, the element data is deleted.

Component Accessibility

Table 2-1 lists each component and its ability to get and set global, application, session, and element data.

Component	Global Data		Application Data		Session Data		Element Data	
	Get	Set	Get	Set	Get	Set	Get	Set
Configurable Elements	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Standard Elements	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Configurations	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Start and End of Call Actions	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hotevents	No	No	No	No	No	No	No	No
Say It Smart Plugins	No	No	No	No	No	No	No	No
Start and End of Application Actions	Yes	Yes	Yes	Yes	No	No	No	No
Loggers	Yes	No	Yes	No	Yes	No	Yes	No
On Error Notification	No	No	No	No	Yes	No	No	No
XML Decisions	No	No	No	No	No	No	No	No
VoiceXML Insert Elements	No	No	No	No	Yes	Yes	Yes	Yes

Table 2-1

Notes:

- Hotevents, being simply VoiceXML code appearing in the root document, do not have access to any server-side information.
- A Say It Smart Plugin’s sole purpose is to convert a value into a list of audio files and so do not have a need to access server-side information.
- A Logger's sole responsibility is to report or log data and therefore has access to all variables types but cannot set them.
- On Error Notification classes are given the session data that existed at the time the error occurred.

APIs

To facilitate the development of components requiring programming effort, Universal Edition provides two application programming interfaces (APIs) for developers to use. The first is a Java

API. The second API involves the use of XML sent via HTTP, thereby allowing components to be built using programming languages other than Java. Some more complex and tightly integrated components can be built only through the Java API, though in most other aspects, the two APIs are functionally identical. The APIs themselves and the process of building components using either API is fully detailed in the Javadocs published with the software and in the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio. The two components that do not require the use of high-level programming, XML decisions and VoiceXML insert elements, are fully explained in this document.

The APIs are used to interface with Call Services in order to retrieve data or change information. The API provided to each component has slightly different functionality reflecting each component's unique abilities. The following lists abilities provided by the API that is common to most components used within a callflow:

- Getting call information such as the ANI, DNIS, call start time, application name, etc.
- Getting or setting global data, application data, element data or session data.
- Getting information about the application's settings such as the default audio path, voice browser, etc.
- Setting the maintainer and default audio path. Changing the maintainer allows multiple people to maintain different parts of a single application. Changing the default audio path allows an application to change the persona or even language of the audio at any time during the call.
- Sending a custom event to all application loggers (see Chapter 5: Call Services Logging for more on logging with Call Services).

Table 2-2 shows which API can be used to construct the various components listed.

Call Services Component	***Build With Java API***	***Build Using XML-over-HTTP API***	***VoiceXML Knowledge Suggested***
Configurable Action and Decision Elements	Yes	No	No
Configurable Voice Elements	Yes	No	Yes
Standard Elements	Yes	Yes	No
Dynamic Element Configurations	Yes	Yes	No
Start or End of Call Actions	Yes	Yes	No
Hotevents	Yes	No	Yes
Say It Smart Plugins	Yes	No	No
Start and End of Application Actions	Yes	No	No
Loggers	Yes	No	No
On Error Notification	Yes	No	No
XML Decisions	NA	NA	NA
VoiceXML Insert Elements	NA	NA	Yes

Table 2-2

Configurable Elements

Most of the elements in a typical Universal Edition application are pre-built, reusable elements whose configurations are customized by the application designer. Using a configurable element in a call flow requires no programming or VoiceXML expertise and since they can encapsulate a lot of functionality, greatly simplifies and speeds up the application building process. Call Services includes dozens of elements that perform common tasks such as collecting a phone number or sending e-mail. A need may exist, however, for an element with functionality not available in the default installation. Additionally, while Universal Edition elements have been designed with configurations that are as flexible as possible, there may be situations where a desired configuration is not supported or difficult to implement.

To satisfy these concerns, a developer can construct custom configurable elements that, once built, can be used and reused. The developer can design the element to possess as large or as small a configuration as desired, depending on how flexible it needs to be. Once deployed, custom elements appear in Builder for Call Studio in the Element Pane and are configured in the same way as Universal Edition Elements.

Due to the level of integration with the Universal Edition software required, only the Java API provides the means for building configurable elements. Using this API, configurable action, decision, and voice elements can be built. Voice elements, due to the fact that they are responsible for producing VoiceXML, use an additional Java API, the Voice Foundation Classes (VFCs). The VFCs are used to abstract the differences between the various voice browsers supported by Universal Edition. The VFCs follow a design that parallels VoiceXML itself and only a developer familiar with VoiceXML and the process whereby a voice browser interprets VoiceXML will be fully suited to utilize the VFCs to build voice elements.

The Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio describes the process of building configurable elements including detailing the VFC API for building voice elements.

Standard Action and Decision Elements

Unlike configurable action or decision elements, a standard action or decision element is designed more as a one-off as they satisfy an application-specific purpose. As a result, standard action and decision elements do not require configurations.

There are many situations where programming effort is required to perform some task specific to an application. Since the task is very specialized, pre-existing reusable elements are too general to perform the effort. Additionally, building a configurable element for this purpose would be overkill since there is little chance it would be needed anywhere but in this application. The developer would use a standard action or decision element to perform just this task. If the task is

applicable to multiple situations, the developer most likely would put in the extra effort to construct a configurable, reusable element.

Universal Edition provides a means of defining standard decision elements without programming by writing an XML document directly within Builder for Call Studio. This format should be investigated when desiring simple or moderately complex standard decision elements, falling back on the programming API should the built-in format prove insufficient. The XML format that the Builder for Call Studio user interface produces for standard decision elements is described later in this chapter.

Dynamic Element Configurations

Each configurable voice, action, and decision element used in an application must have a configuration. Usually, the configuration will be fixed - it acts the same for every caller that visits it. In these situations, the designer using Builder for Call Studio creates this configuration in the Configuration Pane. This configuration is saved as an XML file when the application is deployed.

There are situations, though, that a configuration for an element depends on information known only at runtime – it is dynamic. An example would be to configure the Universal Edition audio voice element to play a greeting message depending on the time of the day. Only at runtime does the application know the exact calling time and therefore what greeting message to play.

To produce dynamic configurations, programming is required. Dynamic element configurations are responsible for taking a base configuration (a partial configuration created in the Builder for Call Studio), adding to it or changing it depending on the application business logic, and returning the desired element configuration to Call Services.

Start / End of Call Actions

Universal Edition provides mechanisms to execute some code when a phone call is received for a particular application or when the call ends. The end of a call is defined as either a hang up by the caller, a hang up by the system, a move from one Universal Edition application to another Universal Edition application, or other rarer ways for the call to end such as a blind transfer or session timeout.

The purpose of the start of call action is typically to set up dynamic information that is used throughout the call, for example, the current price of a stock or information about the caller identified by their ANI in some situations. The end of call action is typically used to export information about the call to external systems, perform call flow history traces, or execute other tasks that require information on what occurred within the call.

The start of call action is given the special ability to change the voice browser of the call. This change applies 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 compatible with the appropriate browser. This task can only be done in the start of call action because the call technically has not started when this action occurs.

The end of call action is given the special ability to produce a final VoiceXML page to send to the browser. Even though the caller is no longer connected to the browser by the time the end of call action is run, some voice browsers will allow for the interpretation of a VoiceXML page sent back in response to a request triggered by a disconnect or hang-up event. Typically this page will perform final logging tasks on the browser.

Hotevents

A hotevent is some developer-defined action performed whenever a VoiceXML event is triggered. Hotevents can take two forms. The first simply moves the caller to a new part of the call flow when the event is triggered. In this case the hotevent can be defined entirely in the Builder for Call Studio by the application designer and does not require any additional work by a developer. The second form of a hotevent executes custom VoiceXML content when the event is triggered (and may optionally also move to another part of the call flow).

The second form requires a Java class that returns the VoiceXML to execute when the event is triggered. This class is accessed once per call, *not* when the event is triggered. This is because hotevents appear in the VoiceXML root document, which is generated when the application is started or updated. For this reason the API provided to hotevents do not have access to call information normally given to other components. The purpose of a hotevent is solely to produce VoiceXML that will execute when the event is triggered.

A hotevent can be created to react based on a standard VoiceXML event such as nomatch or noinput, but a more common use for a hotevent is to be triggered by a developer-specified or a voice browser-specific event. For example, one can create a hotevent to play audio when a custom event occurs.

As with any other component that produces VoiceXML, the hotevent utilizes the VFCs and therefore can only be constructed with the Java API.

Say It Smart Plugins

In Call Services, developers can create their own Say It Smart plugins. Similar to custom elements, Say it Smart plugins are pre-built Java classes that when deployed in the Builder for Call Studio can be used as a new Say It Smart type. As with custom elements, the level of integration required with the Universal Edition software restricts the creation of Say It Smart plugins to the Java API.

Custom Say It Smart plugins can be constructed to read back formatted data not handled by Universal Edition Say It Smart plugins, such as spelling playback or reading the name of an airport from its three-digit code. Plugins can also be created to extend the functionality of existing plugins, such as adding new output formats to play the information in another language. For example, a plugin can define a new output format for the Universal Edition Date Say It Smart plugin that reads back dates in Spanish.

Refer to the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for a full description of the process of building custom Say It Smart plugins.

Start and End of Application Actions

Universal Edition provides mechanisms to execute some code when an application is launched or shut down. A start of application action is run when the Call Services web application archive (WAR) starts up (which occurs when the application server first starts up or the application server reloads the WAR), or the application is updated. An end of application action is run when the application is updated, released, or the web application is shut down (which occurs if the application server reloads or shuts down the web application or the application server itself is shut down).

The purpose of the start of application action is typically to set up global data or application data that would be accessed by components within the callflow. Since global and application data's lifetime is the lifetime of the application and they can contain Java objects, the start of application action could even set up persistent database connections or other communications to external systems that would remain connected as long as the application were running. Note that should an error occur within the start of application class, the application deployment will continue unchanged. The designer can specify that an error in a particular start of application class should stop the application deployment. This would be done if the class performs mandatory tasks that are necessary for the application to run correctly.

The purpose of the end of application action would be to clean up any data, database connections, etc. that would no longer be needed once the application is shut down. Note that the end of application action is called even when the application is updated because the update may have changed the data that is needed by the application.

Every application deployed on Call Services has the ability to define any number of start and end of application actions that are executed in the order in which they appear in the application settings.

Loggers

The act of logging information about callers to the system is performed by loggers. An application can reference any number of loggers that "listen" for logging events that occur. These events range from events triggered by a call, such as a caller entering an element or

activating a hotlink to administration events such as an application being updated to errors that may have been encountered. Loggers can take the information on these events and do whatever desired with them. Typically the logger will store that information somewhere such as a log file, database or reporting system.

Call Services includes default loggers that store the information obtained from logging events to parse-able text log files. A need may exist, however, for a logger with functionality not available in the default installation or a logger that takes the same data and stores it using a different mechanism.

To satisfy these concerns, a developer can construct custom loggers that listen for logger events and report them in their own way. The developer can design the logger to use a configuration to customize how the logger functions, depending on how flexible it needs to be. Due to the level of integration with the Universal Edition software required, only the Java API provides the means for building loggers.

Refer to Chapter 5: Call Services Logging in the section entitled Application Loggers for descriptions of the loggers included with Call Services. Refer to the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for a description of the process of building custom loggers.

On Error Notification

When errors occur on the Call Services, the application-specific error voice element will handle how to handle the caller. If specified, the on error notification Java class can be configured to be activated when an error occurs. The class is given information about the application and some basic call information to allow the developer to specify the action accordingly. The developer can write this class to perform whatever they wish.

The most common purpose for the on error notification class is to perform some custom notification, something to indicate at runtime that an error occurred. This could involve paging an administrator or integrating with a third-party trouble ticket and notification process. Since the notification usually involves an administrator whose responsibility is the entire Call Services, the Java class, once specified, will apply to any error that prematurely ends a call on any Universal Edition application.

Note that this class is used for notification purposes; it does not allow the call to recover from the error. Note also that there is no XML API equivalent for the on error notification; if done at all, it must be written in Java.

Universal Edition XML Decisions in Detail

Many commercial applications with decisions driven by business logic utilize an external rules engine to codify the definition of rules. These rules engines help describe the definition of a rule

and then manage the process of making decisions based on the criteria at hand. Call Services bundles a rule engine in the standard installation and provides an XML data format for defining decision elements within the framework of a voice application. The XML format is simple enough for an application designer to enter within Builder for Call Studio without requiring separate programming resources.

A detailed description of the structure of the XML format is warranted. The centerpiece of a rule is one or more expressions. An expression is a statement that evaluates to a true or false. In most cases, there are two parts (called terms) to an expression with an operator in between. The terms are defined by Call Services to represent all of the most common items one would want to base decisions on in a voice application such as telephony data, element or session data, times and dates, caller activity, user information, etc. The operators depend on the data being compared. For example, numbers can be compared for equality or greater than or less than while strings can be compared for equality or if it “contains” something. One kind of expression breaks this format: an “exists” expression which itself evaluates to a true or false and does not need anything to compare it to. For example: “has this caller called before?” or “does the system have a social security number for the user?” Each of these checks for the existence of something which is itself a complete expression.

One or more of these expressions are combined to yield one exit state of the decision element. Multiple expressions can be combined using “ands” or “ors”, though not a combination. For example: “if the ANI begins with 212 OR if the ANI begins with 646 then return the exit state ‘Manhattan’”. If a combination of “ands” and “ors” is desired, multiple expressions that return the same exit state would be used. For example, “if the ANI begins with 212 and the user is a gold or platinum customer, then return the exit state ‘discount’” would not work as a single rule because the discount would be given to callers with a 212 area code who are gold customers and all platinum customers (there is no way to set precedence). This would have to be expressed as two rules with the same exit state: 1) “if the ANI begins with 212 AND the user is a gold customer, return the exit state ‘discount’” and 2) “if the ANI begins with 212 AND the user is a platinum customer, return the exit state ‘discount’”.

It is possible to define an exit state that returns when all other exit states fail to apply, called the default exit state. When not specified, all possible cases must be caught by the defined rules. For example, if a rule checks if a number is greater than 5, there should be another rule checking if the number is less than or equal to 5, unless the default exit state is defined. One can even create a set of rules that start from being restrictive, looking for only very specific matches, to progressively looser since the first rule to be true will yield an exit state and no more rules are tested.

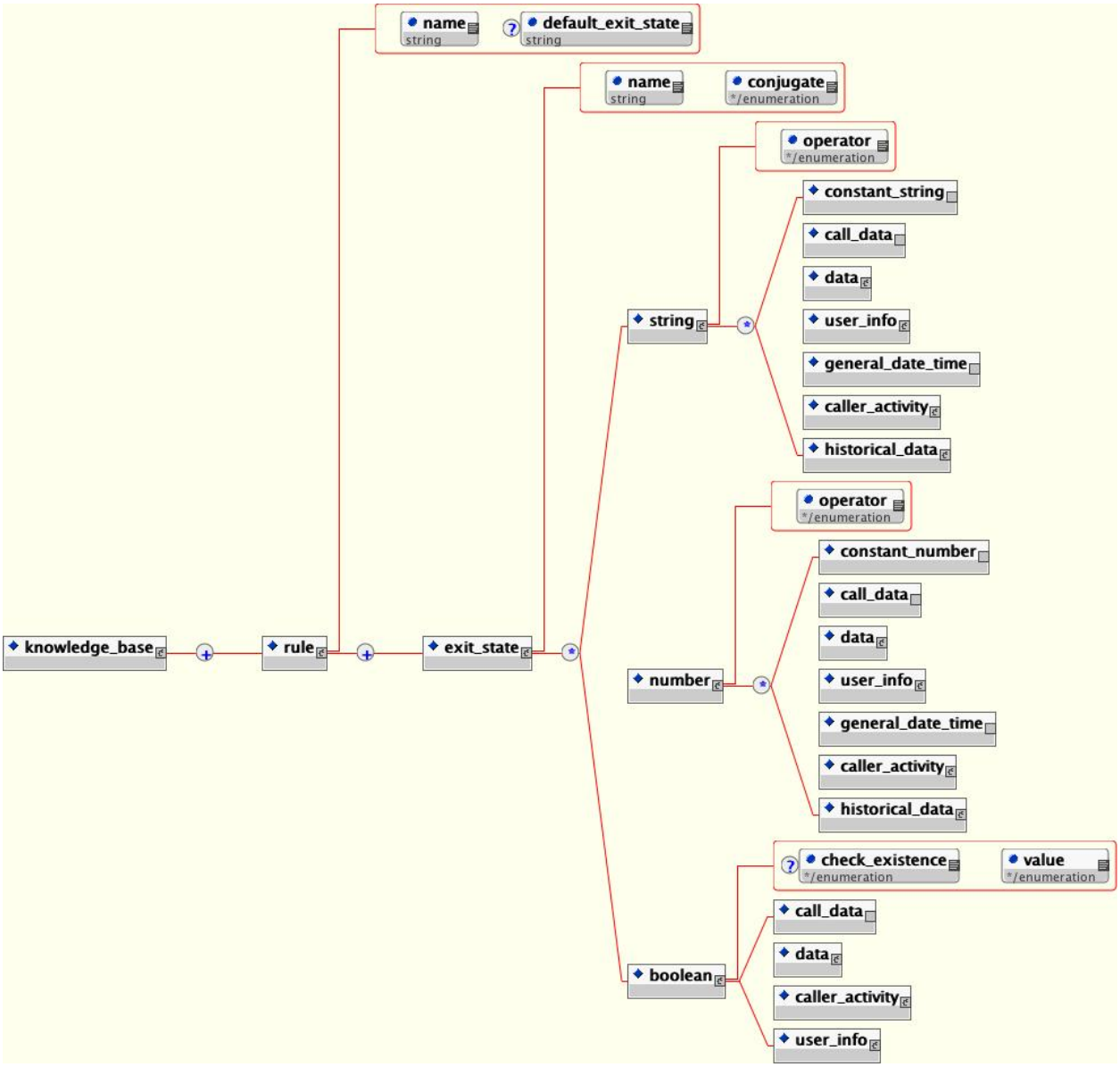


Figure 2-1

Figure 2-1 shows the main tags of the XML file format for defining a decision. The elements in this XML document are:

- **rule** – This tag names the rule for the decision. There can only be one `<rule>` tag in the document. The tag contains any number of exit states that make up the decision. The optional `default_exit_state` attribute lists the exit state to return if no other exit states apply (essentially an "else" exit state).
- **exit_state** – This tag encapsulates the expressions that when true, return a particular exit state. The `name` attribute must refer to the same value chosen when the decision element was

defined in the Builder for Call Studio. The `conjugate` attribute can be either *and* or *or*. If the exit state contains only one expression the `conjugate` attribute is ignored. The content of the `<exit_state>` tag is the type of data to be compared, each type containing different kinds of data. There can be any number of children to the `<exit_state>` tag, each representing another expression linked with the conjugate.

- **string** – This tag represents an expression comparing strings. The `operator` attribute can be: *contains*, *not_contains*, *ends_with*, *not_ends_with*, *equal*, *not_equal*, *starts_with*, and *not_starts_with*. There can be only two children to the `<string>` tag, representing the two terms of the expression. If there are less than two, an error will occur. If more, the extra ones will be ignored. The content can be tags representing a constant string entered by the developer, data about the call, session and element data, user information, date and time information, the activity of the caller, and historical activity of the user. These tags are fully defined in the following sections.
- **number** – This tag represents an expression comparing numbers. The `operator` attribute can be: *equal*, *not_equal*, *greater*, *greater_equal*, *less*, and *less_equal*. There can be only two children to the `<number>` tag, representing the two terms of the expression. If there are less than two, an error will occur. If more, the extra ones will be ignored. The content can be tags representing a constant number entered by the developer, data about the call, session and element data, user information, date and time information, the activity of the caller, and historical activity of the user. These tags are fully defined in the following sections.
- **boolean** – This tag represents an expression which evaluates to a boolean result, requiring only a single term. If the `check_existence` attribute is *yes*, and the `value` attribute is *true*, it is checking if the data defined by the child tag exists. If `check_existence` is *yes*, and `value` is *false*, it is checking if the data defined by the child tag does not exist. If `check_existence` is *no*, the `value` attribute is used to compare the data defined by the child tag with either true or false. *True* means the expression is true if the data defined by the child tag evaluates to true. The child tags are a smaller subset of those allowed in `<string>` and `<number>`: data about the call, session and element data, user information, or the activity of the caller (each of these is fully defined in the following sections). When testing if the child tag's value is true or false, it must be able to evaluate to a boolean value. If it cannot, the decision will act as if the rule did not activate.
- **constant_string / constant_number** – These tags store string and number data in the `value` attribute. The number can be any integer or floating-point number. Note that the number can also be treated as a string. For example "if 1234 starts with 12".

The following sections explain the contents of the individual tags found within the `<string>`, `<number>` and `<boolean>` tags.

The <call_data> Tag

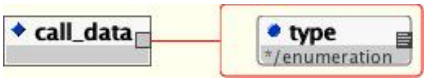


Figure 2-2

Figure 2-2 shows the term that represents information about the current call. The `type` attribute can be *ani*, *dnis*, *uui*, *iidigits*, *source*, *appname*, *duration*, *language*, or *encoding*. The ANI, DNIS, UUI, and IIDIGITS will be "NA" if it is not sent by the telephony provider. *Source* is the name of the application that transferred to this application or `null` if this application was the first to be called. *Duration* is the duration of the call up to this point in seconds.

The <data> Tag

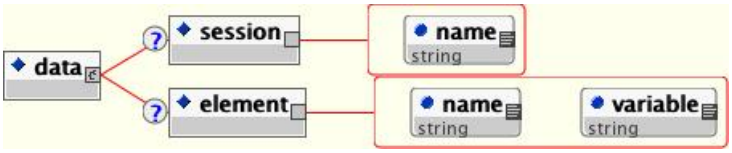


Figure 2-3

Figure 2-3 shows the term that represents session or element data. The `<session>` tag refers to session data with its name in the `name` attribute. The `<element>` tag refers to element data with the name of the element in the `name` attribute and the name of the variable in the `variable` attribute.

The <user_info> Tag

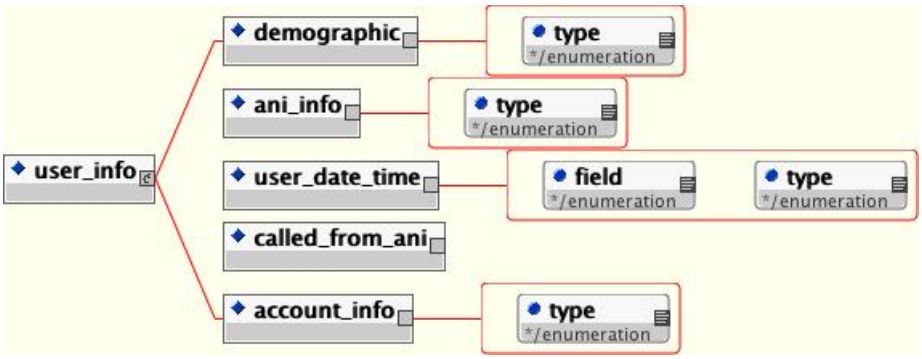


Figure 2-4

Figure 2-4 shows the term that represents user information. Note that if the application has not been configured to use the user management system, and the call was not associated with a specific UID, using this term will cause an error. Only one piece of user information can be returned per tag. Refer to Chapter 4: User Management for more details about the user management system. The possible user information to be compared is:

- **demographic** – This tag refers to the user's demographic information. The `type` attribute can be *name*, *zipcode*, *birthday*, *gender*, *ssn*, *country*, *language*, *custom1*, *custom2*, *custom3*, or *custom4*.
- **ani_info** – This tag refers to the various phone numbers associated with the user account. If the `type` attribute is *first*, the first number in the list of numbers is returned. This would be returned if there was only one number associated with an account. If the attribute is *num_diff* the total number of different phone numbers associated with the account is returned.
- **user_date_time** – This tag refers to date information related to the user account. The `type` attribute indicates which user-related date to access and the `field` attribute is used to choose which part of the date to return. `Type` can be *last_modified* (indicating the last time the account was modified), *creation* (indicating the time the account was created), and *last_call* (indicating the last time the user called). `Field` can be *hour_of_day* (which returns an integer from 0 to 23), *minute* (which returns an integer from 0 to 59), *day_of_month* (which returns an integer from 1 to 31), *month* (which returns an integer from 1 to 12), *day_of_week* (which returns an integer from 1 to 7 where 1 is Sunday), or *year* (which returns the 4 digit year).
- **called_from_ani** – This tag returns "true" if the caller has previously called from the current phone number, "false" if not.
- **account_info** – This tag refers to the user's account information. The `type` attribute can be *pin*, *account_number*, or *external_uid*.

The <general_date_time> Tag

Figure 2-5

Figure 2-5 shows the term that represents general date information. The `type` attribute indicates which date to access and the `field` attribute is used to choose which part of the date to return. `Type` can be *current* (indicating the current date/time) or *call_start* (indicating the time the call began). `Field` can be *hour_of_day* (which returns an integer from 0 to 23), *minute* (which returns an integer from 0 to 59), *day_of_month* (which returns an integer from 1 to 31), *month* (which returns an integer from 1 to 12), *day_of_week* (which returns an integer from 1 to 7 where 1 is Sunday), or *year* (which returns the 4 digit year).

The <caller_activity> Tag

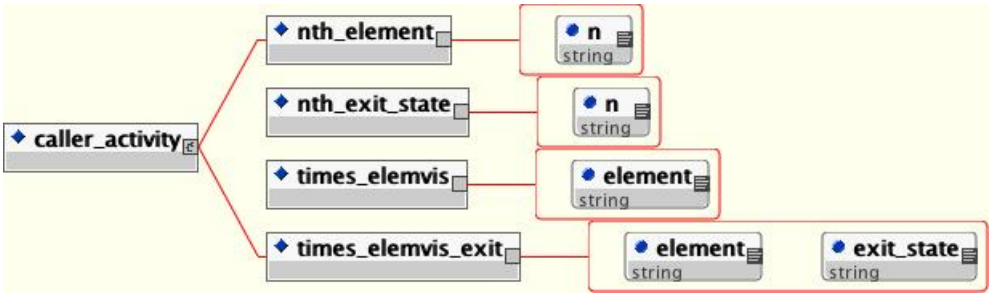


Figure 2-6

Figure 2-6 shows the term that represents the activity of the caller in the current call. The `<nth_element>` tag returns the nth element visited by the caller where the attribute `n` is the number (starting at 1). The `<nth_exit_state>` tag returns the exit state of the nth element visited by the caller where the attribute `n` is the number (starting at 1). The `<times_elemvis>` tag returns the number of times the caller visited the element whose name is given in the `element` attribute. The `<times_elemvis_exit>` tag returns the number of times the caller visited the element whose name is given in the attribute `element` which returned an exit state whose name is given in the `exit_state` attribute.

The <historical_data> Tag

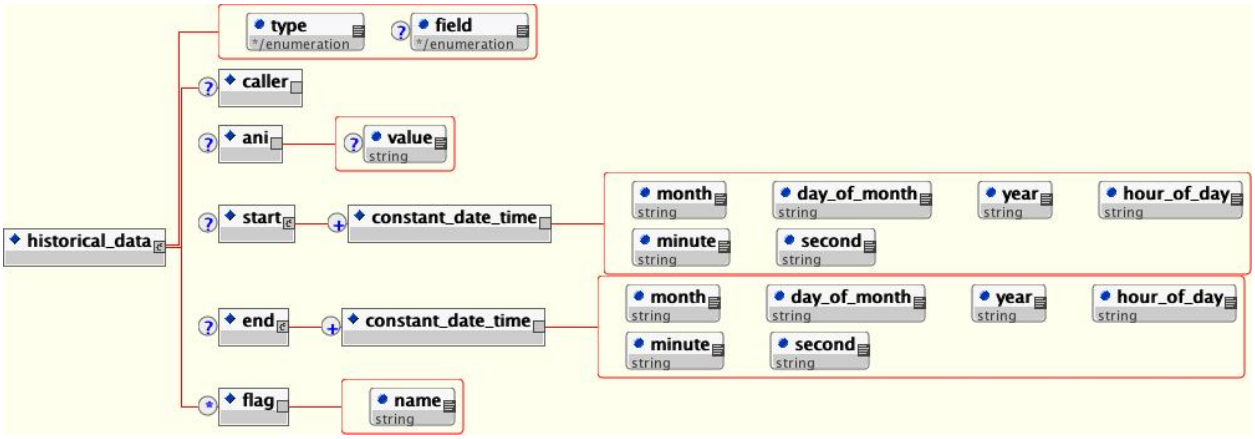


Figure 2-7

Figure 2-7 shows the term that represents the historical activity of the user associated with the call on the current application. Note that if the application has not been configured with a user management database, using this term will cause an error. Refer to Chapter 4: User Management for more details about the user management system. The `type` attribute determines what kind of value is returned. A value of *num* means that the value returned is the number of calls matching the criteria defined by the children tags. A value of *last_date_time* means that the value returned is the last date/time a call was received matching the criteria defined by the children tags. A value of *first_date_time* returns the first date/time a call was received that matched the criteria. The `field` attribute is used if the `type` attribute is *first_date_time* or *last_date_time* and indicates

which part of the date to compare. `Field` can be *hour_of_day* (which returns an integer from 0 to 23), *minute* (which returns an integer from 0 to 59), *day_of_month* (which returns an integer from 1 to 31), *month* (which returns an integer from 1 to 12), *day_of_week* (which returns an integer from 1 to 7 where 1 is Sunday), or *year* (which returns the 4 digit year). The children tags are used to turn on various criteria to add to the search. The different search criteria are:

- **caller** – If this tag appears, the search looks for calls made by the current caller only. If it does not appear, it will search all calls made by all callers. Note that if the call was not associated with a specific UID, an error will occur if this tag is used.
- **ani** – If this tag appears, the search looks for calls made by the ANI specified in the `value` attribute. If the `value` attribute is not included, the ANI of the current caller is used.
- **start** – If this tag appears, the search looks for calls whose start date/time are between two times specified by successive `<constant_date_time>` children tags. The attributes of `<constant_date_time>` define the specific date to use. The `month` attribute must be an integer from 1 to 12. The `day_of_month` attribute must be an integer from 1 to 31. The `year` attribute must be a four digit integer. The `hour_of_day` attribute must be an integer from 0 to 23. The `minute` attribute must be an integer from 0 to 59. The `second` attribute must be an integer from 0 to 59.
- **end** – If this tag appears, the search looks for calls whose end date/time are between two times specified by successive `<constant_date_time>` children tags. See `<start>` above for the description of the `<constant_date_time>` tag.
- **flag** – If this tag appears, the search looks for calls where a flag with the name given in the `name` attribute was triggered.

XML Decision Example #1

An application named “Example1” would like to play “Welcome back” for callers who have previously called this application. The users are identified by their ANI (this application uses the user management database only for its history tracking). A decision element named “CalledBefore” would be needed which had two rules, one for those who the application recognizes and one for the rest (this is being done rather than using the default exit state for demonstration purposes). In English, the rules are:

Rule	***Expression***	***Exit State***
1	The caller has called from this ANI before	say_welcome_back
2	The caller has not called from this ANI before	say_welcome

The Universal Edition decision element XML file would be named “CalledBefore” and be saved in `AUDIUM_HOME/applications/Example1/data/misc`

The XML content will be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE knowledge_base SYSTEM "../../../../dtds/DecisionKnowledgeBase.dtd">
<knowledge_base>
   <rule name="CalledFromAni">
      <exit_state name="say_welcome_back" conjugate="and">
         <boolean check_existence="no" value="true">
             <user_info>
                <called_from_ani/>
             </user_info>
         </boolean>
      </exit_state>
     <exit_state name="say_welcome" conjugate="and">
         <boolean check_existence="no" value="false">
             <user_info>
                <called_from_ani/>
             </user_info>
         </boolean>
      </exit_state>
  </rule>
</knowledge_base>
```

XML Decision Example #2

An application named “Example2” randomly chooses two letters of the alphabet and gives a prize to the caller whose name begins with either letter. The letters are chosen by an action element named “GetRandomLetter” and stored in element data named “letter1” and “letter2”.

A decision element named “IsCallerAWinner” would be needed which has three exit states:

- For a user whose name begins with either letter.
- For users whose name does not begin with the letters.
- For users whose name is not in the records (this could be an error or could prompt the application to ask the user to register on the website).

Even if the application assumes that all users will have their names on file, it is prudent to add this third exit state just to make sure. In this example, the default exit state will be set to when the users do not match.

In English, the rules are:

Rule	***Expression***	***Exit State***
1	The caller’s name begins with the value stored in the element “GetRandomLetter” with the variable name “letter1” or begins with the value stored in the element “GetRandomLetter” with the variable name “letter2”	is a winner
2	The caller’s name does not begin with the value stored in the element “GetRandomLetter” with the variable name “letter1” and does not begin with the value stored in the element “GetRandomLetter” with the variable name “letter2”	not a winner
3	The caller’s name does not exist	no name

The Universal Edition decision element XML file would be named “IsCallerAWinner” and be saved in `AUDIUM_HOME/applications/Example2/data/misc`.

The XML file content will be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE knowledge_base SYSTEM "../../../../dtds/DecisionKnowledgeBase.dtd">
<knowledge_base>
   <rule name="NameStartsWith" default_exit_state=”not a winner”>
      <exit_state name="no name" conjugate="and">
         <boolean value="false" check_existence="yes">
            <user_info>
               <demographic type="name"/>
            </user_info>
         </boolean>
      </exit_state>
      <exit_state name="is a winner" conjugate="or">
         <string operator="starts_with">
            <user_info>
               <demographic type="name"/>
            </user_info>
            <data>
               <element name="GetRandomLetter” variable="letter1"/>
            </data>
         </string>
         <string operator="starts_with">
            <user_info>
               <demographic type="name"/>
            </user_info>
            <data>
               <element name="GetRandomLetter” variable="letter2"/>
            </data>
         </string>
      </exit_state>
  </rule>
</knowledge_base>
```

Notes:

- The “no name” exit state is listed first. This is because before we try to analyze the user’s name, we have to first know that it exists. So we check if the name *does not* exist first and if it fails it means the name exists and we can go on.
- The second exit state must check if the name begins with the first *or* second letter but the last exit state must check if the name does not begin with the first *and* second letter.

XML Decision Example #3

An application named “Example3” is designed to trigger a flag named “account menu” when a caller chooses to manage their account. As of June 15, 2004, the menu options were changed for the account menu. We want to tell people the options have changed, but only if we know they’ve visited that part of the application before June 15. If not, there is no reason to say anything because the caller is experiencing this for the first time. A decision element is needed that distinguishes between those to play the changed audio to from those who should encounter the menu normally. A tricky part of the rule is that it must deal with the day, month, and the year,

making sure that callers from previous years and future years are handled correctly as well. Since the current state of the XML format does not allow date comparisons, a way must be determined to make this restriction. The solution is to use multiple rules which progressively get more restrictive in a sort of process-of-elimination manner. Since all conditions are to be handled, the rule must include those who do not hear the changed message using the same scheme (there is no need to use the default exit state). In English, the rules are:

Rule	***Expression***	***Exit State***
1	The year the last time the caller triggered the flag “account menu” is less than 2004	play changed
2	The year the last time the caller triggered the flag “account menu” is greater than 2004	normal
Note that at this time, if the above two rules were not triggered, the caller triggered the flag in the year 2004.		
3	The month of the year the last time the caller triggered the flag “account menu” is less than 6	play changed
4	The month of the year the last time the caller triggered the flag “account menu” is greater than 6	normal
Note that at this time, if the above two rules were not triggered, the caller triggered the flag in June 2002.		
5	The day of the month the last time the caller triggered the flag “account menu” is less than or equal to 15	play changed
6	The day of the month the last time the caller triggered the flag “account menu” is greater than 15	normal

The Universal Edition decision element XML file would be named “IsCallerAWinner” and be saved in `AUDIUM_HOME/applications/Example3/data/misc`.

The content of the XML file will be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE knowledge_base SYSTEM "../../../../dtds/DecisionKnowledgeBase.dtd">
<knowledge_base>
   <rule name="NewMessageTest">
      <exit_state name="play changed" conjugate="and">
         <number operator="less">
             <historical_data type="last_date_time" field="year">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="2004"/>
         </number>
      </exit_state>
```

```
      <exit_state name="normal" conjugate="and">
         <number operator="greater ">
             <historical_data type="last_date_time" field="year">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="2002"/>
         </number>
      </exit_state>
      <exit_state name="play changed" conjugate="and">
         <number operator="less">
             <historical_data type="last_date_time" field="month">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="6"/>
         </number>
      </exit_state>
      <exit_state name="normal" conjugate="and">
         <number operator="greater">
             <historical_data type="last_date_time" field="month">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="6"/>
         </number>
      </exit_state>
      <exit_state name="play changed" conjugate="and">
         <number operator="less_equal">
             <historical_data type="last_date_time" field="day_of_month">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="15"/>
         </number>
      </exit_state>
      <exit_state name="normal" conjugate="and">
         <number operator="greater">
             <historical_data type="last_date_time" field="month">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="6"/>
         </number>
      </exit_state>
  </rule>
</knowledge_base>
```

VoiceXML Insert Elements

VoiceXML insert elements are different from other elements in that they are built almost entirely outside Call Services using VoiceXML directly. One can think of an insert element as a way to insert custom VoiceXML content into a Universal Edition voice application without sacrificing the ability to interface with other elements in the call flow. While there are guidelines to follow to make these elements work, there are few restrictions on the VoiceXML content itself.

There are two common reasons an insert element is used, the first being the ability to leverage VoiceXML content that has already been created and integrate it into a Universal Edition application without having to do much recoding. The second reason is in situations where the requirement is to write a VoiceXML-producing element that is a one-off without having to go through the effort of writing a configurable voice element in Java and the VFCs. This is very similar to the reasons for writing standard action and decision elements instead of producing a configurable element. Writing VoiceXML is simpler than creating a voice element from scratch since that requires knowledge of both VoiceXML as well as the Universal Edition Java API.

One of the disadvantages of using insert elements is the fact that since the VoiceXML must be written to comply with a specific voice browser, the browser-agnostic capability of the voice application is lost. If the application is moved to another voice browser, all Universal Edition elements would automatically work, but the insert elements would have to be retested and tweaked to conform to the new browser's requirements. Another disadvantage is the insert element's lack of a configuration. If the desire is a reusable, configurable element, it is preferable to construct a voice element.

VoiceXML insert elements are accessed via a VoiceXML `<subdialog>`. The VoiceXML specification provides this tag as a way of allowing simple reusable functionality. It acts very much like a "function" in programming where inputs are sent to a function that performs some actions and returns the results. The subdialog definition itself can be located anywhere accessible with a URI. In this way, the Universal Edition application sees an insert element as simply another function to access.

The inputs and outputs are the means by which the insert element interfaces with the rest of the system. Most of the important data available to Universal Edition elements are sent as input to each Insert element. Once the insert element is complete, the return information contains any element or session data to create, log entries, the exit state of the insert element, and other data to act upon.

Restrictions

The following restrictions apply to a VoiceXML insert element. An insert element conforming to these restrictions will be assured full integration with the Universal Edition application. These restrictions will be clarified later.

- The insert element cannot define its own root document, a root document generated by Call Services must be used.
- The variables to return to Call Services, including the exit state, must conform to a strict naming convention.
- When using the <return> tag, Universal Edition-specified arguments must be returned along with the custom variables.

Inputs

As with any element in the application, an insert element would need to be able to access information about the call such as element and session data, call data (such as the ANI), and even information found in the user management database if the application is configured to use one. Normally, this information is available in the Java or XML API. Since an insert element is written in VoiceXML, this information must be made available for the insert element to use from within the VoiceXML.

Universal Edition achieves this by creating VoiceXML variables in the root document containing all the desired information. The variable names conform to a naming convention so the Insert element developer can refer to them appropriately. This is one reason why Universal Edition requires the use of the Call Services-generated root document.

In order to cut down on the number of variables appearing in the root document, the application designer is given the option of choosing which input groups are passed to the insert element. Additionally, the designer can individually choose which element and session data to pass. By minimizing the inputs to only the data required by the insert element, the overhead involved in using an Insert element is minimized.

Each input type is listed below:

- **Telephony**. This information deals with telephony data. The inputs start with "audium_telephony_".
  - **audium_telephony_ani**. The phone number of the caller or "NA" if not sent.
  - **audium_telephony_dnis**. The DNIS or "NA" if not sent.
  - **audium_telephony_iidigits**. The IIDIGITS or "NA" if not sent.
  - **audium_telephony_uui**. The UUI or "NA" if not sent.
  - **audium_telephony_area_code**. The area code of the caller's phone number. Will not appear if the ANI is "NA".
  - **audium_telephony_exchange**. The exchange. Will not appear if the ANI is "NA".

- **Call**. This information deals with the call. The inputs start with "audium_call_".
  - **audium_call_session_id**. The session ID.
  - **audium_call_source**. The name of the application which transferred to this one. Will not appear if this application is the first application in the call.
  - **audium_call_start**. The start time of the call in the format "DAY MNAME MONTH HH:MM:SS ZONE YEAR" where DAY is the abbreviated day of the week (e.g. "Wed"), MNAME is the abbreviated name of the month (e.g. "Jun"), HH is the hour (in military time), MM is the minute, SS is the seconds, ZONE is the time zone (e.g. "EDT"), and YEAR is the four-digit year.
  - **audium_call_application**. The name of the current application.
- **History**. This information deals with the history of elements visited so far in the call. The inputs start with "audium_history_".
  - **audium_history**. This entire content of the element history (including exit states) is contained in this variable. The format is [ELEMENT]:[EXITSTATE]|..|[ELEMENT]:[EXITSTATE] where ELEMENT is the name of the element and EXITSTATE is the name of the exit state of this element. The order of the element/exit state pairs is consistent with the order in which they were visited. This will not appear if this insert element is the first element in the call.
- **Data**. This is the element and session data created so far in the call.
  - **audium_[ELEMENT]_[VARNAME]**. This is an element variable where ELEMENT is the name of the element and VARNAME is the name of the variable. Note that both the element and variable names will have all spaces replaced with underscores. There may be no instances of this input if no element variables exist when this insert element is visited. For example, the variable "audium_MyElement_the_value" is element data named "the value" from the element "MyElement".
  - **audium_session_[VARNAME]**. This is a session variable whose name is VARNAME. Note that the variable name will have all spaces replaced with underscores. The value is expressed as a string even if the type is not a string (the `toString()` method of the Java class is called). There may be no instances of this input if no session variables exist when this insert element is visited.
- **User Data**. This element information associated with the caller. It will only appear if the application has associated the call with a UID and a user management database has been set up for this application. The data will appear in the input exactly as in the database. The inputs start with "user_".
  - **user_uid**. This is the UID of the user.
  - **user_account_number**. The account number of the user.
  - **user_account_pin**. The PIN of the user.
  - **user_demographics_name**. The name of the user.

- **user_demographics_birthday**. The birthday of the user.
- **user_demographics_zip_code**. The zip code of the user.
- **user_demographics_gender**. The gender of the user.
- **user_demographics_social_security**. The social security number of the user.
- **user_demographics_country**. The country of the user.
- **user_demographics_language**. The language of the user.
- **user_demographics_custom1**. The value of the first custom column.
- **user_demographics_custom2**. The value of the second custom column.
- **user_demographics_custom3**. The value of the third custom column.
- **user_demographics_custom4**. The value of the fourth custom column.
- **user_account_external_uid**. The external UID of the user.
- **user_account_created**. The date the account was created in the format. The value is in the format "DAY MNAME MONTH HH:MM:SS ZONE YEAR".
- **user_account_modified**. The date the last time the account was modified. The value is in the format "DAY MNAME MONTH HH:MM:SS ZONE YEAR".

- **User By ANI**. This provides historical information about the phone number of the caller with regards to this application. It will only appear if a user management database has been set up for this application. The inputs start with "user_by_ani_".
  - **user_by_ani_num_calls**. The number of calls made by this phone number.
  - **user_by_ani_last_call**. The last call made by the phone number. Will not appear if there were no calls made by this phone number in the past.

Outputs

Just like any element, VoiceXML insert elements can create element and session data, set the UID of the user to associate with the call, send custom logging events, and can return one of a set of exit states. Like voice elements, insert elements can have internal logging of caller activity and have global hotlinks and hotevents activated while the caller is visiting the Insert element. All of these actions involve variable data set within the Insert element and returned to Call Services. These are crucial in order to properly integrate with the rest of the elements in the application. Each of the return arguments is listed below:

- **audium_exit_state**. The exit state of this VoiceXML insert element. The value of this variable must be exactly as chosen in the Builder for Call Studio when defining the insert element.

- **element_log_[VARNAME]** / **element_nolog_[VARNAME]** These create new element data for this VoiceXML insert element whose name is VARNAME and which either sends a logging event to log the element data value or not, respectively. The data type will be assumed to be a string. The variable name cannot include spaces.
- **session_[VARNAME]**. This creates a new session variable whose name is VARNAME. The data type is assumed to be a string. The variable name cannot include spaces. If the variable name already exists, the old value will be replaced with this one. If the old data type was not a string, the new data type will be a string.
- **custom_[NAME]**. This sends a custom logging event whose contents is the action named NAME and the value of the variable being the description.
- **set_uid**. This associates the UID passed to the call.
- **audium_hotlink, audium_hotevent, audium_error, audium_action.** These four Universal Edition variables are created in the root document and must be passed along in the return `namelist`. The content of each deals with the occurrence of any global hotlinks, hotevents, errors, or actions (e.g., a hang-up) while in this insert element. Since the subdialog has its own context and root document, this data has to be explicitly passed for any of these events to be recognized by Call Services. The developer should not alter the contents of these variables.
- **audium_vxmlLog**. This variable contains the raw content for an interaction logging event. Adding to the interaction log is not required - the `audium_vxmlLog` variable can be passed empty. In order for Call Services to parse the interaction data correctly, a special format is required for the content of the `audium_vxmlLog` variable. This format is defined below:

The format for interaction logging is:

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

where:

- ACTION is the name of the action. The following lists the possible action names and the corresponding contents of VALUE:
  - **audio_group.** This is used to indicate that the caller heard an audio group play. 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 key presses. VALUE should be contents of the `inputmode` VoiceXML shadow variable.
  - **utterance.** This is used to report the utterance as recorded by the speech recognition engine. VALUE should be the contents of the utterance VoiceXML shadow variable.
  - **interpretation.** This is used to report the interpretation as recorded by the speech recognition engine. VALUE should be the contents of the interpretation VoiceXML shadow variable.

  - **confidence.** This is used to report the confidence as recorded by the speech recognition engine. VALUE should be the contents of the confidence VoiceXML shadow variable.
  - **nomatch.** This is used to indicate the caller entered the wrong information, incurring a nomatch event. VALUE should be the count of the nomatch event.
  - **noinput.** This is used to indicate the caller entered nothing, incurring a noinput event. VALUE should be the count of the noinput event.

- ELAPSED is the number of milliseconds since the VoiceXML page was entered. The root document provides a JavaScript function named `application.getElapsedTime(START_TIME)` which returns the number of milliseconds elapsed since the time specified in `START_TIME`.

The root document created by Call Services for use in all VoiceXML insert elements contains a VoiceXML variable named `audium_element_start_time_millisecs` that must be initialized with the time in order for the elapsed time intervals to be calculated correctly. This variable need only be initialized once in the *first VoiceXML page* of the insert element. All subsequent pages in the VoiceXML insert element must not initialize the variable because Call Services requires the elapsed time from the start of the element, not the page. So, in VoiceXML, the line to appear must look like:

```
<assign name="audium_element_start_time_millisecs" expr="new Date().getTime()" />
```

For best results, this should appear as early as possible in the first page, preferably in a `<block>` in the first `<form>` of the page, certainly before any additional logging is done.

In VoiceXML, setting the value of an existing variable requires the `<assign>` tag. Since the expression contains a JavaScript function, the `expr` attribute must be used. Additionally, in order to avoid overwriting previous log information, the expression must append the new data to the existing content of the variable. For example, to add to the interaction log the fact that the *xyz* audio group was played, the VoiceXML line would look like

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

In another example, the utterance of a field named *xyz* is to be appended to the log. The VoiceXML would look like

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

See Chapter 5: Call Services Logging for more detail about Universal Edition logging.

Root Document

The subdialog context written by the developer must refer to a Universal Edition-generated root document. This is essential for proper integration of the VoiceXML insert element with Call Services. The root document call must look like:

"`/CallServices/Server?audium_vxml_root=true&calling_into=APP&namelist=element_log_value|RTRN1|RTRN2|…`"

Where APP is the application name and RTRNX represent the names of all the element data, session data, and custom log entries (delimited by '|' characters) the insert element returns, using the same naming convention described in the outputs section above.

The purpose for this requirement is related to how events are handled within the root document. The Universal Edition-generated root document catches events such as the activation of a global hotlink or a hangup, which then requires the call flow to leave the insert element. The insert element, however, may have created element and session data or added custom content to the log. This information is stored in VoiceXML variables that would be deleted once the subdialog context is exited. So the root document needs to be told which VoiceXML variables to send along to Call Services when one of these events is triggered so that it can store them accordingly. In order to avoid problems that might occur if a global hotlink or hotevent was activated right after the insert element began the variables to be returned should be declared as near the start of the VoiceXML insert element as possible, even if they are not assigned initial values.

Notes:

- The ability to use a standard ampersand in the root document URL instead of escaping it (as "&") is voice browser dependent. Most browsers will accept the escaped version so try that first.
- If the insert element does not need to send back any data in the `namelist` parameter, only the `element_log_value` variable need be included (the parameter should look like this: "...`namelist=element_log_value`").

Example

In the example below, a block is used to log the playing of the `initial_prompt` audio group. After this action, some inputs passed to it from Call Services are played. Once this is done, it creates two element variables named `var1` and `var2` and a session variable named `sessvar`. After this, it goes through a field that catches a number and when done saves the utterance to the activity log and returns the exit state *less* if the number is less than 5 and *greater_equal* otherwise. The `<return>` tag returns the exit state, log variable, the four variables from the root document (error, hotlink, hotevent, and action), the two element data variables, the session data variable and a custom log entry (the number captured). Also note that these last four variables are also passed to the root document call in the `<vxml>` tag so that events triggered within the insert element will correctly pass the data if it was captured by then. Note that the VoiceXML listed below may not function on all browsers without modification.

```
<?xml version="1.0"?>
<vxml version="1.0"
application="/CallServices/Server?audium_vxmlroot=true&amp;calling_into=MYAPP&
amp;namelist=element_log_var1|element_nolog_var2|session_sessvar|custom_custlo
g">
   <form id="testform">
      <block>This is the initial prompt
         <assign name="audium_element_start_time_millisecs" expr="new
Date().getTime()"/>
         <assign name="audium_vxmlLog"
expr="'|||audio_group$$$initial_prompt^^^'
+application.getElapsedTime(audium_element_start_time_millisecs)"/>
      </block>
      <block>In the VoiceXML element.
      The ani is <value expr="audium_telephony_ani"/>.
      The element history is <value expr="audium_history"/>.
      User by ani num calls is <value expr="user_by_ani_num_calls"/>.
      Element data foo from element first <value expr="audium_first_foo"/>.
      Session variable foo2 <value expr="audium_session_foo2"/>.
      </block>
      <var name="element_log_var1" expr="'log me'"/>
      <var name="element_nolog_var2" expr="'do not log me'"/>
      <var name="session_sessvar" expr="'session_data_value'"/>
      <field name="custom_custlog" type="number">
         <property name="inputmodes" value="voice" />
         <prompt>Say a number.</prompt>
         <filled>
            <assign name="audium_vxmlLog" expr="audium_vxmlLog +
'|||utterance$$$' + custom_custlog.$utterance + '^^^'
+application.getElapsedTime(audium_element_start_time_millisecs)"/>
            <if cond=" custom_custlog < 5">
               <assign name="audium_exit_state" expr="'less'"/>
               <else/>
               <assign name="audium_exit_state" expr="'greater_equal'"/>
            </if>
            <return namelist="audium_exit_state audium_vxmlLog audium_error
audium_hotlink audium_hotevent audium_action element_log_var1
element_nolog_var2 session_sessvar custom_custlog" />
         </filled>
      </field>
   </form>
</vxml>
```

Chapter 3: Administration

Administration is an essential feature of any enterprise system. Once started, a system must remain operational for long periods of time with no downtime so it must expose ways for an administrator to manage it at runtime. This applies to both changes and updates to the application as well as providing information concerning its health. The more flexible and informative a system, the better an administrator will be able to ensure it runs efficiently and detect any issues with the system quickly.

Call Services has been designed to afford maximum flexibility for administrators to control how it runs and to monitor vital statistics of its health. Administrators can add, remove and change applications deployed, get information on the system and the applications, and even change the behavior of system or components, without requiring a restart of Call Services.

This chapter details the administration functions and statistics exposed by Call Services and the mechanisms by which these functions can be accessed and executed.

Introduction to Call Services Administration

Call Services exposes three methods for an administrator to control it and obtain information. Each method is accessed differently and exposes different levels of functionality or information. The first method, and the most flexible, is the JMX-compatible management interface. The second method is through the use of administration scripts. The third is via the system information web page.

JMX Management Interface

Java Management Extensions (JMX) is a Java technology specifically designed for managing Java applications. It is part of the standard Java Virtual Machine and defines a standard interface for clients and servers. An application that wishes to be managed by JMX will register MBeans to the JMX context. An MBean can be used to expose information about the system that an administrator can fetch (for example the total simultaneous calls on the system). An MBean can also be used to expose a function that an administrator can execute (for example to suspend an application). A client application communicates with the server application via the JMX interface to allow administrators access to the information and function that is exposed.

Call Services, being a server application, exposes many informational MBeans for information regarding itself as well as the applications deployed on it. It also exposes administrative MBeans for controlling important administrator functions. It does this in a fully JMX-complaint manner so that any JMX-compatible client will be able to interface with Call Services to gain access to the information and functions. One such client is JConsole, which is a client bundled with JDKs provided by Sun Microsystems and others. Some JVMs and application servers provided by other companies may utilize alternative JMX-compatible clients that should work as well.

It is also possible for a developer to create their own custom MBeans for exposing functions or information that will then be viewed by a JMX-compatible client alongside the MBeans exposed by Call Services. See the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more on creating custom MBeans.

Most JVMs do not start up with JMX enabled by default and require a parameter to be passed to the JVM to turn it on (for example, Sun Microsystems JVMs require the parameter `-Dcom.sun.management.jmxremote`). Any change to the JVM parameters must be implemented prior to the Java application server is started.

Once Call Services is started, a JMX client can then be launched and configured to point to the machine on which Call Services runs, whether it be on the same machine or a remote one. Once connected, the client provides a graphical interface for displaying the information and functions. The client will be able to display information about the JVM itself and typically the Java application server will publish its own set of MBeans. Call Services information will be displayed where the MBeans are listed in its own "domain". The domain is typically rendered in a tree structure and will list global information and functions (i.e. information having to do with Call Services itself) as well as information on the deployed voice applications. Detailed explanations of the individual MBeans are provided in the following sections.

To address security, JMX client consoles will request a user name and password if they attempt to connect to a remote server (no user name or password is required to connect to a local Call Services because the client already has access to the local system). These credentials can be defined at installation time. Security is most important for the administration functions as they do affect the live system and if misused could cause instability. Note that many JMX clients do not provide role-based authentication, so once a user has successfully logged in, the user has access to all information and the ability to run all administrator functions. Therefore it is recommended to provide this user name and password only to designated administrators.

Of the available administration interfaces, the JMX interface for Call Services provides the greatest functionality and flexibility. It does, however, require the JVM to have JMX active and a JMX-compatible client. It also has a higher risk and overhead due to this flexibility.

Administration Scripts

Most of the administration functions and some of the information about Call Services are provided via command-line scripts that can be executed by an administrator manually or an automated system directly. The administrator scripts do not use the JMX interface described in the previous section and are functional by default without requiring any configuration on the administrator's part. The included scripts act as the client. The scripts are provided in two forms: batch scripts for Microsoft Windows (ending in `.bat`) and shell scripts for Unix (ending in `.sh`).

Scripts are provided to execute global functions (on Call Services itself) or functions for individual applications. The scripts used for global administrator functions are found in the

`admin` directory of Call Services. The scripts used for individual application administration are located in the `admin` directory of each application.

The provided scripts are primarily used to expose Call Services functions to administrators such as loading a new application, updating an existing application, suspending Call Services, etc. Some scripts provide information, such as the number of active simultaneous calls on the server. This chapter describes in detail all available scripts and their functionality.

Security is an important concern when it comes to administration functions that are access from the command-line. There are several precautions Universal Edition sets up to allow only the appropriate people access to these scripts. First, by providing scripts or batch files (as opposed to through a graphical or web interface), the administrator must be logged into the machine in order to access them. Therefore, accessing these programs is as secure as the remote login process (such as SSH) and the permissions given to these scripts or the entire `admin` folder. Secondly, Call Services will only accept commands from the local machine, so even scripts stored on one machine cannot issue commands to an instance of Call Services running on another machine. These two precautions ensure that only authorized administrators can access these functions.

Since the global administration scripts are stored in a different location from application scripts, each directory can be assigned different permissions. That way an administrator can be given access to the global administration scripts while still allowing the application scripts to be accessed by voice application developers.

Finally, every administration script can be configured to ask for confirmation before the action is taken, to prevent the accidental execution of the script. By default the confirmations are on. They can be turned off by passing the command-line argument “noconfirm” to the script. This can be useful if the administration scripts are executed by automated systems like cron jobs.

While not as flexible as the JMX interface, administration scripts provide easy access to Call Services functions for both administrators and automated systems out of the box. The risk potential is similar to that of the JXM interface, although there is less overhead because JMX is not enabled.

System Information Page

The system information page provides basic information about Call Services including the license information, the deployed gateway adapters and applications. Status of the application server on which Call Services is running, and some miscellaneous system and Java information such as the version and memory usage. It does not provide the ability to execute any functions, it is meant to be a quick way to check relevant information. It is also the easiest of the three methods to obtain information because all that is needed is a web browser. The system information page can be seen by pointing a web browser to the URL:

```
http://[HOST][:PORT]/CallServices/Info
```

where:

- HOST is the host name of the machine on which Call Services is installed.
- PORT is the port the application server is configured to listen on. If the application server is configured to use port 80, there is no need to include the port in the URL.

The system information page can only be reached after proper authentication using the administrator user name and password defined during installation time. This is the same user name and password used when configuring Call Services licenses.

The system information page is the easiest and safest way of obtaining administrative information, though it is also the least flexible.

Administration Information

Using the tools listed above, an administrator can obtain a significant amount of information regarding Call Services and the applications that are deployed on it. This information aids the administrator in determining the health of the system, detecting signs of issues that should be caught early, and debugging issues as they occur.

Much of the information made available by Call Services can be found only via the JMX interface as that is the strength of JMX. Some of the more important information is available via scripts and some of the static information is available through the system information page.

Application and System Status

Call Services provides functions for reporting the status of a specific voice application or all voice applications running on the system. They are provided as functions to allow the administrator to query Call Services to get the latest information immediately.

The application status function reports the following information:

- Whether the application is running, suspended, or has been suspended before being slated for removal.
- How many active sessions are currently visiting the application. Active sessions are defined as the number of callers that are interacting with the application at the time the status script is called.
- How many sessions are waiting to end. When an active caller ends their application visit, Call Services delays the closing of the corresponding session to allow completeion of session accessed by final logger and end of call class actions. A session waiting to end does not take up a license port. The amount of time a session remains open after a call ends is a Call Services configuration option (see Chapter 6: Call Services Configuration for more).

- How many open sessions are experiencing the most recent past version of the application. Open sessions are the sum of active callers visiting the application and those sessions that are in the process of ending. The reason open sessions are listed here is because both active and ending sessions do need access to session information and an administrator would need to know when it is safe to disable any systems that the old application configuration depends on. This information is helpful for an administrator when performing an application update or suspension in determining when the executed function is complete. See the following sections for more on updating and suspending applications.

- How many callers are on hold waiting to get into the application. A call that is received when the system has used up all the allowed sessions defined in the license will hear a message asking them to stay on the line. This call then checks if a license session has become available and then lets the call into the application.

The Call Services status function provides an easy to read report with the following information:

- If Call Services itself has been suspended, this fact is listed first. See the following sections for more on suspending Call Services.
- The total number of concurrent active callers visiting applications on this instance of Call Services, how many concurrent sessions the license allows, the number of available ports (the license sessions minus the active callers), and the number of callers on hold (which would only appear if the number of current callers exceeds the number of license sessions).
- How many active callers, sessions ending, and callers on hold for each application currently deployed on the system. This data is the same as would be displayed by the application-specific status function. Note that no on hold column will appear unless there are callers on hold.
- Whether each application is running or suspended.

JMX Interface

To get an application's status using the JMX interface. Using a JMX client connected to the server, navigate to the `VoiceApplication/APPNAME/Command` MBean, where `APPNAME` is the name of the application to update. The operations tab of this MBean will list a function named "status". Pressing this button will display a dialog box with the application status. To get the status of all applications using the JMX interface, navigate to the `Global/Command` MBean and click on the function named "status" in the operations tab. Pressing this button will display a dialog box with the status of each application deployed on Call Services in a table.

Administration Scripts

The script for obtaining an application status is found in the `admin` folder of the application to be updated. Windows users should use the script named `status.bat` and Unix users should use the script named `status.sh`. The script for obtaining the status of all applications is found in the

admin folder of Call Services. Windows users should use the script named status.bat and Unix users should use the script named status.sh. The scripts do not take any parameters.

Call Services Information

Call Services reports information about itself that is static so the administrator knows exactly what is installed. The following information is reported:

- The exact name and version of Call Services.
- The installation key, expiration date, number of ports, and the supported gateway adapters listed in the Call Services license. Note that the gateway adapter list is not a comprehensive list of the adapters installed on Call Services but rather a list of the gateway adapters the license allows the system to use.
- A detail of the version numbers of all components included with Call Services. This information can be helpful for tracking changes made to individual components of the software installed at different times and this detailed information will typically be requested by Universal Edition support representatives when a question is raised about the software. The components whose versions are displayed are:
    - The Call Services web application archive (WAR) and the components residing within Audium Home. Note that this version is different from the Call Services product version as that is a version for the whole system and this one is only for the WAR file.
    - The core Call Services elements, Say It Smart plugins, and loggers (both application and global) included with the software.
    - The Gateway Adapters installed on the system.

JMX Interface

To obtain Call Services information using the JMX interface, navigate to the Info MBean. The attributes tab displays all the information listed above. To see all the gateway adapters supported in the license, one must open the value for the "LicensedGWAdapters" attribute (in JConsole this is done by double-clicking on the value). The same procedure applies for obtaining the component versions by opening the value for the "ComponentVersions" attribute.

Administration Scripts

The only Call Services information available via script is the versions of the components installed on Call Services, though the name and version of Call Services is displayed when it initializes and the license ports is always displayed using the global status script.

The script is found in the admin folder of Call Services. Windows users should use the script named getVersions.bat and Unix users should use the script named getVersions.sh. In order to report on the version of the Call Services web application archive (WAR), the script should be passed as an argument the full path of the WAR location (e.g. "C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\").

System Information Page

The same information is displayed in the system information page at the top of the table. It will also provide a list of the applications deployed on Call Services as well as information on the application server, operating system, and Java memory usage.

Server Status Checks

Many load balancers can be configured to access a URL that is used to determine if a server is running. This request is usually made periodically. The load balancer makes a request to the URL and if a response comes back within an acceptable time, it considers the server available to handle connections. To gauge the health of Call Services, the URL to access must contain the argument "probe=true" in the request. The URL may look like either of the following:

```
http://[DOMAIN][:PORT]/CallServices/Server?probe=true
http://[DOMAIN][:PORT]/CallServices/Server?application=[APPLICATION]&probe=true
```

The "probe=true" argument is required to ensure that probe calls do not affect the session (i.e., license) usage statistics.

If Call Services is accessible and is not suspended, it will respond with:

```
The Cisco Unified Call Services, Universal Edition is up and running
```

However, if it is suspended (via the `suspendServer` administrative script), it will respond with:

```
The Cisco Unified Call Services, Universal Edition is running, however it has
been suspended.
```

Configuration Updates

When an administrator monitors a Call Services installation, they want to be aware of any warning signs that the system is overloaded. In these scenarios, it is advantageous if the administrator is able to tweak a few settings to better handle the given load without worrying about updating or suspending applications or shutting down the Java application server. These tweaks may enable a system to better handle spikes in call activity with no adverse effects. To this end, Call Services exposes some of its configuration options and allows an administrator to change them at runtime. It also allows the administrator to change some application settings values for deployed application.

It is important to note that the administrator must be very careful when altering these configuration options at runtime as improperly chosen values could make the system unstable and achieve the opposite effect than desired.

The ability to change Call Services configuration options and application settings is available only through the JMX interface. The configuration options are exposed as attributes of an MBean, one for the Call Services configuration options and one for each application's settings. Those attributes that allow their values to be changed will have editable values. When a new

value is given, it takes affect immediately with no confirmation so it is important to ensure that the value entered is correct. There is some simple validation that takes place by Call Services and if the value entered is inappropriate (such as entering -1 where a positive integer is required), the change will not take place and the original value will remain unchanged. The administrator will know that their entry was accepted if the value does not revert back.

It is very important to note that any changes made to these attributes are ***not persisted***. The changes affect Call Services in memory and do not affect the XML files that hold these values. As a result, should the Java application server or the Call Services web application be restarted or for application-specific attributes the application is updated, the attributes will revert back to the values specified in their respective XML files.

Call Services Configuration Options

To view the Call Services configuration options using the JMX interface, navigate to the `Global/Configuration` MBean. There are five attributes listed. The first, named "LoggerEventQueueSize", will show the current size of the queue that holds logger events waiting to be sent to loggers and is not editable. The next three are related in that they control aspects of the logger thread pool. The final configuration option deals with a period of time Call Services waits after a caller ends their call before the call session is invalidated. All of these options affect the performance of the system and are defined fully in Chapter 6: Call Services Configuration. Use the following table to reference the JMX attribute name with the `global_config.xml` tag name.

JMX Attribute Name	Tag Name
LoggerMaximumThreadPoolSize	`<maximum_thread_pool_size>` in the `<logger>` tag.
LoggerMinimumThreadPoolSize	`<minimum_thread_pool_size>` in the `<logger>` tag.
LoggerThreadKeepAliveTime	`<keep_alive_time>` in the `<logger>` tag.
SessionInvalidationDelay	`<session_invalidation_delay>`

Table 3-1

Tuning Logger Options

The most important indication of whether Call Services is encountering issues with loggers is the "LoggerEventQueueSize" attribute. A brief explanation of how Call Services handles loggers is warranted (for more details refer to Chapter 5: Call Services Logging). In order to prevent logging from holding up calls, all logging is done in separate threads. The threads are managed within a thread pool, which has a maximum and minimum value. When Call Services starts up, the thread pool allocates the minimum number of threads. As calls begin to be handled, they generate logger events, which are put into a queue of events. The activation of a logger event also prompts Call Services to request a thread from the pool and in that thread, have the appropriate logger handle the top most event in the queue. The length of time this thread handles the event depends on the logger, but the event is typically handled in a very short period of time, measured in milliseconds. However as call volume on the system increases, more threads are used simultaneously to handle the increase in logger events added to the queue. As more threads

are needed, the thread pool grows until it reaches the maximum number of threads allowed. At that point the queue would grow until threads become available. Threads that complete their work and cannot find new logger events to handle because the queue is empty will be garbage collected after a certain amount of time being idle (this is governed by the LoggerThreadKeepAliveTime option).

Under typical operation, the logger event queue size should not be a large number (one might see it set to 0 to 10 most of the time). There could be spikes where the queue grows quickly but with plenty of available threads to handle the events, the queue size should shrink rapidly. The administrator should take note if the queue size shows a high number, though should be very wary if this number seems to grow over time (minutes, not seconds). A growing queue size is an indication that either the load on the system is too high for the thread pool to handle (which is more likely the smaller the maximum thread pool size is set) or for some reason loggers are taking longer to do their logging. In the latter case this could be due to a slow database connection, overloaded disk IO or other reasons. Regardless of the cause, a growing queue is a warning sign that if the call volume is not reduced, the Java application server is at risk of encountering memory issues and, in the worst case, running out of memory.

It is for this reason that choosing an appropriate maximum thread pool size is important. While the temptation to give the maximum number of threads a very high number this can also cause problems on the system as severe as memory issues. Using too many threads could cause what is called “thread starvation” where the system does not have enough threads to handle standard background processes and could exhibit unpredictable inconsistent behavior and could also cause the Java application server to crash.

The JMX interface supports the ability to change the maximum and minimum thread pool size at runtime. The administrator should only do this if they believed the change could avert an issue listed above. For example, if the system is encountering a temporary spike in activity and the administrator sees the LoggerEventQueueSize attribute report a growing number, then they can increase the maximum thread pool size to potentially allow for a more rapid handling of the queued events. Once the queue shrinks to a manageable number the maximum thread pool size can then be changed back to its original value.

The maximum number of threads set by default in Call Services is sufficient to handle a very heavy load so the administrator is urged to use caution when changing these values.

Session Invalidation Delay Option

The session invalidation delay option is also an important value that an administrator could be tuned. A brief explanation of what this option does is warranted (for more details refer to Chapter 6: Call Services Configuration). When a caller ends the call by either hanging up, going to another application, or the application hangs up on the caller, Call Services must perform some final clean up of the call session. This is primarily for processing logging events that occurred when the call ended. Additionally, application developers can configure their applications to execute code at the end of a call to perform their own clean up operations. In

sophisticated applications this could involve closing database connections or generating call detail records. These end of call operations can take a non-trivial amount of time and may require access to information about the call session such as element or session data. As a result, Call Services waits for a preset period of time after a call ends before it invalidates the session, allowing all activities requiring additional time to complete. This period of time is governed by the SessionInvalidationDelay attribute and is measured in seconds.

It is important to understand the consequences of changing this value. If too low a time is given then there could be situations where the system under load cannot handle the end of call tasks in the given time and the global error log may see many errors containing the Java exception `IllegalStateException` which occurs when attempting to access data from an invalidated call session. One has to understand that system resources are limited and when it is under load what may have taken 100ms to complete could take longer depending on what it is that needs to be done.

The administrator should refrain from making this number too large. This is because while a call session is still valid but not representing a live call, all that information remains in memory. This may not be much but could be significant depending on the amount of data stored in element and session data by the application. Even though the session has not been invalidated, since the call has ended, Call Services is ready and will accept new calls which will allocate additional memory. Under high load, the Java application server could encounter memory issues if call sessions remain in memory for too long a period.

The JMX interface supports the ability to change the session invalidation delay at runtime. The administrator would increase this setting if `IllegalStateException` appears in the logs. They would lower the value if the JVM memory usage stays close to the maximum after each garbage collection. Keep in mind that there are many potential causes for JVM memory utilization to rise and is certainly not limited to this cause.

The default value of the session invalidation delay is sufficient to handle a heavy load without issues so the administrator is urged to be cautious when changing this value.

Application Configuration Options

To view the configuration options of an application using the JMX interface, navigate to the `VoiceApplication/<APPNAME>/Command` MBean, where `APPNAME` is the name of the application. There are four attributes listed:

- **DefautAudioPath** - this shows the audio path defining where the audio files are located (assuming the application was designed to take advantage of it).

- **GatewayAdapter** - this shows the gateway adapter that the application is using and is not editable. It is for informational purposes only.

- **SessionTimeout** – this shows the length of time, in seconds, of inactivity to consider a call session timed out.

- **SuspendedAudioFile** – The path for the audio file to play to callers when calling into an application that is suspended.

An administrator may choose to change the default audio path of an application at runtime should there be a need to change the audio callers hear quicky. One use case would be if the server that hosts the audio files is being restarted and the administrator wishes all audio to be fetched from a backup server. Note that the effectiveness of this change will be based on how consistently the application was designed to use the default audio path and also if the application explicitly sets the default audio path itself by not enabling the default audio path configuration setting, which would override the value passed here.

An administrator may choose to change the session timeout value at runtime as part of the process of debugging a problem. Under normal circumstances no session should time out because the voice browser and Call Services should be in constant communication regarding when a call starts and ends. An administrator expreiancing some sessions timing out may choose to increase this attribute to see if it resolves the issue and if not, should look into network issues. The administrator should be careful not to set this value too small because there is a risk that a normal call could time out due to the caller visiting a particularly large VoiceXML page or taking their time entering a long DTMF input. Too large a number will mean that sessions that are no longer valid will remain in memory longer and the administrator would not be able to see which sessions timed out until the timeout period elapsed.

An administrator may choose to change the suspended audio file at runtime if the application needed to be suspended due to a specific reason. For example, if a weather event required an application to be suspended, the administrator could point the suspended audio message to a recording explaining why the application is suspended rather then just pointing to a generic message. The administrator is taking advantage of the fact that this change is not persisted since it is expected that the event that caused the application's suspension is temporary.

Administration Functions

Call Services exposes several functions that allow an administrator to make both small and large changes to the applications and Call Services at runtime. They are divided into two categories: those that affect a specific application and those that affect all applications running on Call Services. An administrator can use the JMX interface as well as administration scripts to execute these functions.

Each administrator function, when activated, prompts Call Services to send a logger event reporting the function and its result so that any loggers listening to these events can log the information. The logs will then maintain a history of administration activity that can be analyzed later.

Administrator functions include the ability to add, update, and remove applications as well as suspend both an application and Call Services itself. This section describes all functions available.

Graceful Administration Activity

Administration functions are used primarily to alter an application, whether it be to update its contents or suspend its activity. Whenever changes are made to a live system handling callers, a concern is how these changes affect live callers. A robust, reliable system should strive for maximum uptime and minimal disruptions of live calls, and Call Services does this by implementing a “graceful” process for managing changes.

In the graceful process, existing callers continue to experience the application as it existed before the change, while new callers experience the change. Only after all existing callers have naturally ended their calls will the change apply to all live callers. At this time, Call Services will perform any necessary cleanup required to remove the old application configuration. In this manner, changes can be made to applications at any time, the administrator need not worry about the impact of the change on live callers as the transition will be handled gracefully.

Due to the interactive nature, when using administration scripts to perform graceful functions, the script will display a count down of callers that are actively visiting the application as they end their calls. This is provided as an aid to the administrator in determining how many callers are still experiencing the application before the change. Command line arguments passed to the scripts can turn off this countdown if desired.

When using the JMX interface or if the countdown is turned off in the administration script, the only way to track the number of callers that are still experiencing the old configuration would be to get the system status..

Updating Applications

Occasionally, an application will need to be updated. Possible changes can be small, such as renaming an audio file or altering a TTS phrase, or large, such as adding another item to a menu and creating a new call flow branch. They can involve simple configuration changes or may involve new or changed Java class files. While most changes are implemented during development time, there is a requirement to support updating an application at runtime.

The update functionality acts gracefully in that any callers on the system, at the time of update, continue to experience their calls as if the application had not been updated, while new callers experience the updated voice application. In this manner, there is no downtime when a change is implemented for an application, the callers are handled as expected.

Call Services exposes an update function for every application deployed This will updates just that application. It also has a function that will update all applications at once.

There are a few items to note when updating individual voice applications.

- The gracefulness of the update applies only to those resources controlled by Call Services. These include the application settings and call flow, element configurations, Universal Edition decision elements, and Java classes placed in the `java / application` directory of the application. The following changes are not managed by Call Services and therefore *will not* be gracefully updated:
    - Java classes placed anywhere else (including the `common` folder),
    - XML content passed to Call Services via the XML API.
    - The content of VoiceXML insert elements.
    - Other applications that the updated application transfers to or visits as part of a subroutine.
    - External back-end systems such as web services and databases (including the user management database).
    - Web servers hosting static content used by the application such as audio or grammar files.

  When each of these resources become unavailable or change, *all* callers would be affected. For small changes such as a revised audio file, this situation may be acceptable. For large-scale changes that span multiple systems, this could cause problems such as callers who are visiting an application when the update is made experiencing an error because a database is down.

  For large changes, the application should be suspended and the changes made once all callers have left the system (see the following section on suspending applications). Once the application is fully suspended, the administrator is free to make the changes and when done, the application should be updated followed by resuming it from its suspended state. This way, no caller will be in the system when the changes are made. The only disadvantage to this approach is that it will make the application unavailable for a period of time as opposed to a transparent change if the update feature alone is used. This may be a necessary tradeoff considering the consequences.
- When the update occurs, the event created by Call Services to send to any loggers that are listening will reflect when the update function was executed, not when it completed.
- If an error occurs during the update process, e.g., due to an incorrectly configured XML file, a description of the error is displayed and sent to any loggers listening to the appropriate logger events and the update is *cancelled*.

JMX Interface

To update an application using the JMX interface, navigate to the `VoiceApplication/<APPNAME>/Command` MBean, where `APPNAME` is the name of the application to update. The operations tab of this MBean will list a function named “updateApplication”. Pressing this button will cause the application to be updated and the result of the update will be displayed in a dialog box.

The administrator should be aware that there is *no confirmation* when this function is called, the update happens immediately once executed.

Note that while the function returns immediately, the old application may still be active if there were calls visiting the application at the time of the update. Only when all existing callers end will the old application configuration be removed from memory. To determine when that occurs, use the status function.

To update all applications at once using the JMX interface, navigate to the `Global/Command` MBean and click on the function named "updateAllApps" in the operations tab. The results will be displayed in a dialog box, listing each application updated. As with the application-specific update, use the status function to determine if there are callers experiencing old versions of the applications.

Administration Scripts

The scripts for updating an application are found in the `admin` folder of the application to be updated. Windows users should use the script named `updateApp.bat` and Unix users should use the script named `updateApp.sh`.

The script will first ask for confirmation of the desired action to prevent accidental execution. To turn off the confirmation, pass the parameter "noconfirm". By default, the script does not return to the command prompt until all pre-update callers are finished. Interrupting the countdown will not stop the update process, only the visual countdown. To turn off the countdown, pass the parameter "nocountdown". If the countdown is interrupted or the script is passed the nocountdown parameter then the only way to determine how many callers are experiencing the old application is to execute the status script for the system.

The script to update all applications is found in the `admin` folder of Call Services. Windows users should use the script named `updateAllApps.bat` and Unix users should use the script named `updateAllApps.sh`. The script behavior is the same as if the update script for individual application deployed on Call Services were executed in series.

The `updateAllApps` script also displays a confirmation prompt which can be turned off by passing the "noconfirm" parameter. Unlike the `updateApp` script, the `updateAllApps` script does not display a countdown of callers, it lists all the applications that are updated. The administrator would need to execute the status function to determine how many callers are visiting the old versions of the applications.

Suspending Applications

There are many situations when an application needs to be temporarily suspended. There could be scheduled maintenance to the network, the voice application could have an expiration date (say it runs a contest that must end at a specific time), or the application is to be turned off while enterprise-wide improvements are made. There may also be situations where all applications are to be put in suspension if modifications are being made that affect all applications. In each of

these situations, a caller would need to be played a designer-specified message indicating that the application has been temporarily suspended, followed by a hang-up. This is preferable to simply not answering or taking down the system, which would cause a cryptic outage message to be played.

First, the application designer defines the suspended message in the Application Settings pane in Builder for Call Studio. When the suspend order is given, Call Services produces a VoiceXML page containing this suspended audio message to all new calls followed by a hang-up. Since Call Services gracefully allows all calls currently on the system to finish normally when the command was issued, existing callers are unaware of any changes. Call Services will keep track of the active callers visiting the application and make that information available for the administrator to access. Only when this number reaches 0 will it be safe for the administrator to perform the system maintenance that required the suspension.

Call Services exposes suspend and resume function for every application deployed that acts on just that particular application. It also exposes a function that will suspend Call Services itself, which has the effect of suspending all applications. A separate resume function, resumes Call Services that restores the previous state of each application. So if an application was already suspended when Call Services was suspended, resuming Call Services leaves the application in a suspended state.

There are a few items to note when suspending a voice application:

- Only when all existing callers have exited the system will the application be officially suspended. Depending on the average length of calls to the voice application, this may take some time. Note though that the application status will appear as suspended since new callers can not enter the application and will hear the suspended audio message.
- If changes were made to an application while it was suspended, the application should first be updated before being resumed (see the previous section on the update administration function).
- The suspension applies only to those resources under the control of Call Services. External resources such as databases, other web servers hosting audio or grammar files, or servers hosting components via XML documents over HTTP are accessed at *runtime* by Call Services. If any of these resources become unavailable while there are still pre-suspension callers on the system, those calls will encounter errors that will interrupt their sessions. Any maintenance made to backend systems should be initiated after the application status shows that all pre-suspended callers are finished with their calls.
- When the suspension occurs, the event created by Call Services to send to any loggers that are listening will reflect when the suspend function was executed, not when it completed.
- If an error occurs during suspension, a description of the error is displayed and sent to any loggers listening to the appropriate logger events and the update is *cancelled*.

- Suspending a voice application still requires Call Services (and hence the Java application server) to be running in order to produce the VoiceXML page containing the suspended message. If the application server itself requires a restart, there are four possible ways to continue to play the suspended message to callers. Remember to execute the suspend function before any of these actions are taken as this is the prerequisite. The solutions are listed in order of effectiveness and desirability.
    - **Load balance multiple instances of Call Services**. In a load-balanced environment, one machine can be shut down, restarted, or reconfigured while the rest continue serving new calls. Once removed from the load balance cluster, a machine will not receive new call requests. Eventually, all existing callers will complete their sessions, leaving no calls on the machine removed from the cluster. That machine can then be safely taken down without affecting new or existing callers.
    - **Use a web server as a proxy.** In a smaller environment, a web server can be used as a proxy for an application server so that when that application server becomes unreachable, the web server itself can return a static VoiceXML page containing the suspended message to the voice browser. The web server need not be on the same machine as the application server. Once the web server is configured, Call Services can be suspended to flush out all existing callers, then the application server can be taken down and the proxy server will take over producing the suspended message VoiceXML page. The disadvantage of this approach is that the web server setup is done outside of Universal Edition and if the suspended message changes it would need to be changed in both the Builder for Call Studio and the web server configuration.
    - **Redirect the voice browser.** The voice browser can be configured to point to another URL for calls coming on the specific number. This can point to another machine running Call Services or even just a web server with a single static VoiceXML document playing the suspended message. A separate file would be needed for each application. This is a manual process and requires another machine with at least a web server (it can be on the same machine which would allow the Java application server to be restarted but would not allow the machine itself to be restarted).

JMX Interface

To suspend an application using the JMX interface, navigate to the `VoiceApplication/<APPNAME>/Command` MBean, where `APPNAME` is the name of the application to be suspended. The operations tab of this MBean will list a function named "suspendApplication". Pressing this button will cause the application to be suspended and the result will be displayed in a dialog box. To resume the application, select the function named "resumeApplication". The result will be displayed in a dialog box.

The administrator should be aware that there is *no confirmation* when these functions are called, the suspension and resumption occurs immediately once executed.

Note that while the suspend function returns immediately, the application may still be active if there were calls visiting the application at the time of the suspension. Only when all existing

callers end the calls will the application be fully suspended and the administrator is safe to take down any resources that the application depends on. To determine when all calls have ended, use the status function.

To suspend Call Services itself using the JMX interface, navigate to the `Global/Command` MBean and click on the function named "suspendCallServices" in the operations tab. The results will be displayed in a dialog box. As with the application-specific suspension, use the application-specific status function to determine if there are callers still visiting the applications. Click on the function named "resumeCallServices" to resume Call Services and restore the previous states of the applications.

Administration Scripts

The scripts for suspending and resuming applications are found in the `admin` folder of the application to be suspended. Windows users should use the script named `suspendApp.bat` and Unix users should use the script named `suspendApp.sh`. To resume the application, use the script named `resumeApp.bat` or `resumeApp.sh`.

It is possible to suspend all applications at once by accessing a script found in the `admin` folder of Call Services. Windows users should use the script named `suspendServer.bat` and Unix users should use the script named `suspendServer.sh`. To restore all applications to their original status, use the script named `resumeServer.bat` or `resumeServer.sh`. Note that these scripts do not resume all applications; they simply restore the administrator-specified status of each application. So if an application was already suspended when the server was suspended, resuming the server leaves the application in a suspended state.

Adding Applications

When Call Services starts up, it will load all applications that have been deployed to its `applications` folder. A new application that is created in Builder for Call Studio and deployed to a machine on which Call Services is already running cannot begin accepting calls until Call Services loads the new application. To load the application, execute the deploy application function. If the application is already deployed, executing this function will do nothing. If multiple new applications are to be deployed together, one can execute the deploy all applications function and all new applications will be deployed, and leave existing applications untouched.

JMX Interface

To deploy all new applications using the JMX interface, navigate to the `Global/Command` MBean and click on the function named "deployAllNewApps" in the operations tab. Pressing this button will display a dialog box with the status of each application's deployment.

Alternatively, to deploy a single new application, first use the function named "listAllNewApps" in the operations tab to get a list of new application names. Then use the "deployNewApp" function to deploy the desired application by name.

Administration Scripts

The script for deploying a specific application is found in the `admin` folder of the application to be deployed. Windows users should use the script named `deployApp.bat` and Unix users should use the script named `deployApp.sh`. The script for deploying all new applications at once is found in the `admin` folder of Call Services. Windows users should use the script named `deployAllNewApps.bat` and Unix users should use the script named `deployAllNewApps.sh`.

Removing Applications

Call Services exposes two administrative functions to handle the removal of application(s) from memory at runtime. Determining which function to use will depend on the operating system and whether the application being removed is actively handling calls.

The first method involves executing the release application function of the application to be removed. This prompts Call Services to first suspend the application then remove it from memory when all the active callers at the time the function was executed, have naturally ended their sessions. It suspends the application first to prevent new callers from entering the application. Once all active callers are done visiting the application, the folder of the application can be deleted (or moved) from the Call Services `applications` folder. This function affects only a single application so if multiple applications are to be removed using this method, the administrator would have to execute this function for each application.

Note that on the Microsoft Windows operating system, a user attempting to delete an application folder after the releaseApp function is called may be prevented from doing so by the OS if the application references Java application archive (JAR) files placed within the `java/application/lib` or `java/util/lib` directories. This is due to the system keeping an open file handle for JAR files that will not be released until a garbage collection event occurs. As a result, the administrator will have to wait until the garbage collector activates before being able to delete the directory. The time to wait will be determined by how often garbage collection is run. A rule of thumb is that a high load system or one with a small amount of memory will encounter garbage collection often, a low volume system or one with a large amount of memory will take longer.

The second method supports the ability to delete multiple applications at once. This time one must first delete (or move) the folders holding the desired applications to be deleted. After which, the flush all old applications function is executed and Call Services will suspend then remove from memory all the applications that it no longer finds in the `applications` folder. As with the other method, the application is not removed from memory until all callers have ended their visits.

There are certain issues with the second method:

- If an application relies on files found within its folder at runtime, there may be problems with existing callers reaching a point where these files are needed and they will not be found.

- This process may not work on Microsoft Windows since Windows will not allow the deletion of a folder when resources within it are open. For example, the application loggers may have open log files located within the application's `logs` folder. This may work if no loggers are used or the only loggers used are those that do not rely on files stored within the application folder.

JMX Interface

To delete an application using the JMX interface, navigate to the `VoiceApplication/<APPNAME>/Command` MBean, where `APPNAME` is the name of the application to update. The operations tab of this MBean will list a function named "releaseApplication". Pressing this button will cause the application to be suspended and then removed from memory when all active callers visiting the application at the time the function was executed have completed.

The administrator should be aware that there is *no confirmation* when this function is called, the application is suspended and removed from memory immediately once executed.

Note that while the function returns immediately, the application will remain active if there were calls visiting the application at the time of the release. Only when all existing callers end the call will the application be removed from memory. To determine if there are active callers, use the status function.

To delete all applications whose folders have been removed from the `applications` folder of Call Services using the JMX interface, navigate to the `Global/Command` MBean and click on the function named "releaseAllOldApps" in the operations tab. The results will be displayed in a dialog box, listing each application deleted. As with the application-specific update, use the status function to determine when the callers finish their visits to the applications.

Administration Scripts

The scripts for deleting an application are found in the `admin` folder of the application to be updated. Windows users should use the script named `releaseApp.bat` and Unix users should use the script named `releaseApp.sh`.

The script will first ask for confirmation of the desired action to prevent accidental execution. To turn off the confirmation, pass the parameter "noconfirm". By default, the script does not return to the command prompt until all callers are finished with their calls. Interrupting the countdown will not stop the release process. To turn off the countdown, pass the parameter "nocountdown". If the countdown is interrupted or the script is passed the nocountdown parameter then the only way to determine how many callers are actively in the application is to execute the status script for the system.

The script to release all applications whose folders have been removed from the `applications` folder of Call Services is found in the `admin` folder of Call Services. Windows users should use the script named `flushAllOldApps.bat` and Unix users should use the script named `flushAllOldApps.sh`. All applications whose folders have been removed will be suspended and when their active calls have ended will be removed from memory.

The `flushAllOldApps` script also displays a confirmation menu which can be disabled by passing it the "noconfirm" parameter. Unlike the `releaseApp` script, the `flushAllOldApps` script does not display a countdown of active callers, it will lists all the applications that were

deleted. The administrator would need to execute the status function to determine how many callers are actively in the applications.

Updating Common Classes

When performing an application update, all the data and Java classes related to an application will be reloaded. Java classes placed in the `common` folder of Call Services are not included in the application update. Call Services provides a separate administrative function to update the `common` folder.

There are a few items to note about this function:

- The update affects *all* applications that use classes in the `common` folder, so executing this function could affect applications that have not changed. Therefore, take precaution when executing this function.
- The update affects *all* classes in the `common` folder, whether they were changed or not. This is usually not a issue unless those classes contain information in them that reloading would reset (such as static variables).
- Due to the fact that this function reloads classes that affect all applications, and those classes may themselves prompt the loading of configuration files from each application that uses those classes, the function may take some time to complete depending on the number of classes in the `common` folder and the number and complexity of the deployed applications.
- Changes are immediate, they are *not* done gracefully. Since this potentially affects all applications, the administrator must be aware of this.

JMX Interface

To update common classes using the JMX interface, navigate to the `Global/Command` MBean and click on the function named “updateCommonClasses” in the operations tab. The results will be displayed in a dialog box.

Administration Scripts

The script for updating common classes is found in the `admin` folder of Call Services. Windows users should use the script named `updateCommonClasses.bat` and Unix users should use the script named `updateCommonClasses.sh`. The script will ask for confirmation of the desired action to prevent accidental execution. disable the confirmation, pass the parameter “noconfirm”.

Accessing Logger Methods

The Call Services JMX interface has the ability to access logger instances and execute methods on them. The logger class must first tag the methods it considers acceptable to be called by an administrator. Refer to the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for information on how to build custom loggers for serviceability.

To run logger methods via the JXM interface, navigate to the `VoiceApplication/<APPNAME>/Logger/LOGGERNAME` MBean, where `APPNAME` is the name of the application to update and `LOGGERNAME` is the instance name of the logger to access. Note that if the logger does not expose any methods that can be called by an administrator, no methods will appear in the Operations tag.

No loggers shipped with Call Services exposes methods to allow runtime changes. Custom loggers build by developer, however, may still provide this functionality.

Getting/Setting Global and Application Data

Global data holds information that applications decide to share across other applications deployed on Call Services. Application data holds information that applications decide to share across all calls to the application. The Call Services JMX interface provides the ability for an administrator to view the contents of these variables, change their values, and even create new variables.

This functionality provides an administrator direct access to live data that is being created on the system and can provide them some control of how applications operate. This is only possible when the application designers design that functionality into the applications. For example, an application designer for a utility company can build their application to look for the existence of a global data variable reporting a power outage. The administrator then creates the global data variable when a power outage occurs and automatically the applications will start reporting the power outage to callers. The administrator can then delete the global data variable to signify the power has been restored. While this same functionality could be achieved with a database, this is a simpler approach to handle predictable situations without the need to use a database.

Global Data Access

To access global data using the JMX interface, navigate to the `Global/Data` MBean. The Attributes tab lists all the global data variable names in an attribute named "AllGlobalDataNames" (the value may need to be expanded in order to see all the global data names). The Operations tab lists four functions that can be executed by the administrator for global data:

- **setGlobalData** – This function allows the administrator to create a new global data variable. The function takes two inputs, the first being the name of the variable and the second is the value. Click on the button to set the global data and the result will appear in a dialog box. Note that if there exist global data with the same name it will be overridden.
- **removeGlobalData** – This function allows the administrator to delete a global data variable. The function takes one input: the name of the global data variable to delete. Click on the button to remove the global data and the result will appear in the dialog box.
- **removeAllGlobalData** – This function allows the administrator to delete all global data, whether it was created by the administrator or applications. Click on the button to remove all

global data and the result will appear in the dialog box. Be careful when using this function as it could affect the performance of applications that rely on global data.

- **getGlobalData** – This function allows the administrator to retrieve the value of a global data variable. The function takes one input: the name of the global data variable to retrieve. Click on the button to display a dialog box showing value of the global data.

Application Data Access

To access application data using the JMX interface, navigate to the `VoiceApplication/<APPNAME>/Data` MBean, where `APPNAME` is the name of the application whose application data is to be accessed. The Attributes tab lists all the application data variable names in an attribute named "AllApplicationDataNames" (the value may need to be expanded in order to see all the application data names). The Operations tab lists four functions that can be executed by the administrator for application data:

- **setApplicationData** – This function allows the administrator to create a new application data variable. The function takes two inputs, the first being the name of the variable and the second being the value. Click on the button to set the application data and the result will appear in a dialog box. Note that if there is already application data with the same name it will be overridden.
- **removeApplicationData** – This function allows the administrator to delete a application data variable. The function takes one input: the name of the application data variable to delete. Click on the button to remove the application data and the result will appear in the dialog box.
- **removeAllApplicationData** – This function allows the administrator to delete all application data, whether it was created by the administrator or applications. Click on the button to remove all application data and the result will appear in the dialog box. Be careful with this function as it could affect the performance of the application.
- **getApplicationData** – This function allows the administrator to retrieve the value of a application data variable. The function takes one input: the name of the application data variable to retrieve. Click on the button to display a dialog box with the value of the application data.

Administrator Log Access

Call Services ships with various default loggers, including administration history loggers that store a history of the administration activity taken such as when Call Services started up, when an application was updated, the results of he suspension of Call Services, etc. These logs, which are rotated daily, are useful to an administrator as an audit history of administrator activity. As a convenience, the JMX interface exposes methods for the administrator to access the contents of these logs instead of viewing the files in a text editor.

Please note that the application designer and administrator has the ability to define any loggers desired for the applications as well as for Call Services, including removal of the default administration history loggers. If this is done, then these functions will return error messages that explain that the log files could not be found.

To view an application's administration history log using the JMX interface, navigate to the `VoiceApplication/<APPNAME>/Command` MBean, where `APPNAME` is the name of the application to view. The operations tab of this MBean will list functions named "retreiveAdminHistoryToday" and "retreiveAdminHistoryAll". Clicking on the first will open up a scrollable window listing the contents of the administration history log file from the day the function is called. Clicking on the second will open up a scrollable window listing the contents of all administration history logs concatenated.

To view Call Services' administration history log using the JMX interface, navigate to the `Global/Command` MBean. The operations tab of this MBean list functions with the same name and functionality as the application functions do except that the files accessed are for the global administration history.

Administration Function Reference

The following lists all the administration functions provided by Call Services and whether they are available from the JMX interface and/or via script.

Application-Level Functions

Function	JMX	Script	Description
Suspend Application	Yes	Yes	Suspends the application in which the function belongs to.
Resume Application	Yes	Yes	Resumes the application in which the function belongs to.
Deploy Application	No	Yes	Prompts Call Services to load the application in which the function belongs to (does nothing if the application is already deployed).
Update Application	Yes	Yes	Prompts Call Services to reload into memory the configuration of the application in which the function belongs to.
Release Application	Yes	Yes	Prompts Call Services to remove from memory the application in which the function belongs to so that its folder can be deleted.
Logger Method Call	Yes	No	Allows an administrator to call a method on a logger instance for the application. The logger must specify which methods can be called.

Table 3-2

Call Services-Level Functions

Function	JMX	Script	Description
Suspend Server	Yes	Yes	Suspends all applications deployed on Call Services.
Resume Server	Yes	Yes	Restores the status of each application to the original state at the time Call Services was suspended.
Deploy All New Applications	Yes	Yes	All applications deployed to Call Services since the last time the application server started up or the deploy all new applications function was called are now loaded into memory and can handle calls.
List All New Applications	Yes	Yes	Lists the names of all new voice applications so that their names may be known to be deployed using Deploy New Application.
Deploy New Application	Yes	No	Loads and deploys the specified voice application.
Flush All Old Applications	Yes	Yes	When called, all applications in Call Services whose folders were deleted are removed from memory.
Update All Applications	Yes	Yes	Prompts each application deployed on Call Services to load its configuration from scratch from the application files.
Update Common Classes	Yes	Yes	Reloads all classes deployed in the `common` directory of Call Services.

Table 3-3

Call Services Metrics

The more information an administrator has, the better he will be in determining the health of the system. Call Services provides a significant amount of information on various metrics to allow the administrator to understand what is going on within the system. Armed with this information, the administrator will be able to react quickly to situations which could degrade the stability of the system.

The information falls into three categories: aggregate information, information on peaks, and average information. Aggregate information, such as the total number of calls handled, is helpful in determining how much work Call Services has done so far. Peak information, such as the maximum concurrent calls occurring in the last 10 minutes, is very helpful in understanding how load is distributed on the system and can help the administrator understand how the volume is changing. Average information, such as the average HTTP request completion time, helps the administrator compare current metrics against historical averages.

The metrics maintained by Call Services is available only through the JMX administration interface. To view the metrics, navigate to the `Global/Metrics` MBean. The Operations tab lists 15 separate functions that the administrator can call to obtain very specific information concerning how the system is running as well as how it has performed in the past. Many of the

functions take a time duration as an input. It will dispay information of the specified period up to maximum of 60 minutes.

The following list describes each function and the information it returns:

- **totalCallsSinceStart** – Returns the total cumulative number of calls handled by Call Services since it launched. This number will continually rise and only resets when Call Services or the Java application server is restarted.
- **maxConcurrentCallsInLast** – Returns the most number of simultaneous callers that occurred in the last X minutes where X is entered by the administrator (maximum of 60 minutes) and when the maximum was reached. This is helpful in determining how close the call volume reached the license limit on simultaneous callers. Knowing when the maximum value is reached can be very helpful in determining if call volume is rising. For example if the peak call volume for the last 10 minutes was achieved very close to present time, that would indicate that call volume is rising.
- **avgConcurrentCallsInLast** – Returns the average number of simultaneous callers encountered in the last X minutes where X is entered by the administrator. This data is helpful in determining if a peak was an isolated occurrence or a sign of a trend. For example if the maximum number of concurrent calls in the last hour was 100 but the average is 10, then there is less to be alarmed about since the 100 peak did not last long and can be attributed to a temporary spike. If the average were 90, then this would indicate that the call volume is very steady.
- **maxReqRespTimeInLast** – Returns the maximum time, in milliseconds, it took Call Services to produce an HTTP response in the last X minutes where X is entered by the administrator and when the maximum was reached. A voice browser makes an HTTP request to Call Services, which then must respond with a VoiceXML page. Clearly a large response time would be cause for concern as a slow performing system will cause callers to think that the application has encountered errors and in extreme cases could cause the voice browser to time out a request and end a call with an error.
- **avgReqRespTimeInList** – Returns the average time, in milliseconds, it took to produce an HTTP response in the last X minutes where X is entered by the administrator. This value gives the administrator a good idea of how long it takes Call Services to handle responses given the call volume. This could help the administrator decide if the system is overloaded and beginning to affect the perception of callers regarding the responsiveness of the application. It also establishes a baseline to compare with the maximum response time. A maximum response time significantly higher than the average could be an indication that there is a problem with an external resource accessed by a custom element such as a database or web service and the few calls that visited that element suffered from bad performance. It could also help determine if the maximum response time was an isolated event or an indication of a trend. For example if the maximum response time were 500ms which occurred near the present, the average was 400ms, the fact that the peak was 500ms is not alarming, because the average is so high. In this situation the administrator may choose to

throttle down the calls being handled by the system to bring the response times back down to more acceptable levels.

- **timeoutCallsInLast** – Returns the total number of calls that ended with a timeout in the last X minutes where X is entered by the administrator. More specifically, this counts calls where the "result" action of the "end" category is *timeout*. See Chapter 5: Call Services Logging in the section entitled The Application Activity Logger for more on the different results and how ended values. Under normal circumstances a call should never time out. Many different types of conditions can yield session timeouts on Call Services and so knowing if there are timeouts in the last period of time would tell the administrator how widespread these issues are.
- **failedCallsInLast** – Returns the total number of calls that ended with an error in the last X minutes where X is entered by the administrator. More specifically, this counts calls where the "result" action of the "end" category is *error*. See Chapter 5: Call Services Logging in the section entitled The Application Activity Logger for more on the different results and how ended values. This helps the administrator determine how widespread a bug or other issue that caused a call to end in an error is. For example, if the last 60 minutes yielded only one failed call, while the issue should be investigated, it may not be a symptom of a larger more prevalent issue.
- **timeoutCallsSinceStart** – Returns the total number of calls that ended with a timeout since Call Services launched. More specifically, this counts calls where the "result" action of the "end" category is *timeout*. See Chapter 5: Call Services Logging in the section entitled The Application Activity Logger for more on the different results and how ended values. This information is good to compare with the number of timed out calls in the past X minutes as if the numbers are close it could mean that the issue that is causing the timeouts is a recent occurrence. It also gives an indication of the stability the system and will allow the administrator to calculate the percentage of calls that had encoutnered timeouts.
- **failedCallsSinceStart** – Returns the total number of calls that ended with an error since Call Services launched. More specifically, this counts calls where the "result" action of the "end" category is *error*. See Chapter 5: Call Services Logging in the section entitled The Application Activity Logger for more on the different results and how ended values. This information is good to compare with the number of failed calls in the past X minutes as if the numbers are close it could mean that the issue that is causing the errors is a recent occurrence. It also gives an indication of the stability the system and will allow the administrator to calculate the percentage of calls that had errors.
- **maxLoggerEventQueueSizeInLast** – Returns the largest the logger event queue encountered in the last X minutes where X is entered by the administrator and when the maximum was reached. For an explanation of the logger queue, see the section titled Tuning Logger Options earlier in this chapter. This value will help the administrator understand, in an abstract way, how much Call Services is logging. While it is not unusual for this number to be large, the administrator can track a trend and if this number continually increase, it could be an indication that the system cannot handle the logger event load and could eventually result in

memory problems. The time when the maximum was reached can help indicate if Call Services is able to handle the incoming stream of logger events.

- **maxLoggerThreadCountInLast** – Returns the most simultaneous threads Call Services was using to handle loggers in the last X minutes where X is entered by the administrator and when the maximum was reached. For an explanation of the logger thread pool, see the section titled Tuning Logger Options earlier in this chapter. This would be another indication of whether Call Services is able to keep up with the stream of logger events as if the number is close to the maximum thread pool size it is an indication that Call Services has almost reached its limit in handling events. When the maximum was reached will help determine if this is happening recently. Keep in mind that when all the threads in the pool are actively handling logger events, the logger event queue will rise rapidly. So if this value is at the maximum thread pool size, then the maxLoggerEventQueueSizeInLast function would display rapidly increasing queue sizes.
- **callTransferRate** – Returns the percentage of calls that ended in a blind telephony transfer. More specifically, this counts calls where the "how" action of the "end" category is *call_transfer*. This could help the administrator determine what percentage of callers decided to speak to an agent rather than complete the call within the automated voice application.
- **callAbandonRate** – Returns the percentage of calls that ended with the caller hanging up. More specifically, this counts calls where the "how" action of the "end" category is *hangup*. See Chapter 5: Call Services Logging in the section titled The Application Activity Logger for more on the different results and how ended values. Keep in mind, though, that despite the name, a caller hanging up is not necessarily a bad thing since the caller could hang up right before the application hung up on the caller and the end category would still be *hangup*. This value would therefore be a good indication of how callers interact with the applications on the system.
- **callCompleteRate** – Returns the percentage of calls that ended normally. More specifically, this counts calls where the "result" action of the "end" category is *normal*. See Chapter 5: Call Services Logging in the section titled The Application Activity Logger for more on the different results and how ended values. This does not count calls into a suspended application, calls ending in an error or timeout, or calls ending due to an element manually invalidating the session. It is expected that this percentage be close to 100%.
- **averageCallDuration** – Returns the average duration of all calls handled by Call Services, in seconds. This helps the administrator determine if a particular call being analyzed represents a typical call since a particularly long call could indicate a caller having trouble with the application and a short call could indicate caller frustration with the application.

Chapter 4: User Management

Call Services includes a user management system for basic personalization and user activity tracking. The primary reason for a user management system is to facilitate the customization of voice applications depending on user preferences, demographics, and prior user activity. It is not meant to be a replacement for fully featured commercial user management systems and can be used in conjunction with those systems. Additionally, Universal Edition voice applications do not require he presence of a user management system, it is provided as an aid to application designers.

While the bulk of the user management system is designed to track individual users, its most basic form can still prove useful for those applications that do not need (or want) to track individual users but would still like to be able to provide very simple personalization such as playing “Welcome back” when a call is received from a phone number that has called before. When turned on, the user management system automatically keeps track of information based on the phone numbers of callers. This is available automatically; the developer need not do any additional work.

The user management system is fully integrated into Call Services. An API is included to provide two different interfaces to the user management system. The first interface is used to manage the user database, allowing separate, external processes to populate, maintain, and query the system. The second interface is provided for dynamic components of a voice application to allow runtime updates and queries to the system. This second interface allows a voice application to perform tasks such as playing a customized message to registered users, making decisions based on user demographics or history, and even adding new users after the caller completes a successful registration process. The API has both Java and XML versions. These APIs are fully detailed in the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio.

Deployment

The user management system is simply a database accessed by Call Services. Each hosted voice application may refer to a separate user management database or may share databases if users are to be shared across applications. The user management system can be activated by providing a JNDI name for the relational database the user data is to be stored. This is done in the settings pane for the application in the Builder for Call Studio. Currently, the databases supported are MySQL and SQLServer. Note that the application server must be set up to manage connections to this database beforehand.

Once the database itself is set up, Call Services automatically handles the process of creating the database tables.

Database Design

Figure 4-1 displays an ER diagram of the database tables comprising the user management system. The following sections describe each table individually and its purpose.

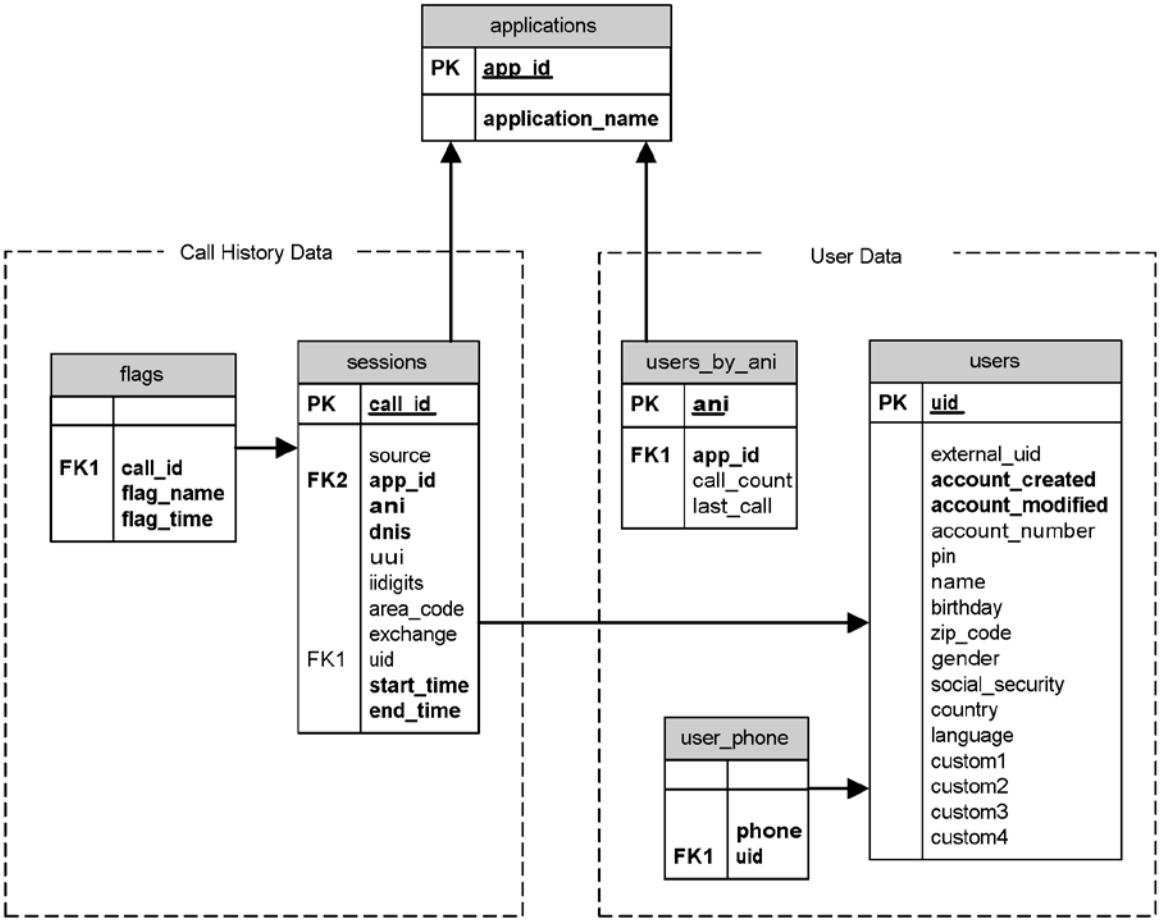


Figure 4-1

Applications

This table is used to provide a primary key for the voice applications utilizing this user management database. Most voice applications will utilize their own user management system in which case this table will have only one entry in it. For those applications that share a common user management system, this table's key is used to keep track of the activities of users visiting each application, should that separation be necessary.

Column	Type	Description
app_id	integer (primary key)	Automatically generated application ID.
application_name	varchar(50)	The name of the application with the specified application ID.

User Data

The tables under this category are used to store information about the users in the system.

users

This table is the main user table. Each row contains the information for a single user. Both demographic and account information are stored here. The table specification is as follows:

Column	***Type***	***Description***
uid	integer (primary key)	This is a user ID automatically generated by the system to identify a particular user. Once a call is associated with a UID, the system knows the caller's identity. The user management system relies on this UID throughout.
external_uid	varchar(50)	If an external user management system is used in conjunction with this one there must be a way to link a user on the Universal Edition system with one in the external system. This column stores the ID for this user on the external system to provide that link. Can be null if the Universal Edition user management system is used exclusively.
account_created	datetime	This stores the time the user was added to the system. It will always have a value.
account_modified	datetime	This stores the time of the last update to this user in the system. It will always have a value.
account_number	varchar(50)	Some voice applications identify users by account numbers. If so, the account number should be stored here, otherwise it can be null.
pin	varchar(20)	If the voice application uses a PIN to verify the user, the PIN is stored here. Null if no PIN is used or required.
name	varchar(50)	The user's name. Can be null.
birthday	varchar(50)	The user's birthday. Can be null.
zip_code	varchar(10)	The user's zip code. Can be null.
gender	varchar(10)	The user's gender: "male", "female", or null if not stored.
social_security	varchar(10)	The social security number of the user. Can be null.
country	varchar(50)	The user's full country name. Can be null.
language	varchar(50)	The language the user speaks or prefers. This can be used to provide audio content in different languages. Can be null.
custom1-custom4	varchar(200)	These columns are provided to allow the developer to place custom user-related data in the system. It can be used for such data as e-mail addresses, financial account

		balances, proprietary IDs, etc. Can be null.

user_phone

This table is an adjunct to the main user table. It is used to store the phone numbers associated with the user. The reason this data is placed in a separate table is to allow an application to associate more than one phone number with a user. For example, a voice application allowing a user to associate with their account both their home and work numbers can automatically recognize who the caller is when calls are received from either number, rather than requiring them to log in. If multiple phone numbers are not required or necessary, this table can contain one entry per account or remain empty. Since there may be multiple rows in the system with the same UID, there is no primary key to this table. The table specification is as follows:

Column	***Type***	***Description***
phone	varchar(10)	A phone number to associate with this account.
uid	integer (foreign key)	The UID identifying the user.

users_by_ani

This table is used to track calls made from specific phone numbers (ANIs). This table is automatically updated by Call Services and need only be queried by the developer when information about a caller is desired. The table contains information about the number of calls and the last call made from a phone number. This information can be used to welcome a caller back to the application or warn that menu options have changed since their last call even if the application itself is not set up to track individual users through logins. The table specification is as follows:

Column	***Type***	***Description***
ani	varchar(10)	The phone number of the caller.
app_id	integer	The application the caller called into. This exists in case multiple applications share a common user management system.
call_count	integer	The number of calls received by this phone number to this application.
last_call	datetime	The last time a call was received by this phone number to this application.

Historical Data

Tracking user information is only part of a user management system. Many applications benefit from knowing information about the past history of a user's interaction with the phone system. This component of the user management system is automatically updated by Call Services and need only be queried by the developer when information about user(s) is desired.

sessions

This table contains records of every call made to the system. It stores telephony information about the call as well as when the call was made. The table specification is as follows:

Column	***Type***	***Description***
call_id	integer	This is an automatically incremented ID for the call. It is used exclusively within the user management system.
source	varchar(50)	This column contains the name of the application which transferred to this one or is null if the application was called directly.
app_id	integer	The application ID of the application called. If the user management system is not shared across multiple applications, this ID would be the same for all calls.
ani	varchar(10)	The ANI of the originating caller. Is NA if the ANI was not sent by the telephony provider.
dnis	varchar(10)	The DNIS of the originating caller. Is NA if the DNIS was not sent by the telephony provider.
uui	varchar(100)	The UUI of the originating caller. Is NA if the UUI was not sent by the telephony provider.
iidigits	varchar(100)	The IIDIGITS of the originating caller. Is NA if the IIDIGITS was not sent by the telephony provider.
area_code	varchar(10)	The area code of the originating caller. Is null if the ANI is NA.
exchange	varchar(10)	The exchange of the originating caller. Is null if the ANI is NA.
uid	integer	The UID of the caller if the call was associated with a user. If not, it will appear as null.
start_time	datetime	The date and time the visit to the application began. If no other application can transfer to this one, this will be the time the call was made.
end_time	datetime	The date and time the visit to the application ended. If this application cannot transfer to any other application, this will be the time the call ended in a hang-up or disconnect.

flags

This table contains records of the flags triggered by every call made to the system. Since flags are used to indicate important parts of the voice application, knowing what areas of the voice application people visited in the past can be very useful. The table specification is as follows:

Column	***Type***	***Description***
call_id	integer	This refers to the call ID of the call.
flag_name	varchar(100)	This is the name of the flag that was triggered.
flag_time	datetime	This is the date and time the flag was triggered.

Chapter 5: Call Services Logging

Logging plays an important part in voice application development, maintenance, and improvement. During development, logs help identify and describe errors and problems with the system. Voice applications relying heavily on speech recognition require frequent tuning in order to maximize recognition effectiveness. Voice application design may also be changed often, taking into account the behaviors of callers over time. The more information an application designer has about how callers interact with the voice application, the more that can be done to modify the application to help callers perform their tasks faster and easier.

For example, a developer could determine the most popular part of the voice application and make that easier to reach for callers. If a large proportion of callers ask for help in a certain part of the application the prompt might need to be rewritten to be clearer. After analyzing the utterances of various callers, the effectiveness of grammars can be determined so that additional words or phrases can be added or removed. None of this is possible without detailed logs of caller behavior. While each component of a complete IVR system such as the voice browser and speech recognition system provide their own logs, Call Services provides logs that tie all this information together with the application logic itself. This chapter explains everything having to do with logging on Call Services.

Due to the importance of logging, Call Services has been designed to offer the maximum flexibility with regards to what can be logged, how it is logged, and where it is logged. The logs generated by Call Services by default can be customized to fit the needs of a deployment. In addition, a Java API exists that allows developers to create their own ways of handling logging for better integration with the deployed environment or tailored specifically for special needs.

Loggers

Call Services handles all logging activity through the use of loggers. Loggers are plugins to Call Services that listen for certain logging events and handle them in a custom manner, from storing the information in log files, sending the information to a database, or even to interface with a reporting system. Any number of loggers can be used, even multiple instances of the same logger. A logger may or may not require a configuration that will allow the designer to customize how the logger performs.

Call Services comes with several loggers that provide all necessary information in text log files. Some provide configurations to allow for a level of customization in how the loggers perform. Call Services exposes a Java API to allow developers the flexibility of creating their own loggers to allow for even more customization. See the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for detailed information on how to build custom loggers.

Call Services communicates with loggers by triggering logging events that the loggers listen for and then deal with. Call Services activates loggers in a fully multi-threaded environment to maximize performance.

Loggers are divided into two categories: global loggers and application loggers. Global loggers are activated by logging events that apply to Call Services as a whole and that is not directly related to any particular application (for example a record of all calls made to the Call Services instance). Application loggers are activated by logging events that apply to a particular application running on Call Services (for example a call visiting an element). Each logger type deals with a separate list of possible logging events and a different Java API is used. Each logger type is also given a separate area to store logs, though a logger may choose to ignore this area in the case that it does not log to files.

Global Loggers

The `global_config.xml` file found in the `conf` directory of Audium Home is used to define the global loggers Call Services is to use. The administrator can define any number of global loggers to be simultaneously active, even multiple instances of the same logger class. This file also lists the names of the configuration files for these loggers, if they are configurable. The configuration files must be placed in the same `conf` directory as the `global_config.xml` file. The `global_config.xml` file and any configuration files must be edited by hand, there is no interface for editing them. Refer to Chapter 6: Call Services Configuration for more details about this file and how to define global loggers within it.

Global loggers will be loaded by Call Services when it starts up and remain in memory until it is shut down. Any change made to the `global_config.xml` file will not be loaded until Call Services is restarted.

Call Services provides the `logs` folder of Audium Home for log file storage should the Global Loggers require it. To keep each logger instance's logs separate, a subfolder with the name of the logger instance is created and all logs generated by the logger instance are stored there.

By default, Call Services utilizes three loggers to create text log files containing Call Services-specific information: a log that keeps track of calls made to the system, a log for tracking Call Services administration activity, and an log that shows errors that occur on the Call Services level (as opposed to the application level). The global error logger requires a configuration that allows for detailed control over how the logger operates.

The following sections describe these three pre-built global loggers, their configurations (if any), and the information stored in their logs.

The Global Call Logger

The global call logger records a single line for every application visit handled by Call Services into a text call log. Most calls will begin and end in a single application so in that case a line in

the call log is equivalent to a physical phone call. For situations where one application performs an application transfer to another application, a separate line will be added to the call log for each application visit despite the fact that they all occur in the same physical call. Since each application visit is logged separately in each application's own log file, the call log provides a way to stitch together a call session that spans multiple applications.

The call log file names are in the format "call_logYYYY-MM-DD.txt" where YYYY, MM, and DD are the year, month, and day when the call log was first created. By default, the log folder for is named "GlobalCallLogger" (though the name is set in the `global_config.xml` file and can be changed by the administrator). Call log files are rotated daily. The file is organized in a comma-delimited format with 6 columns:

- **CallID**. This is a non-repeating value generated by Call Services to uniquely identify calls. It is designed to be unique even across machines, as the log files of multiple machines running the same applications may be combined for analyses. The format of the session ID is IP.SECS.INCR where IP is the IP address of the Call Services instance on which the call originated, SECS is a large integer number representing the time the application visit was made and INCR is an automatically incremented number managed by Call Services. Each part is delimited by dots and contains no spaces. For example: 192.168.1.100.1024931901079.1.

  NOTE: If a voice application uses a Subdialog Invoke element to transfer across multiple Call Services instances, the IP address included in the CallID is the IP address of the instance the call started on. Because of this, it is possible that a CallID in log files on one machine may contain an IP address for another machine. This allows a physical call to be traced across multiple servers (from a logging standpoint), even if Subdialog Invoke is used to transfer to between various voice applications.

- **SessionID**. The session ID is used to track a visit to a specific application. Therefore, with application transfers, one call ID may be associated with multiple session IDs. For this reason, session IDs are simply the call ID with the application name appended to the end. For example: 192.168.1.100.1024931901079.1.MyApp.

- **callers**. This integer represents the total number of callers interacting with the system at the time the call was received (including the current call).

- **order**. A number indicating the order of each application visited in a call. The order begins at 1. This column exists to report the order in which a caller visited each application should the data be imported to a database.

- **Application**. The name of the application visited.

- **Time**. A timestamp of the application visit in the format "MM/DD/YYYY HH:MM:SS.MMM" where the hour is in 24-hour time and MMM represents a 3-digit millisecond value. This represents when the call was received or the application transfer occurred.

The Global Error Logger

The Global Error Logger records errors that occur outside the realm of a particular application. Application-level errors are logged by application-level loggers, which are described later in this chapter. Another type of error that the Global Error Logger receives is an application-level error that encountered trouble with its logging. In order to prevent the loss of the data, Call Services activates a global logger event with the original application error as a backup.

The error log file names are in the form "error_logYYYY-MM-DD.txt" where YYYY, MM, and DD are the year, month, and day when the error log was first created. By default, the log folder is named "GlobalErrorLogger" (though the name is set in the `global_config.xml` file and can be changed by the administrator). Global error log files are rotated daily. Note that if no error occurred on a particular day, no error log will be created. The file is organized in a comma-delimited format with 2 columns:

- **Time**. The time the error occurred.
- **Description**. The error description. One possible value can be *max_ports*, indicating the caller was put on hold because all the Universal Edition license ports were taken. While the call was eventually handled correctly, this is placed here as a notice that the license may not have enough Universal Edition ports to match caller volume. Another value is *bad_url:[URL]*, indicating that a request was made to Call Services for a URL that could not be recognized. This most likely will occur if the voice browser refers to an application that does not exist. The last description is *error*, indicating that some other error occurred.

Note that the global error log is not designed to be parsed, even though the columns are separated with commas. This is because when the error log reports a Java-related error, it may include what is called a "Java stack trace", which contains multiple lines of output.

The Global Error Logger utilizes a configuration to control how it logs certain types of errors and how often the log files should be purged. The configuration is specified as an XML file created by the designer and placed in the `conf` directory of Audium Home.

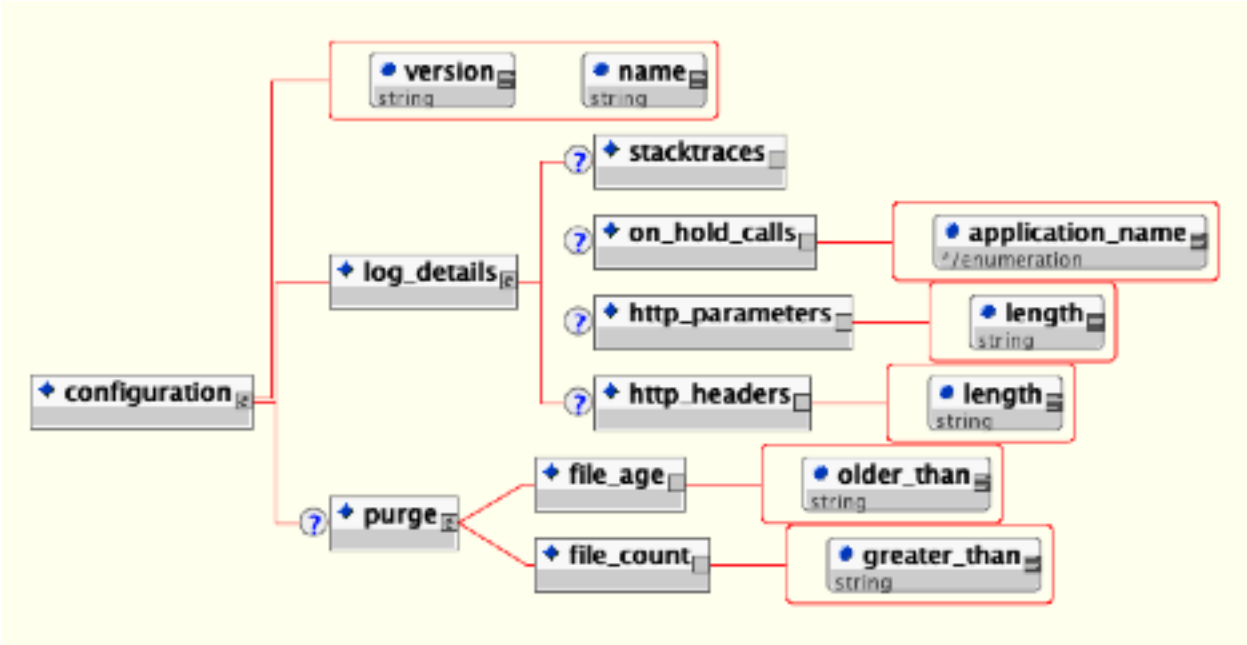


Figure 5-1

Figure 5-1 displays the format for the XML Global Error Logger configuration file. The main tag for the configuration, `configuration`, has two attributes, `name` and `version`. Name is expected to contain the logger instance name. The version is expected to include the version of the configuration, which is currently "1.0". The subsequent sections describe the functionality of the various tags in the configuration.

Global Error Logger Configuration: Log Details

The `<log_details>` tag controls which errors to log and what information to include about those errors. The possible child tags are:

- **<stacktraces>**. This optional tag is used to indicate that any Java errors that occur should also have their stack traces printed in the log. The absence of this tag indicates not to include stack traces.
- **<on_hold_calls>**. This optional tag is used to indicate that a call that was put on hold should be logged. The `application_name` attribute can have the values `true` and `false`, `true` being to include the name of the application the caller attempted to reach when being put on hold and `false` to not include the application name.
- **<http_parameters>**. This optional tag is used to indicate that an error caused by an unrecognized URL (such as a request for an application that does not exist) should include the HTTP parameters passed to the URL. This can be helpful to know since it could help determine why the request was made. The `length` attribute provides a limit, in a number of characters, to be included in the log. This prevents the log from being filled up with too much parameter data. Note that the parameter data appears on one line, no matter how long.

- **<http_headers>**. This optional tag is used to indicate that an error caused by an unrecognized URL (such as a request for an application that does not exist) should include the HTTP headers passed to the URL. This can be helpful to know since it could help determine why the request was made. The `length` attribute provides a limit, in a number of characters, to be included in the log. This prevents the log from being filled up with too much header data. Note that the header data appears on one line, no matter how long.

Global Error Logger Configuration: File Purging

The Global Error Logger can be configured to automatically delete files that conform to certain criteria. Properly configured, this will allow an administrator to avoid having the system's hard drive fill up with logs, which would prevent new calls from being logged.

A few notes about file purging must be given:

- Since loggers are activated only when events occur in a call, the file purging activity will only take place when an error event occurs. As a result, a system that encounters no errors will not automatically delete files until a new error occurs.
- When the Global Error Logger starts up for the first time, it will apply the purging strategy on any files that exist in the logger directory. Therefore, if an application server is shut down with files in the logger directory and then restarted a long time later, these files could be deleted when the application server starts up and the logger initializes.
- The Global Error Logger applies its purging strategy to any files found in its logger directory, including non-error log files. So should other files be added to the logger folder after the application server has started could be deleted when the Error Logger encounters a new error.

The optional `<purge>` tag defines the purging strategy. If this tag does not appear in the configuration, no file purging will take place. The tag can contain one of the following child tags:

- **file_age**. The Global Error Logger will delete error log files older than X days, where X is an integer greater than 0 specified in the `older_than` attribute.
- **file_count**. The Global Error Logger will delete error log files if the logger folder contains greater than X files, where X is an integer greater than 0 specified in the `greater_than` attribute. When the files are deleted, the oldest ones are deleted first until the folder reaches the desired file count.

Global Error Logger Configuration Example #1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration SYSTEM "../dtds/GlobalErrorLoggerConfig.dtd">
<configuration version="1.0" name="MyGlobalErrorLogger1">
      <log_details>
            <stacktraces/>
            <http_parameters length="100"/>
            <http_headers length="300"/>
```

```
      </log_details>
      <purge>
            <file_age older_than="14"/>
      </purge>
</configuration>
```

This configuration has the following features:

- Java stack traces will appear in the error logs. Note that since stack traces span multiple lines, including stack traces may complicate the process of importing the error logs into spreadsheets or databases. This is rarely done for error logs anyway.
- If there is a bad URL error message, it will include 100 characters of the URL input parameters and 300 characters of the HTTP headers, all on one line in the log file.
- Nothing is logged for a call that is put on hold.
- When a new file is added to logger instance's dedicated directory by the Global Error Logger, if the directory contains files that are older than 14 days (2 weeks), the files will be deleted.

Error Logger Configuration Example #2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration SYSTEM "../dtds/GlobalErrorLoggerConfig.dtd">
<configuration version="1.0" name="MyGlobalErrorLogger2">
      <log_details>
            <on_hold_calls application_name="true"/>
      </log_details>
      <purge>
            <file_count greater_than="100"/>
      </purge>
</configuration>
```

This configuration has the following features:

- Java stack traces will not appear in the error logs. When a Java exception occurs, only the error message itself will appear in the error log without the stack trace.
- When a call is put on hold, that fact will be logged along with the application name that the caller was attempting to visit.
- If there is a bad URL error message, only the URL itself will be logged without any HTTP parameters or headers.
- No file purging will take place. The administrator is responsible for maintaining the logs on the system.

The Global Administration History Logger

The Global Administration History Logger records administration events that occur on Call Services itself. Application-level administration history is logged by application-level loggers, which are described later in this chapter. These events are triggered by an administrator

executing administration script (see Chapter 3: Administration for more on administering Call Services).

The administration log file names begin with "admin_historyYYYY-MM-DD.txt" where YYYY, MM, and DD are the year, month, and day when the administration log was first created. By default, the log folder for is named "GlobalAdminLogger" (though the name is set in the `global_config.xml` file and can be changed by the administrator). Administration history log files are rotated daily. Note that if no administration activity occurred on a particular day, no administration history log will be created.

The file contains three columns: the time, what script was run, and its result, separated by commas. The result is usually "success" and if not, contains the description of the error encountered. The possible values of the result are:

- **server_start** - Listed when the Call Services web application archive initializes. This would occur if the Java application server on which Call Services is installed starts up or the administrator of the application server explicitly started up the Call Services web application archive.
- **server_stop** - Listed when the Call Services web application archive is stopped. This would occur if the Java application server on which Call Services is installed shuts down or the administrator of the application server explicitly stops the Call Services web application archive.
- **deploy_all_new_apps** - Listed when the `deployAllNewApps` script is run.
- **flush_all_old_apps** - Listed when the `flushAllOldApps` script is run.
- **suspend_server** - Listed when the `suspendServer` script is run.
- **resume_server** - Listed when the `resumeServer` script is run.
- **update_common_classes** - Listed when the `updateCommonClasses` script is run.

Note that running the `status` script does not trigger an administration event and thus does not update the history log.

Application Loggers

Application loggers are defined in the settings for that application. The application designer can choose any number of application loggers they wish to listen to events for a particular application, giving each instance a name. A logger may or may not require a configuration that will allow the designer to customize how the logger performs. The configuration files must be placed in the `data/application` directory of the deployed application.

Unique to application loggers is the ability for one to specify that all logging events for a call be passed to the logger it in the order in which they occurred in the call. Some application loggers

may even require this to be turned on as their functionality depends on the events arriving in order. The application designer can choose to ensure this is the case even for application loggers that do not explicitly require it to have logs appear orderly. There is some performance degradation as a result of this so an application logger that does not require this should not enable it.

Call Services provides the `logs` folder of a particular application for log file storage should the loggers require it. To keep each application logger instance's logs separate, a subfolder with the name of the instance is created and all logs created by the logger instance are stored there.

By default, Call Services includes four loggers that provide various application-specific information: an activity logger that records caller behavior, an application administration history logger that records administration activities, an error logger that lists errors that occur within calls to the application, and a debug logger that provides additional information useful when creating and debugging a new application. The activity logger and error logger require configurations that allow for detailed control over how the loggers operate.

The following sections describe these four pre-built application loggers, their configurations (if any), and the information stored in their logs.

The Application Activity Logger

The Activity Logger is the main application logger included with Call Services. It records into text log files all the activity taken by callers when they visit an application. It stores information about the call itself such as its ANI, what elements the caller encountered and in what order, and even detailed actions such as the values entered by the caller or the confidences of their utterances. The names of the log files created by the Activity Logger begin with "activity_log" and are delimited for easy importing into spreadsheets and databases. These logs have a fixed number of columns:

- **SessionID**. The session ID of the application visit as described in the Call Services Call Log section.
- **Time**. A timestamp of the event in a readable format.
- **[Element name]**. The name of the current element the activity belongs to. Only functional elements (voice elements, action elements, decision elements, and insert elements) can appear here. This column would be empty if the activity does not apply to an element.
- **Category**. The category of the action. A list of categories is given below:
  - **start**. Information on new visits to the application.
  - **end**. Information on how the application visit ended.
  - **element**. Information on the element visited and how the element was exited. The element column is empty for the `start` and `end` categories.
  - **interaction**. Detailed information about what a caller did within a voice element.

  - **data**. Element data to be logged.
  - **custom**. Custom developer-specified data to log.
- **Action**. A keyword indicating the action taken. A list of actions is given in Table 5-1.
- **Description**. Some qualifier or description of the action.

The following table lists all possible category and actions that can appear in the activity log and descriptions on what they represent.

Category	*Action*	*Description*
start	newcall or source	*newcall* is used when the application visit is a new call. The description is empty. *source* is used when another application transferred to this application. The name of the application transferred from is listed in the description.
start	ani	The description is the ANI of the caller. *NA* if the ANI is not sent.
start	areacode	The area code of the ANI. *NA* if the ANI is not sent.
start	exchange	The exchange of the ANI. *NA* if the ANI is not sent.
start	dnis	The description is the DNIS of the call. *NA* if the DNIS is not sent.
start	iidigits	The description is the IIDIGITS of the call. *NA* if the IIDIGITS is not sent.
start	uui	The description is the UUI of the call. *NA* if the UUI is not sent.
start	uid	The application visit is associated with a user. The UID is listed in the description.
start	parameter	An HTTP parameter attached to the initial URL that starts a Universal Edition application. The description lists the parameter name followed by an "=" followed by the value. A separate line will appear for each parameter passed.
start	error	An error occurred in the on call start action (either a Java class or XML-over-HTTP). The description is the error message.
end	how	How the call ended. The description is either *hangup* to indicate the caller hung up, *disconnect* to indicate the system hung up on the caller, *application_transfer:APPNAME* to indicate a transfer to another Universal Edition application occurred (where APPNAME stands for the name of the destination application), *call_transfer* to indicate a telephony blind transfer occurred, or *app_session_complete* to indicate that the call session ended via another means such as a timeout or the call being sent to an IVR system outside of Universal Edition.

end	result	The description explains why the call ended. *normal* indicates the call ended normally, *suspended* indicates the application is suspended, *error* indicates an error occurred, *timeout* indicates that the Call Services session timed out, and *invalidated* indicates the application itself invalidated the session.
end	duration	The duration of the call, in seconds.
end	error	An error occurred in the on call end action (either a Java class or XML-over-HTTP). The description is the error message.

Category	***Action***	***Description***
element	enter	The element was entered. The description is empty. This is always the first action for an element.
element	hotlink	A hotlink was activated while in the element. This can be either a global or local hotlink. The description lists the hotlink name.
element	hotevent	A hotevent was activated while in the element. The description lists the hotevent name.
element	error	An error occurred while in the element. The description lists the error message.
element	flag	A flag was triggered. The description lists the flag name.
element	exit	The element was exited. The description lists the exit state of the element or is empty if a hotlink, hotevent or error occurred within the element.
interaction	audio_group	An audio group was played to the caller. The description is the audio group name.
interaction	inputmode	How the caller entered data. The description can be *dtmf* or *speech*.
interaction	utterance	The caller said something that was matched by the speech recognition engine. The description lists the match it made of the utterance. This action will always appear with the interpretation and confidence actions.
interaction	interpretation	In a grammar, each utterance is mapped to a certain interpretation value. The description holds the interpretation value for the caller's utterance. This action will always appear with the utterance and confidence actions.
interaction	confidence	The confidence of the caller's matched utterance. This is a decimal value from 0.0 to 1.0. DTMF entries will always have a confidence of 1.0. This action will always appear with the utterance and interpretation actions.
interaction	nomatch	The caller said something that did not match anything in the grammar.
interaction	noinput	The caller did not say anything after a certain time period.
data	[NAME]	When an element creates element data, one can specify if to log the element data. Element data slated to be logged will appear here

Category	*Action*	*Description*
		with the element data name as the action and the value as the description.
custom	[NAME]	Anywhere the developer adds custom name/value information to the log will have the name appear as the action and the value stored within as the description.

Table 5-1

Notes on the Activity Logger:

- Due to its complexity, the Activity Logger requires that the enforce call event order option to be set for the logger instance using it and will throw an error if it is not set.
- When one Universal Edition application performs an application transfer to another application, the reported timestamps of the *end* category of the source application and the *start* category of the destination application could be imprecise when the source application ends with the playing of audio content. This is due to the fact that voice browsers typically request VoiceXML pages in advance if the current page contains only audio and a submit to the next page. In other words, the browser could be playing audio to the caller while making a request for the next VoiceXML page. If that page were the last of an application, the subsequent request would begin the process of entering the new application including having the Activity Logger handle start and end of call logging for the two applications. It would then report the end time for the source application as being before the time the caller actually "experienced" the destination application by hearing its audio.

The Activity Logger utilizes a configuration to control the finer details of the information it stores in its log files. The configuration controls five different aspects of the Activity Logger: the format of the files, how much data to store in them, how often to rotate the files, how caching should work, and how often should log files be purged. This configuration is specified as an XML file created by the designer in Builder for Call Studio.

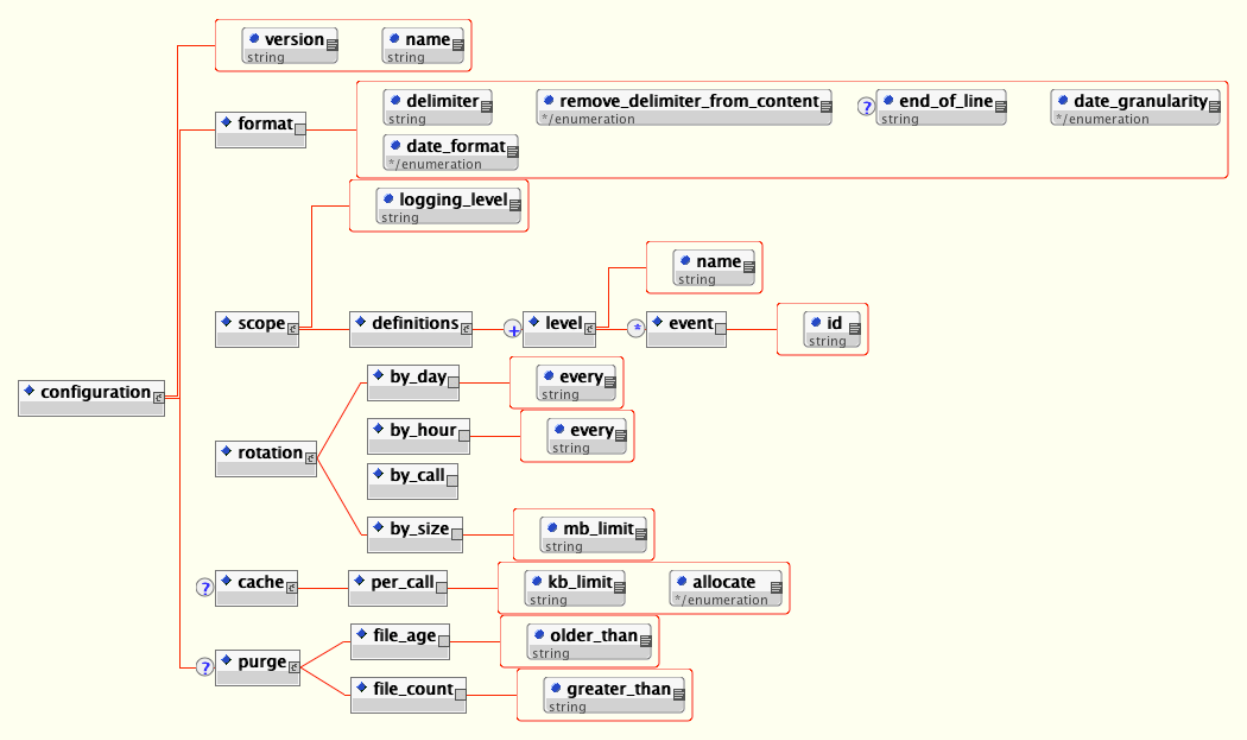


Figure 5-2

Figure 5-2 displays the format for the XML Activity Logger configuration file. The main tag for the configuration, `configuration`, has two attributes, `name` and `version`. Name is expected to contain the logger instance name though can be given any name desired. The version is expected to include the version of the configuration, which is currently "1.0". The subsequent sections describe the functionality of the various tags in the configuration.

Activity Logger Configuration: Format

The `<format>` tag allows for the modification of how the activity log files are formatted. All Activity Logger configurations are required to define a format. The possible attributes are:

- **delimiter**. This required attribute defines the delimiter to use to separate columns on a line. Delimiters can be any string value, though typically will be a comma or tab. To use a special white space character such as a new line or tab, use the escaped version. The possible values are "\n" (denoting a new line), "\t" (denoting a tab), "\r" (denoting a return), and "\f" (denoting a line feed).
- **remove_delimiter_from_content**. When this required attribute is set to `true`, the Activity Logger will attempt to eliminate the delimiter from any content appearing in the logs to ensure that the log file can be imported flawlessly. For example, if the delimiter is a comma and the configuration is set to remove the delimiter, when it is to log the content "This, is the description", it will appear in the log as "This is the description" so as not to affect the accuracy of the importing process. This extra step, though, does incur a slight performance hit. This step will not be performed if this attribute is set to `false`.

- **end_of_line**. This optional attribute controls the delimiter used to separate lines. The recommended option is to not include the attribute. In this case, the Activity Logger will separate lines appropriate to the operating system on which Call Services is running. Set the attribute to explicitly set the new line delimiter. Delimiters can be any string value, though typically will be a white space character. To use a special white space character such as a new line or tab, use the escaped version. The possible values are "\n" (denoting a new line), "\t" (denoting a tab), "\r" (denoting a return), and "\f" (denoting a line feed).
- **date_format and date_granularity**. These required attributes set how the second column of the activity log references a date when the event occurred. The format and granularity are specified. There are three possible values for the `date_format` attribute:
  - **standard**. This is a standard readable date format in the form "MM/DD/YYYY HH:MM[:SS][.MMM]" where the hour is in 24-hour time and the last three digits are the milliseconds. The seconds and milliseconds are displayed with brackets to indicate that their appearance are based on the `date_granularity` attribute. For a `date_granularity` attribute set to `minutes`, just the hours and minutes of the time will be displayed. For a granularity set to `seconds`, just the hours, minutes and seconds will be displayed. For a granularity set to `milliseconds`, all components will be displayed.
  - **minimal**. This is a minimal time value that omits the date and is in the form "HH:MM[:SS][.MMM]" where the hour is in 24-hour time and the last three digits are the milliseconds. The seconds and milliseconds are displayed with brackets to indicate that their appearance are based on the `date_granularity` attribute. For a `date_granularity` attribute set to `minutes`, just the hours and minutes will be displayed. For a granularity set to `seconds`, just the hours, minutes and seconds will be displayed. For a granularity set to `milliseconds`, all components will be displayed.
  - **number**. This displays a large integer number representing the full date and time as an elapsed time since January 1, 1970, 00:00:00 GMT. For a `date_granularity` attribute set to `minutes`, the number will be 8 digits in length (representing the number of minutes elapsed since that date). For a granularity set to `seconds`, the number will be 10 digits in length (representing the number of seconds elapsed since that date). For a granularity set to `milliseconds`, the number will be 13 digits in length (representing the number of milliseconds elapsed since that date).

Activity Logger Configuration: Scope

The Activity Logger configuration provides the administrator the ability to control what is logged based on their own needs. This is done by defining logging levels and the events that each level contains. During the debugging stage, for example, the logging level can be set to record all events and once in production, the logging level can be set to record more important events.

The `<scope>` tag defines the logging level to use in the `logging_level` attribute. The child tag `<definitions>` encapsulates all possible logging levels. All Activity Logger configurations are required to define a scope with at least one logging level.

To define a logging level, a separate `<level>` tag is added within the `<definitions>` tag and given a name in the `name` attribute. This tag will include a separate `<event>` tag for each event the logging level includes. The `id` attribute defines the name of the event. Table 5-2 lists all possible event IDs and describes when that event occurs.

Note that at minimum, the *start* and *end* events are required for any logging level as these events are used by the Activity Logger to maintain information about its log files and which calls are using them.

Event ID	***Event Description***
start	This event occurs when a new visit is made to the application (could be a new call or visit via an application transfer). *This event is required in all logging levels.*
end	This event occurs when an application visit ends. *This event is required in all logging levels.*
elementEnter	This event occurs when an element is entered. This applies to both standard and configurable elements as well as VoiceXML Insert elements.
elementExit	This event occurs when an element exits (either normally or due to something occurring within it that took the call flow elsewhere).
elementFlag	This event occurs when a flag element is visited by a caller.
defaultInteraction	This event occurs when a voice element returns interaction logging content as a result of caller activity within a VoiceXML page.
elementData	This event occurs when element data is created that has been configured to be stored in the log.
custom	This event occurs when custom content is to be added to the log, either by visiting an element whose configuration specified content to add or by executing custom code using either the Java or XML APIs that specifies to add to the log.
hotlink	This event occurs when a global or local hotlink that points to an exit state (as opposed to throwing a VoiceXML event) is activated by the caller.
hotevent	This event occurs when a hotevent with an exit state is activated in the call.
warning	This event occurs when a warning is encountered.
systemError	This event occurs when Call Services encounters an internal error (i.e. an error that does not originate from a custom component). This event will include a stack trace.
javaApiError	This event occurs when a custom component created with the Universal Edition Java API encounters an error. This event will include a stack trace.
xmlApiError	This event occurs when a custom component created with the Universal Edition XML API encounters an error. This event will *not* include a stack trace.
vxmlError	This event occurs when an error event is received from the voice browser. This event will *not* include a stack trace.

Table 5-2

Activity Logger Configuration: File Rotation

In any system that stores information in log files, high volume can make these files get very large. The desire is to have a strategy for creating new log files in order to avoid files that are too large. Additionally, file rotation strategies can help separate the log files into more logical parts. The Activity Logger defines several rotation strategies to choose from. Note that in order to ensure that the information for a single call is not split across multiple log files, the Activity Logger ensures that all call information appears in the log that was active when the call was received. As a result, it is possible for calls to be updating both pre and post rotation log files simultaneously. Each rotation strategy determines how the log files are named (though all activity log filenames begin with “activity_log”).

The `<rotation>` tag defines the rotation strategy to use by containing one of the following tags:

- **<by_day>**. This strategy will create a new log file every X days where X is an integer value greater than 0 specified in the `every` attribute. Typically this value is 1, meaning that every day at midnight, a new log file is created. For low volume systems, the value can be given a larger value. For example, when set to 7, a new log file is created once a week. The log files are named “activity_logYYYY-MM-DD.txt” where YYYY is the year, MM is the month, and DD is the day that the file is created.
- **<by_hour>**. This strategy will create a new log file every X hours where X is an integer value greater than 0 specified in the `every` attribute. There is no upper bound on this value, so it can be greater than 24. The log files are named “activity_logYYYY-MM-DD-HH.txt” where YYYY is the year, MM is the month, DD is the day, and HH is the hour that the file is created. Note that the hour is measured in 24-hour time (0 - 23).
- **<by_call>**. This strategy will create a separate log file for each call made to the application. The log files are named “activity_logYYYY-MM-DD-HH-SESSIONID.txt” where YYYY is the year, MM is the month, DD is the day, and HH is the hour that the file is created (in 24-hour time) and SESSIONID is the Call Services session ID (e.g. “activity_log2000-01-01-17-192.168.1.100.1024931901079.1.MyApp.txt”). The session ID is included in the filename to ensure uniqueness of the files. Note that care must be taken before using this log file rotation strategy on systems with high load as this will create a very large number of files.
- **<by_size>**. This strategy will create a separate log file once the previous log file has reached X megabytes in size where X is an integer value greater than 0 specified in the `mb_limit` attribute. Note that due to the fact that multiple calls will be updating the same file and that the Activity Logger will ensure that all data for a single call appear in the same log file, the final log file may be slightly larger than the limit. The log files are named “activity_logYYYY-MM-DD-HH-MM-SS.txt” where YYYY is the year, the first MM is the month, DD is the day, HH is the hour (in 24-hour time), the second MM is the minute, and SS is the second that the file is created. The time information is included in the file name in order to ensure uniqueness.

Activity Logger Configuration: Caching

The Activity Logger has the ability to use a memory cache to store information to log until either the cache fills or the call ends. Using a cache has several advantages. The first is that it increases performance by waiting until the end of the call to perform the file IO. Without a cache, the log file would be updated each time an event occurred. Another advantage is that with caching on, the log file will be more readable by grouping the activities belonging to a single phone call together. Without the cache, the events for all calls being handled by every application running on Call Services would be intermingled. While one can still sort the calls after the log is imported to a spreadsheet or database, it is much more difficult to track a single call when simply reading the log file without the cache. The one disadvantage of using a cache is that the log file is not updated in real-time, only after a call has completed. Should there be a desire to have the logs updated immediately after the events occur, then caching should be left out of the configuration.

The `<cache>` tag has only one child tag: `<per_call>`, indicating that the cache's lifetime is a single call to an application. `<per_call>` defines two attributes: `kb_limit`, an integer value greater than 0 that defines the size of the cache in kilobytes, and `allocate` that defines the cache allocation strategy. The attribute can be set to two different values:

- **once**. The Activity Logger will allocate the full memory needed for the cache once and then fill it up with logging information. When filled, the cache is flushed to the log file and the same section of memory is cleared and then refilled.
- **as_needed**. The Activity Logger will allocate memory as events arrive in the call until the total amount of memory has been allocated. When it is to be flushed, the memory is released and then the allocation begins again.

The advantage of allocating the memory at once is that since a contiguous section of memory is being used, the updating, maintenance, and flushing of that memory will be slightly faster. Additionally, with only one area of memory per call less memory allocations take place, which can affect how often Java garbage collection runs. A disadvantage is that the cache size should be chosen carefully. Too small a cache will incur performance hits as the cache fills up and is logged multiple times within a call. Too large a cache would mean that a large amount of memory is allocated and then never used, potentially starving the rest of the system. A good cache size would be approximately the size of a log for a typical call to the application. Since the cache is flushed at the end of a call, there is little reason to make the cache much larger.

The advantage to allocating the memory as needed is that this minimizes the memory used since only the memory needed to store the information is used. The cache size is not as important, and making it larger will not affect the overall memory usage as drastically as if the cache was allocated all at once since the memory would not be allocated unless needed.

It is recommended to configure the cache to be allocated once for performance and as needed if memory on the system is tight.

Activity Logger Configuration: File Purging

The Activity Logger can be configured to automatically delete files that conform to certain criteria. Properly configured, this will allow an administrator to avoid having the system's hard drive fill up with logs, thereby preventing new calls from being logged. A few notes about file purging that must be understood:

- A logger has control only over the files appearing under the logger instance's dedicated log folder and cannot control those files managed by other logger instances. This even applies to multiple instances of the same logger since each logger instance is given its own unique folder within the `logs` folder of the application. Activity Logger file purging therefore applies only to those files appearing under the logger instance's folder.
- Since loggers are activated only when events occur in a call, the file purging activity will only take place when a call ends. As a result, a system that receives no calls at all will not automatically delete files until a new call is received and completes.
- When the Activity Logger starts up for the first time, it will apply the purging strategy on any files that exist in the logger directory. Therefore, if an application server is shut down with files in the logger directory and then restarted a long time later, these files could be deleted when the application server starts up and the logger initializes. This applies to any file appearing in the logger directory, not just activity logs.
- The Activity Logger keeps information about the activity log files in memory and acts on that to determine whether to delete them rather than by monitoring the remaining hard drive space on the system. This is done to avoid having to do file IO to determine if a file is to be purged and so minimizes overhead (though there still is overhead in simply deleting files). One consequence of this is that the logger keeps track only of those files it is managing. The logger is unaware of any files added to the directory after the application server initializes. So the purging strategy will affect those files only.

The optional `<purge>` tag defines the purging strategy. If this tag does not appear in the configuration, no file purging will take place. The tag can contain one of the following child tags:

- **file_age**. The Activity Logger will delete activity log files older than X days, where X is an integer greater than 0 specified in the `older_than` attribute.
- **file_count**. The Activity Logger will delete activity log files if the logger folder contains greater than X files, where X is an integer greater than 0 specified in the `greater_than` attribute. When the files are deleted, the oldest ones are deleted first until the folder reaches the desired file count.

Activity Logger Configuration Example #1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration SYSTEM "../../../../dtds/ActivityLoggerConfig.dtd">
<configuration version="1.0" name="MyLogger1">
      <format delimiter="\t" remove_delimiter_from_content="true"
end_of_line="\n" date_format="standard" date_granularity="milliseconds"/>
      <scope logging_level="Complete">
            <definitions>
                  <level name="Minimal">
                        <event id="start"/>
                        <event id="end"/>
                  </level>
                  <level name="Complete">
                        <event id="start"/>
                        <event id="end"/>
                        <event id="elementEnter"/>
                        <event id="elementExit"/>
                        <event id="elementFlag"/>
                        <event id="defaultInteraction"/>
                        <event id="elementData"/>
                        <event id="custom"/>
                        <event id="hotlink"/>
                        <event id="hotevent"/>
                        <event id="warning"/>
                  </level>
            </definitions>
      </scope>
      <rotation>
            <by_day every="2"/>
      </rotation>
      <cache>
            <per_call kb_limit="10" allocate="once"/>
      </cache>
      <purge>
            <file_age older_than="3"/>
      </purge>
</configuration>
```

This configuration has the following features:

- The activity logs will be delimited with a tab (“\t”) and will have any tabs that appear in the content removed.
- The activity logs will use a Unix-style new line character (“\n”) to delimit lines. As a result, these log files would not appear orderly on Windows Notepad because it does not recognize these new line characters.
- Dates in the activity logs will appear in the standard format with millisecond granularity. For example: “05/09/2006 15:45:02.654”
- Two logging levels are defined: Minimal, which logs only when a caller entered and exited an application, and Complete, which logs all events. The Complete logging level is the one that will be used.
- The activity log files will be rotated every two days, meaning each log file will contain 2 days worth of calls before a new file is created.

- The cache is set to 10K or 5000 characters and is allocated once at the start of a call.
- Files that are older than 3 days that appear in the logger instance's dedicated directory will be purged.

Activity Logger Configuration Example #2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration SYSTEM "../../../../dtds/ActivityLoggerConfig.dtd">
<configuration version="1.0" name="MyLogger2">
      <format delimiter="," remove_delimiter_from_content="false"
date_format="minimal" date_granularity="minutes"/>
      <scope logging_level="MyLoggingLevel">
            <definitions>
                  <level name="MyLoggingLevel">
                        <event id="start"/>
                        <event id="end"/>
                        <event id="elementEnter"/>
                        <event id="elementFlag"/>
                        <event id="elementExit"/>
                  </level>
            </definitions>
      </scope>
      <rotation>
            <by_size mb_limit="100"/>
      </rotation>
</configuration>
```

This configuration has the following features:

- The activity logs will be delimited with a comma and will not remove any commas that appear in the content potentially complicating any importing of these logs into spreadsheets or databases.
- The activity logs will end each line with the character appropriate for the operating system on which it is generated. So if this system is running under Windows, the activity logs will look fine under Notepad and if this system is running under Unix, the activity logs will use the Unix end of line characters that would not be recognized if opened by Windows Notepad.
- Dates in the activity logs will appear in the minimal format with minute granularity. For example: "15:45".
- Only one logging level is defined that logs when calls enter and exit an application, enter and exit an element, and when a flag element is visited.
- A new activity log is created when the previous one has reached approximately 100MB in size, regardless on whether the calls spanned weeks or hours.
- No logging cache is used, meaning that when a logging event occurs in a call, it is placed into the activity log immediately. This allows for real-time logging but incurs a performance overhead in managing much more IO operations.
- No file purging will take place. The administrator is responsible for maintaining the logs on the system.

The Application Error Logger

During the voice application development process, errors can be introduced by configuring elements incorrectly, spelling mistakes in audio filenames, or by Java coding bugs. In each of these cases errors occur while running the application. While the Activity Logger does report errors, it is preferable to isolate errors in a separate file so that they are easily found and dealt with. Additionally, when reporting Java errors, a stack trace is desired. The application Error Logger provides a place for these errors to appear. The error log file names are in the form "error_logYYYY-MM-DD.txt" where YYYY, MM, and DD are the year, month, and day when the error log was first created and is rotated daily.

Note that the application Error Logger will report information on errors that are affiliated with the application in which it is configured. It can even report errors encountered by other loggers in the same application only if the Error Logger is listed *before* other loggers in the application. If another logger is loaded before the Error Logger, any errors it encounters will be logged instead to the Call Services Call Error Log. It is for this reason that by default Builder for Call Studio puts the Error Logger at the top of the list of loggers to use for a new application.

The columns of the error log are:

- **SessionID**. The session ID of the application visit described in the Call Services Call Log section.
- **Time**. The time the error occurred.
- **Description**. The error description including a Java stack trace if applicable.

The Error Logger utilizes a configuration to control two different aspects of the error logs: the format of the files and how often should log files be purged. This configuration is specified as an XML file created by the designer in Builder for Call Studio.

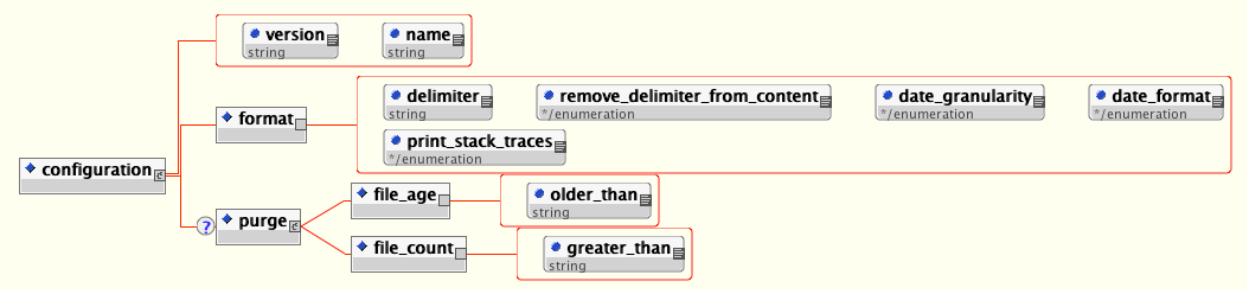


Figure 5-3

Figure 5-3 displays the format for the XML Error Logger configuration file. The main tag for the configuration, `configuration`, has two attributes, `name` and `version`. Name is expected to contain the logger instance name. The version is expected to include the version of the configuration, which is currently "1.0". The subsequent sections describe the functionality of the various tags in the configuration.

Error Logger Configuration: Format

The `<format>` tag allows for the modification of how the error log files are formatted. All Error Logger configurations are required to define a format. The possible attributes are:

- **delimiter**. This required attribute defines the delimiter to use to separate columns on a line. Delimiters can be any string value, though typically will be a comma or tab. To use a special white space character such as a new line or tab, use the escaped version. The possible values are "\n" (denoting a new line), "\t" (denoting a tab), "\r" (denoting a return), and "\f" (denoting a line feed).
- **remove_delimiter_from_content**. When this required attribute is set to `true`, the Error Logger will attempt to eliminate the delimiter from any content appearing in the logs to ensure that the log file can be imported flawlessly. For example, if the delimiter is a comma and the configuration is set to remove the delimiter, when it is to log the content "This, is the description", it will appear in the log as "This is the description" so as not to affect the accuracy of the importing process. This extra step, though, does incur a slight performance hit. This step will not be performed if this attribute is set to `false`. Note that should the error log contain Java stack traces, the error logs could be difficult to import as stack traces fill multiple lines (though their content will be cleaned of the delimiter if desired).
- **date_format and date_granularity**. These required attributes set how the second column of the error log references a date when the event occurred. The format and granularity are specified. There are three possible values for the `date_format` attribute:
  - **standard**. This is a standard readable date format in the form "MM/DD/YYYY HH:MM[:SS][.MMM]" where the hour is in 24-hour time and the last three digits are the milliseconds. The seconds and milliseconds are displayed with brackets to indicate that their appearance are based on the `date_granularity` attribute. For a `date_granularity` attribute set to `minutes`, just the hours and minutes of the time will be displayed. For a granularity set to `seconds`, just the hours, minutes and seconds will be displayed. For a granularity set to `milliseconds`, all components will be displayed.
  - **minimal**. This is a minimal time value that omits the date and is in the form "HH:MM[:SS][.MMM]" where the hour is in 24-hour time and the last three digits are the milliseconds. The seconds and milliseconds are displayed with brackets to indicate that their appearance are based on the `date_granularity` attribute. For a `date_granularity` attribute set to `minutes`, just the hours and minutes will be displayed. For a granularity set to `seconds`, just the hours, minutes and seconds will be displayed. For a granularity set to `milliseconds`, all components will be displayed.
  - **number**. This displays a large integer number representing the full date and time as an elapsed time since January 1, 1970, 00:00:00 GMT. For a `date_granularity` attribute set to `minutes`, the number will be 8 digits in length (representing the number of minutes elapsed since that date). For a granularity set to `seconds`, the number will be 10 digits in length (representing the number of seconds elapsed since that date). For a granularity set

to `milliseconds`, the number will be 13 digits in length (representing the number of milliseconds elapsed since that date).

- **print_stack_traces**. This required attribute is set to either `true` or `false` and determines whether the error log will contain Java stack traces. Stack traces are very useful to a developer in tracking down the cause of a Java error so it is recommended to keep this option on.

Error Logger Configuration: File Purging

The Error Logger can be configured to automatically delete files that conform to certain criteria. Properly configured, this will allow an administrator to avoid having the system's hard drive fill up with logs, which would prevent new calls from being logged.

A few notes about file purging must be given:

- Since loggers are activated only when events occur in a call, the file purging activity will only take place when an error event occurs. As a result, a system that encounters no errors will not automatically delete files until a new error occurs.
- When the Error Logger starts up for the first time, it will apply the purging strategy on any files that exist in the logger directory. Therefore, if an application server is shut down with files in the logger directory and then restarted a long time later, these files could be deleted when the application server starts up and the logger initializes.
- Unlike the Activity Logger, the Error Logger applies its purging strategy to any files found in its logger directory, including non-error log files. So should other files be added to the logger folder after the application server has started could be deleted when the Error Logger encounters a new error.

The optional `<purge>` tag defines the purging strategy. If this tag does not appear in the configuration, no file purging will take place. The tag can contain one of the following child tags:

- **file_age**. The Error Logger will delete error log files older than X days, where X is an integer greater than 0 specified in the `older_than` attribute.
- **file_count**. The Error Logger will delete error log files if the logger folder contains greater than X files, where X is an integer greater than 0 specified in the `greater_than` attribute. When the files are deleted, the oldest ones are deleted first until the folder reaches the desired file count.

Error Logger Configuration Example #1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration SYSTEM
"../../../../dtds/ApplicationErrorLoggerConfig.dtd">
<configuration version="1.0" name="MyErrorLogger1">
```

```
      <format delimiter="," remove_delimiter_from_content="true"
date_format="standard" date_granularity="seconds" print_stack_traces="true"/>
      <purge>
            <file_count greater_than="10"/>
      </purge>
</configuration>
```

This configuration has the following features:

- The error logs will be delimited with a comma and will have any commas that appear in the content removed.
- Dates in the error logs will appear in the standard format with seconds granularity. For example: “05/09/2006 15:45:02”
- Java stack traces will appear in the error logs. Note that since stack traces span multiple lines, including stack traces may complicate the process of importing the error logs into spreadsheets or databases. This is rarely done for error logs anyway.
- When a new file is added to logger instance’s dedicated directory by the Error Logger, if the directory contains more than 10 files the oldest file will be deleted.

Error Logger Configuration Example #2

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration SYSTEM
"../../../../dtds/ApplicationErrorLoggerConfig.dtd">
<configuration version="1.0" name="MyErrorLogger2">
      <format delimiter="***" remove_delimiter_from_content="false"
date_format="minimal" date_granularity="seconds" print_stack_traces="false"/>
</configuration>
```

This configuration has the following features:

- The error logs will be delimited with the string “***” and will not attempt to remove that string from the content. Note that the delimiter does not need to be limited to a single character and can be a multi-character string. Usually, though, it is a single character to make importing into spreadsheets and databases straightforward.
- Dates in the error logs will appear in the minimal format with seconds granularity. For example: “15:45:02”
- Java stack traces will not appear in the error logs. When a Java exception occurs, only the error message itself will appear in the error log without the stack trace.
- No file purging will take place. The administrator is responsible for maintaining the logs on the system.

The Application Administration History Logger

Whenever an application-specific administration script is run, a log file is updated with information on the script that was run. The administration log file names are in the form

"admin_historyYYYY-MM-DD.txt" where YYYY, MM, and DD are the year, month, and day when the administration history log was first created and is rotated daily. The file contains three columns: the time the script was run, what script was run, and its result. The result is usually "success" and if not contains the description of the error encountered. The possible values are:

- **server_start** - Each application's history log contains records of each time the application server starts.
- **deploy_app** - Listed when the `deployApp` script is run.
- **suspend_app** - Listed when the `suspendApp` script is run.
- **resume_app** - Listed when the `resumeApp` script is run.
- **update_app** - Listed when the `updateApp` script is run.
- **release_app** - Listed when the `releaseApp` script is run.
- **update_common_classes** – Listed when the global `updateCommonClasses` script is run. The reason this global admin event is logged by the Application Administration History Logger is because elements that appear in the `common` directory are reloaded by this command, prompting those elements to reload their application-specific configurations.

Running the `status` script does not update the history log.

The Administration History Logger does not utilize a configuration.

The Application Debug Logger

At times when debugging an application it would be advantageous to see information concerning the HTTP requests made by the voice browser and the corresponding HTTP responses of Call Services. The Debug Logger creates a single file per call that contains all HTTP requests and responses that occurred within that call session. The log files are named "debug_logYYYY-MM-DD-HH-SESSIONID.txt" where YYYY is the year, MM is the month, DD is the day, and HH is the hour (in 24-hour time) that the file is created and SESSIONID is the Call Services session ID (e.g. "debug_log2000-01-01-17-192.168.1.100.1024931901079.1.MyApp.txt"). The session ID is included in the filename to ensure uniqueness of the files. The debug log contains:

- A timestamp of when each HTTP request was received from the voice browser as well as when the response was sent back by Call Services.
- All headers of the HTTP request.
- All arguments passed with the HTTP request, whether they be set via GET or POST.
- The entire VoiceXML page returned in the HTTP response.
- It is recommended to use the Debug Logger only when performing debugging and not in a production environment as it incurs overhead on the system in creating and managing file IO and replicating the HTTP response, which must be generated once for the voice browser and

once for each Debug Logger instance. Note the Debug Logger does not require the enforce call event order to be turned on, however without it there could be situations where under load the HTTP requests and responses are out of order or mixed together in the file.

Chapter 6: Call Services Configuration

Call Services can be configured to tailor its behavior. This chapter explains all configuration options and how to change them.

Out of the box, Call Services uses default values for these configuration options and will function without modification. Only an experienced administrator should consider changing these options as improperly chosen values can cause significant performance degradations and could even prevent Call Services from functioning correctly.

Global Configuration File

The mechanism to edit the Call Services configuration is through an XML file named `global_config.xml` found in the `AUDIUM_HOME/conf` directory. This file must be edited by hand, there is no graphical interface.

This file is loaded by Call Services when it is initialized and cached in memory. Loading the file is one of the first tasks performed by Call Services when it starts up since the configuration options affect how it runs. Any changes to this file will require Call Services to be restarted in order for the changes to take affect.

Note that when performing an upgrade of Call Services, the administrator will have to re-implement the configuration changes.

Configuration Options

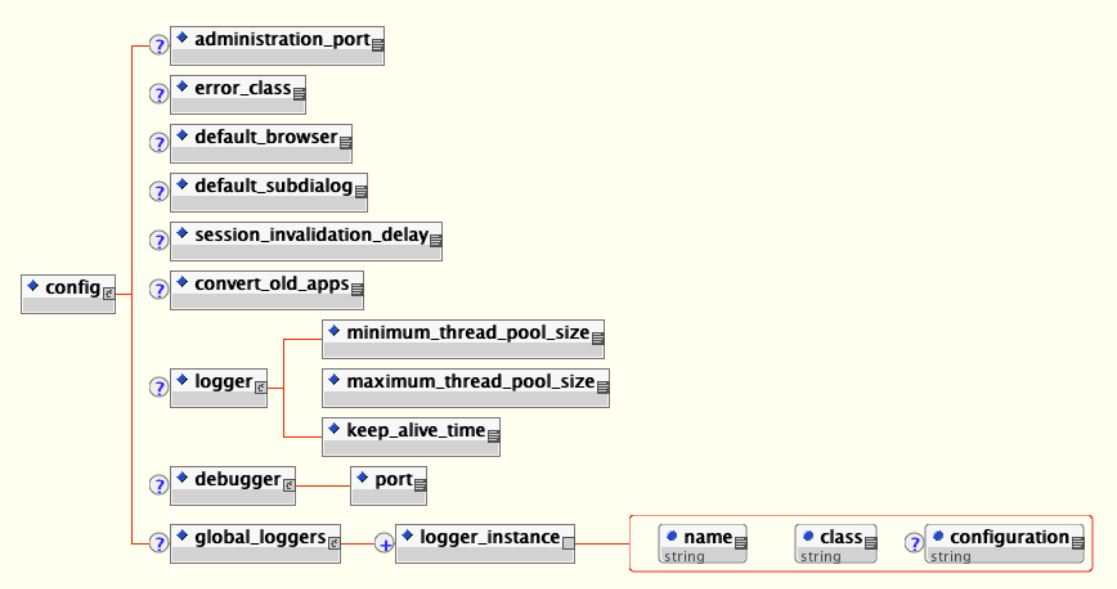


Figure 6-1

Figure 6-1 displays the DTD diagram of the `global_config.xml` file. The elements in the XML document are:

- **administration_port** – This tag defines the port on which administration activity takes place and can be any positive integer. By default, the port is set to 10100. See Chapter 3: Administration for more on administration activities.
- **error_class** – This tag defines the fully qualified Java class name of a class to execute when an error occurs for notification purposes. By default no class is defined. See the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more on how to write the On Error Notification class.
- **default_browser** – This tag defines the real name of the gateway adapter that should be used by default when Call Services needs to produce a VoiceXML page in a scenario where the current application is unknown and therefore the gateway adapter for that application is unknown. One such scenario is an error where the Call Services session is unrecognized. The reason this exists is because some gateways require the VoiceXML to be formatted in a specific way (such as requiring an XML namespace to appear in the document) that if the VoiceXML page were produced in a different format would cause an error on the gateway. An application lists its gateway in its settings and normally this is available to Call Services to produce the correct VoiceXML. However in rare cases, an error occurs and Call Services does not have access to the session and hence the application that the call belonged to and would need to know which gateway to have the resulting VoiceXML page conform to. By default, if left blank in `global_config.xml`, Call Services will search through the directory of installed gateway adapters and use the first one it finds.
- **default_subdialog** – This tag defines whether to treat a call that is not associated with an application as if it were a VoiceXML subdialog and whose possible values are `true` and `false`. Some gateways (such as Cisco gateways) call all Call Services applications as VoiceXML subdialogs. Call Services must be aware of this because it determines how the VoiceXML it produces looks and if not produced correctly would cause an error on the gateway. Typically, a call is made to an application which defines in its settings whether to treat the application as a subdialog. However in rare cases, an error occurs and Call Services does not have access to the session and hence the application that the call belonged to and would need to know whether to treat the call as a subdialog. By default, if left blank in `global_config.xml`, Call Services will consider a call to the application to ***not*** be a subdialog.
- **session_invalidation_delay** – This tag defines the amount of time in seconds that Call Services will wait for after a call session ends before actually invalidating that session (this can be any integer greater than or equal to 0). This configuration option is needed because there may be various activities taken by loggers and end of call classes that require the session to remain alive to access data within it (such as element or session data) and if the session were invalidated would cause errors to occur when attempting to access the data. If this value were too small (such as 0 seconds), many errors could occur for routine actions like logging at the end of a call. If this value were too high, too many sessions would remain

in memory for too long, potentially causing memory issues. It is therefore highly recommended to keep the default value of 30 seconds or to test the system should this value be changed.

- **convert_old_apps** – This tag defines whether to convert applications deployed from a version of Call Studio that Call Services detects is old (possible values are `true` and `false`). By setting this configuration option to `true`, a deployed application can be copied to the `applications` directory of Call Services without requiring the application to be re-deployed from the latest version of Call Studio. Note that for new application settings, the converter will choose default values. Also note that this converter is limited to converting the XML files that define an application with regards to Call Studio and Call Services and will ***not*** convert any other files or Java classes for the application. By default this configuration option is on.

- **logger** – This tag acts as the parent tag for three additional tags having to do with loggers. The first two tags, `<minimum_thread_pool_size>` and `<maximum_thread_pool_size>` define the minimum and maximum size of the thread pool that is used for handling logger threads. The minimum thread pool size value can be any positive integer and the maximum thread pool size value can be any positive number as long as it is greater than the minimum thread pool size value. Note that if the maximum number of logger threads are used, Call Services will queue the logger events to be used when a thread becomes available so the data will not be lost. Since these values affect thread usage, it is highly recommended that any deviation from the default values (1 minimum / 500 maximum) be fully tested for any complications. For example, if the maximum is set to a low value and the system encounters high load, Call Services could encounter a situation where the queued logger events accumulate faster than the logger threads can handle them, *leading to a scenario where the application server **runs out of memory***. On the other hand, if the maximum value were set too high and the system encounters high load the system on which Call Services runs could run out of threads to allocate, which could cause many other problems with the application server as well as the operating system itself. Of all the Call Services configuration options, these two have the highest potential for causing major problems if misused.
  The third child tag, `<keep_alive_time>` defines the amount of time in seconds that a thread should be idle for before it is removed from the thread pool. This allows for the thread pool size to shrink over time as logger volume decreases. This value allows for optimum thread pool size based on the call volume. The default value is 30 seconds. It is recommended not to change greatly from the default as too high a number will keep unnecessary resources around and too low a number will reduce efficiency and defeat the purpose of using a thread pool completely. Refer to Chapter 5: Call Services Logging for more on logging.

- **debugger** – This tag defines the RMI registry port for the Call Studio debugger. This configuration option is used only by Call Services implementations used by Call Studio for debugging purposes and should not be used in a production environment. The default is 8099 and the value can be any positive integer.

- **global_loggers** – This tag defines the global loggers to use within Call Services. Administrators can add additional global loggers as well as change or remove the loggers

listed by default: the global call, admin, and error loggers. Each logger instance is defined by a separate child tag `<logger_instance>`. The required `name` attribute gives the logger instance a name and must be unique. The required `class` attribute gives the fully qualified Java class name that defines the global logger. The optional `configuration` attribute gives the name of a configuration file for the global logger if needed. This configuration file is expected to reside in the same `AUDIUM_HOME/conf` directory. Refer to Chapter 5: Call Services Logging for more on logging and the Programming Guide for Cisco Unified Call Services, Universal Edition and Unified Call Studio for more on creating custom loggers.

Chapter 7: Standalone Application Builder

Normally a designer builds an application in Call Studio and then deploy to Call Services. Call Studio has the ability to deploy an application locally as well as to a remote system via FTP. Deploying an application becomes more difficult in an environment where many designers are working on a single application or when the enterprise follows a strict deployment policy to the runtime servers. In the first scenario, multiple designers are adding content to a source repository system and no single designer may have the full application in necessary to perform the deployment and even if possible, would require coordination among all designers involved. In the second scenario, the production environments do not allow direct access via FTP and require an automated system to place new content on to those environments, providing the flexibility to control exactly how and when the content is deployed. The desire is to extract the ability to create a Call Services application from the Call Studio project without requiring a person to launch Call Studio and deploy.

Universal Edition provides a tool to support this requirement named the Standalone Application Builder. It is packaged with Call Studio and allows for the deployment of an application through a command-line interface. By exposing this as a command-line tool, an administrator can integrate this tool into any process that has the ability to execute scripts. For example, the administrator can configure a crontab to launch this utility every day with the latest content checked into a source repository. Another example is to modify existing build and deploy Ant scripts to deploy the application once all other components such as elements, grammars, etc. are assembled.

This chapter explains what the Standalone Application Builder does and how to use it.

Standalone Application Builder Introduction

The Standalone Application Builder is a utility that deploys a studio application project to a format that is required by Call Services. It is launched via a batch script (for Windows) named `buildApp.bat` or shell script (for Linux) named `buildApp.sh`.

The Standalone Application Builder is bundled with Call Studio as a ZIP file in the folder XXX. The archive can be unzipped on to any location and is completely independent of Universal Edition software Additionally, there is no license required to use the utility. Only the following 32-bit operating systems are supported: Microsoft Windows XP, Microsoft Windows Vista, and Red Hat Enterprise Linux WS 4 for x86.

When launched, the Standalone Application Builder will first validate the Call Studio project to ensure it is a valid application, and if successful, deploys the Call Services version of the application to the destination folder. If there are validation errors, those errors are displayed in the output similar to validation errors that are displayed in Call Studio. The tool only deploys a single application at a time. To deploy multiple applications, the script can be called repeatedly pointing to different projects.

Script Execution

The command-line usage of the Standalone Application Builder is as follows:

```
buildApp <project path> <deploy path> [-quiet <log file>] [-debug]
```

- **<project path>** represents the directory in which the Call Studio project to convert resides. This path should point to the location where Call Studio is configured to store application projects. By default this is the `workspace` folder within the `eclipse` folder.
- **<deploy path>** represents the directory to deploy the application to. If the Standalone Application Builder is installed on the same machine as Call Services, one can pass the <Audium_Home>/`CallServices/applications` of Audium Home so that the application is deployed directly to the Call Services instance. All that would be needed to make the application live would be to call the `deployApp` Call Services administration script.
- **-quiet <log file>** is an optional parameter that is designed to pipe the output the script usually produces into a text file whose name is `<log file>`. This is useful for scenarios where the Standalone Application Builder is executed from an automated system that does not display data printed to the console. By piping the data to a file, any results can be analyzed later.
- **-debug** is an optional parameter that produces additional output to use for debugging purposes should the deployment fail. This option should not be used unless directed to by customer support.

Script Output

The following is how the output of the Standalone Application Builder will look for a successful deployment:

```
Cisco Unified Call Studio 6.0 (Standalone Application Build Mode)
© 1999-2007 Cisco Systems, Inc.
All rights reserved. Cisco, the Cisco logo, Cisco Systems, and the Cisco Systems
logo are trademarks or registered trademarks of Cisco Systems, Inc. and/or its
affiliates in the United States and certain other countries.

Start: Tue Jan 1 11:47:56 EDT 2000
*** Loading project.
*** Validating project ‘MyApp’.
*** Building project ‘MyApp’.
*** Unloading project ‘MyApp’.
*** Done.
End: Tue Jan 1 11:47:58 EDT 2000
```

The following is the output for a deployment that encounters validation errors:

```
Cisco Unified Call Studio 6.0 (Standalone Application Build Mode)
© 1999-2007 Cisco Systems, Inc.
All rights reserved. Cisco, the Cisco logo, Cisco Systems, and the Cisco Systems
logo are trademarks or registered trademarks of Cisco Systems, Inc. and/or its
affiliates in the United States and certain other countries.

Start: Tue Jan 1 11:47:56 EDT 2000
*** Loading project.
*** Validating project ‘MyApp’.
Error: Project is not valid. Aborting. See details below:
[Start Of Call] Exit States Error: Please connect all the exit states for this
element.
```

Appendix A: Substitution Tag Reference

The following table lists the contents of tags used for setting value substitution. To represent each of the data values, the tag is rendered with braces containing the tag content listed below, case sensitive. The fragments rendered in underlined green represent values replaced by the application designer. Optional information is encapsulated in brackets ([]).

Tag Content	Description
CallData.ANI	The ANI of the current call or “NA” if not sent.
CallData.DNIS	The DNIS of the current call or “NA” if not sent.
CallData.UUI	The UUI of the current call or “NA” if not sent.
CallData.IIDIGITS	The IIDIGITS of the current call or “NA” if not sent.
CallData.SOURCE	The name of the application that transferred to this one.
CallData.APP_NAME	The name of the current application.
CallData.DURATION	The duration, in seconds, of the call up to this point.
CallData.LANGUAGE	The VoiceXML encoding for the application, up to this point in the call.
CallData.ENCODING	The language for the application, up to this point in the call.
Data.Session.VAR	The value of Session Data where VAR represents the name of the session variable. The object stored there will be represented as a string.
Data.Element.ELEMENT.VAR	The value of Element Data where ELEMENT represents the name of the element and VAR represents the name of the element variable.
CallerActivity.NthElement.N	The name of a certain element visited in the call where N represents the number for the nth element.
CallerActivity.NthExitState.N	The name of a certain element’s exit state visited in the call where N represents the

	number for the nth element.
CallerActivity.TimesElementVisited.ELEMENT	The number of times an element was visited in the call where ELEMENT represents the name of the element.
Tag Content	Description
CallerActivity.TimesElementVisitedExitState.ELEMENT.EXIT_STATE	The number of times an element was visited in the call with a particular exit state where ELEMENT is the name of the element and EXIT_STATE is the exit state.
GeneralDateTime.HourOfDay.CURRENT	The current hour.
GeneralDateTime.HourOfDay.CALL_START	The hour the call started.
GeneralDateTime.Minute.CURRENT	The current minute.
GeneralDateTime.Minute.CALL_START	The minute the call started.
GeneralDateTime.DayOfMonth.CURRENT	The current day of the month.
GeneralDateTime.DayOfMonth.CALL_START	The day of the month the call started.
GeneralDateTime.Month.CURRENT	The current month.
GeneralDateTime.Month.CALL_START	The month the call started.
GeneralDateTime.DayOfWeek.CURRENT	The current day of the week.
GeneralDateTime.DayOfWeek.CALL_START	The day of the week the call started.
GeneralDateTime.Year.CURRENT	The current year.
GeneralDateTime.Year.CALL_START	The year the call started.

The following tags will cause an error if the User Management System is inactive. Additionally, these tags relate to the current user and will cause an error unless the call is linked to a UID.

Tag Content	Description
UserInfo.Demographic.NAME	The name of the current user.
UserInfo.Demographic.ZIP_CODE	The zip code of the current user.
UserInfo.Demographic.BIRTHDAY	The birthday of the current user.
UserInfo.Demographic.GENDER	The gender of the current user.
UserInfo.Demographic.SSN	The social security number of the current user.
UserInfo.Demographic.COUNTRY	The country of the current user.
UserInfo.Demographic.LANGUAGE	The language of the current user.

Tag Content	Description
UserInfo.Demographic.CUSTOM1	The contents of the first custom column of the current user.
UserInfo.Demographic.CUSTOM2	The contents of the second custom column of the current user.
UserInfo.Demographic.CUSTOM3	The contents of the third custom column of the current user.
UserInfo.Demographic.CUSTOM4	The contents of the fourth custom column of the current user.
UserInfo.AniInfo.FIRST	The first phone number associated with the current user's account.
UserInfo.AniInfo.NUM_DIFF	The total number of different phone numbers associated with the current user's account.
UserInfo.UserDateTime.HourOfDay.LAST_MODIFIED	The hour of the last time the current user's account was modified.
UserInfo.UserDateTime.HourOfDay.CREATION	The hour of the last time the current user's account was created.
UserInfo.UserDateTime.HourOfDay.LAST_CALL	The hour of the last time the current user called.
UserInfo.UserDateTime.Minute.LAST_MODIFIED	The minute of the last time the current user's account was modified.
UserInfo.UserDateTime.Minute.CREATION	The minute of the last time the current user's account was created.
UserInfo.UserDateTime.Minute.LAST_CALL	The minute of the last time the current user called.
UserInfo.UserDateTime.DayOfMonth.LAST_MODIFIED	The day of the month of the last time the current user's account was modified.
UserInfo.UserDateTime.DayOfMonth.CREATION	The day of the month of the last time the current user's account was created.
UserInfo.UserDateTime.DayOfMonth.LAST_CALL	The day of the month of the last time the current user called.
UserInfo.UserDateTime.Month.LAST_MODIFIED	The month of the last time the current user's account was modified.

Tag Content	Description
UserInfo.UserDateTime.Month.CREATION	The month of the last time the current user's account was created.
UserInfo.UserDateTime.Month.LAST_CALL	The month of the last time the current user called.
UserInfo.UserDateTime.DayOfWeek. LAST_MODIFIED	The day of the week of the last time the current user's account was modified.
UserInfo.UserDateTime.DayOfWeek. CREATION	The day of the week of the last time the current user's account was created.
UserInfo.UserDateTime.DayOfWeek. LAST_CALL	The day of the week of the last time the current user called.
UserInfo.UserDateTime.Year. LAST_MODIFIED	The year of the last time the current user's account was modified.
UserInfo.UserDateTime.Year.CREATION	The year of the last time the current user's account was created.
UserInfo.UserDateTime.Year.LAST_CALL	The year of the last time the current user called.
UserInfo.CalledFromAni	"true" if the current user has made calls from the current phone or "false" if not.
UserInfo.AccountInfo.PIN	The PIN number of the current user's account.
UserInfo.AccountInfo.ACCOUNT_NUMBER	The account number of the current user's account.
UserInfo.AccountInfo.EXTERNAL_UID	The external UID of the current user's account.

These tags relate to historical data. While still requiring the User Management System to be active, these do not require a user to be associated with the call.

Tag Content	Description
GeneralAniInfo.AniDateTime.HourOfDay. LAST_CALL[.ANI]	The hour of the last time a call was received from the current phone number. Use ANI to get the last time a call was received from another number where ANI is the number.
GeneralAniInfo.AniDateTime.Minute. LAST_CALL[.ANI]	The minute of the last time a call was received from the current phone number or ANI if specified.

Tag Content	Description
GeneralAniInfo.AniDateTime.DayOfMonth.LAST_CALL[.ANI]	The day of the month of the last time a call was received from the current phone number or ANI if specified.
GeneralAniInfo.AniDateTime.Month.LAST_CALL[.ANI]	The month of the last time a call was received from the current phone number or ANI if specified.
GeneralAniInfo.AniDateTime.DayOfWeek.LAST_CALL[.ANI]	The day of the week of the last time a call was received from the current phone number or ANI if specified.
GeneralAniInfo.AniDateTime.Year.LAST_CALL[.ANI]	The year of the last time a call was received from the current phone number or ANI if specified.
GeneralAniInfo.AniNumCalls[.ANI]	The number of calls received from the current phone number or ANI if specified.

Notes:

- Each Date / Time tag evaluates to 0-23 when referring to the hour, 0-59 when referring to the minute, 1-12 when referring to the month, 1-31 when referring to the day of the month, 1-7 when referring to the day of the week (where 1 is Sunday), and the year is represented as a four-digit number.
- If any data represented by the tag ends up as null, substitution will render it as an empty string. For example, if a setting contained “source{CallData.SOURCE}” and there was no application that transferred to the current application, the setting would be evaluated as “source”. In this case, a warning appears in the Error Log for the application noting that a substitution value was null and was replaced with an empty string.

Appendix B: The Directory Structure

The directory in which the installation is made (referred to as the INSTALLATION_PATH directory) contains all the files necessary for the various components of Universal Edition software. The following table describes what each folder in the INSTALLATION_PATH is used for. Each folder is described in detail in subsequent sections.

Folder	Description
CallServices	This directory contains the files required for Call Services to run, including all voice applications.
CallStudio	This directory contains Call Studio and Builder for Call Studio
UninstallerData	The application inside this folder is used to uninstall Universal Edition software.

The `INSTALLATION_PATH\CallServices` folder (also referred to as the Audium Home directory) contains the following folders:

Folder	Description
admin	This directory holds the scripts that perform administrator functions affecting all applications on Call Services.
admin/appScripts	This directory holds copies of the application-level administration scripts. Should an application's administration scripts require refreshing, the contents of this folder can be copied to the `applications\[APPNAME]\admin` directory.
agent	SNMP agent related files.
applications	The voice applications built by Builder for Call Studio and hosted by Call Services are stored here. Each application has its own folder bearing the name of the application.
applications / [APPNAME] / admin	This directory holds the scripts that perform administrator functions affecting only the application in which the scripts reside.
applications / [APPNAME] / data	This directory contains the application's static data files required for Call Services to load the application.
applications / [APPNAME] / data / application	This directory contains the settings and call flow of the application as well as any configurations for application loggers.
applications / [APPNAME] / data / configurations	This directory holds the static voice, action, and decision element configurations created by Builder for Call Studio for this application. Depending on the size of the voice application, this directory may end up with many element configuration files.

applications / [APPNAME] / data / misc	This directory holds miscellaneous data files used by Universal Edition decision elements or other proprietary files used by the developer.
applications / [APPNAME] / java	This directory contains all Java related classes or JAR files required for this application only. No other application will have access to the Java classes in this directory.
applications / [APPNAME] / java / application	This directory contains all the classes used for this application only. Individual Java classes go in the `classes` directory while complete JAR files go in the `lib` directory.
applications / [APPNAME] / java / util	This directory contains utility classes used by the classes in the application directory. Any utility classes that refer to Universal Edition API classes must be deployed here or in the `application` directory. Individual Java classes go in the `classes` directory and JAR files go in the `lib` directory.
applications / [APPNAME] / logs	This directory contains the administrator, activity and error logs affiliated with this application. Logs are rotated daily so this directory may eventually contain many files.
common	This folder contains the Java classes and JAR files shared across all voice applications hosted on Call Services. Individual Java classes go in the `classes` directory and JAR files go in the `lib` directory.
conf	This directory holds settings files used for Call Services.
docs	Universal Edition documentation is available at http://www.cisco.com. After downloading, place the documentation here. This folder contains third party licenses for components used by Universal Edition.
dtds	The DTDs for all XML documents used throughout Call Services are found here. Many are referred to in XML documents, though others are provided for reference.
gateways	This folder contains all the installed Gateway Adapters for Call Services. Each sub-folder in this directory contains a separate Gateway Adapter.
lib	The JAR files within this folder are necessary for administration scripts to run. They are also used by the developer to compile custom Java code that uses the Universal Edition API.
license	The Call Services license files are to be placed here.
logs	Logs affiliated with Call Services itself are placed here.
management	Files required to support the JMX administration interface are found here.

The `INSTALLATION PATH\CallStudio` folder contains the following directories:

Folder	Description
eclipse	This directory holds all the required files for Call Studio and Builder for Call Studio.
eclipse\features	This folder contains descriptions of the installed features - Call Studio and Builder for Call Studio. Features consist of a set of plugins providing certain functionality.
eclipse\jre	This folder contains the JRE used by Call Studio.
eclipse\plugins	This directory contains a set of plugins defining the functionality of Call Studio.
eclipse\workspace	The voice applications built by Builder for Call Studio are stored here. Each application has its own folder bearing the name of the application.
eclipse\workspace\.metadata	An Call Studio internal system folder containing configuration and settings files.
eclipse\workspace\ [PROJECT NAME]\callflow	This directory contains the configuration files for the given voice application. Those files are used by Builder for Call Studio to properly render the call flow.
eclipse\workspace\ [PROJECT NAME]\deploy	This folder holds all the resources that will be deployed along with the given application. It can contain such components as custom Java classes and libraries as well as custom data files.

Appendix C: Glossary

These terms are used liberally throughout this user guide and it is important to understand them. The glossary includes both telephony terms as well as Universal Edition terms.

Telephony Terms

- **ASR**. Short for Automated Speech Recognition, this is the technology used by modern voice browsers to recognize the caller's spoken utterances and convert them to text. Using this technology, an application can have a completely different interface, creating a much more natural dialog-based interaction with the caller. ASR works by limiting the utterances a caller can say to a manageable number, and making the best determination of which utterance was spoken. Though far from supporting the ability to simply "talk" to the application, well-designed prompts can lead callers to say the right inputs and make the application smooth, consistent, and easy to use.
- **Bargein**. Bargein is the act of interrupting a playing prompt, most of the time to go directly to entering data. This feature can be used by the application designer to allow repeat callers to jump ahead and move faster through the application to perform common tasks. In situations such as error messages, disclaimers, etc. where the application designer would want the caller not to interrupt, bargein can be turned off.
- **Call flow**. The application call flow is the sequence of actions and events that can occur in a voice application. In Builder for Call Studio, the call flow can be represented by a flowchart showing all possible branches. This way the application designer can see all that can happen during the course of a call.
- **Confidence Score**. An ASR engine works by matching the caller's utterance to the grammar option most likely to be the one intended by the caller. The confidence score is a statistical value assigned to each utterance, indicating how certain the speech engine is about the recognition result. Confidence score values are usually represented as a decimal number between 0.0 (no match) and 1.0 (perfect match).
- **DTMF**. Short for Dual Tone Multi-Frequency, this is the technical term describing touch-tone dialing. A caller can interact with a voice application either by speaking an utterance (speech input), or punching in numbers on a telephone keypad (DTMF input). There are twelve DTMF keys on a standard touch-tone telephone pad, and sixteen keys on some special phone pads that include four additional keys called A, B, C and D extended signals. A voice application can be developed using DTMF only, speech only, or both DTMF and speech inputs.
- **Grammar**. A grammar is the mechanism by which an application designer describes the limited number of options for an utterance given to an ASR engine. A grammar can hold words or phrases, and contain guttural utterances such as "um" or "er". It may vary in size, from a couple of words to thousands of words and phrases. The larger the grammar, the less likely the ASR engine will have a dead-on match.

- **Noinput**. Noinput is an event that can occur in a voice application when the system prompts the caller for some input and the caller does not respond. After a configurable amount of time, a noinput event occurs, indicating that nothing was heard. This may be because the caller wasn't listening, was confused, or there was a problem with the connection. In any rate, a well-designed voice application would say something to clarify the prompt or ask for attention. After a configurable number of noinput events occurring one after the other, the application will usually take more drastic actions such as hanging up or transferring the caller to an operator.
- **Nomatch**. Nomatch is an event that occurs when the system prompts the caller for some input and the caller utters or enters information that is not what is expected. Like the noinput event, this will usually cause a message to be played. Nomatch events tend to occur if the caller doesn't understand what is being asked of them, the grammar is large and does not return results with a high enough confidence value, the phone line or environment is noisy, etc. This can happen with DTMF input as well if the caller entered a digit that was not an option. As with noinput events, a count can be assigned limiting the number of times a caller can fail to match an option, before they are disconnected or transferred to an operator.
- **TTS**. Short for Text-To-Speech, this technology is used by voice browsers to read a written phrase in an automated, semi-human sounding voice. The advantage of a TTS engine is the ability to make rapid changes to a phrase without having to do any human voice recording. The technology, however, still sounds robotic at the current stage. Many people have trouble understanding long sentences spoken through TTS, so almost all voice applications rely on pre-recorded audio for their prompts. TTS is still used for data that is hard to predict ahead of time or can have a large number of variable formats (such as an address).

Universal Edition Terms

- **Activity Logger**. The activity logger is one of the loggers included with Call Services that logs all the activities that a caller performs in a call session. The log file generated is application-specific, and it includes a time-stamp for each logged activity along with information such as the duration of the call, call flow components visited, recognition results and events thrown.
- **Component**. Component is used as a general term to mean a part of a Universal Edition application that can be constructed by a developer. They include elements as well as non-element parts of the application, such as a Java class that can be called when calls begin or end.
- **Configurable Elements.** Configurable elements are designed to be very reusable. They are constructed in advance and each is given a configuration that allows the application designer to change how the element functions. The more detailed the configuration, the more flexible the element. Voice, action, and decision elements are the three types of configurable elements.
- **Configuration**. Every element, in order to make it reusable, must have a mechanism by which the user can specify how they wish the element to act in an application. A form element, for

example, would need a way to determine whether it should accept DTMF, speech, or both input types. These behavioral preferences are encapsulated in a configuration, which is then represented as an XML file. The Builder for Call Studio displays an element configuration in the Element Configuration View and Call Services loads the XML configuration when it starts up.

- **Element**. Elements are all components that can appear in a voice application call flow. These include both functional elements such as voice elements and action elements and conceptual elements such as hang-up elements. In Builder for Call Studio, everything appearing in the Elements View is considered an element.
- **Global Hotlink**. A global hotlink is a link grammar that can be activated at any place in an application to perform some action or route the call to some place in the call flow. For example, the utterance "operator" with the DTMF key press '0' is a common global hotlink used in voice applications to allow the caller to seek help from a human agent anywhere in the call flow.
- **Local Hotlink**. A local hotlink is a link grammar that can be activated only from within a voice element and that either causes the element to exit with a specific exit state or throws a VoiceXML event. An example is listening for the utterance "I don't know" while in a voice element that expects numeric input. Without the hotlink, the element would encounter an no match event matching the utterance to a number.
- **Hotevent**. Like a global hotlink, a hotevent can be activated at any time during the call. While most events are triggered by the caller's activity, a hotevent can also be triggered by processes not directly associated with the caller. For example, an event could be triggered by the voice browser indicating when to interrupt on-hold audio. Like a hotlink, a hotevent can be configured to move the caller to another place in the call flow, though a hotevent may additionally execute custom developer-specified VoiceXML. This custom VoiceXML code could, for example, perform logging, or set the value of a VoiceXML variable.
- **Java**. Java is a high-level programming language that the Universal Edition software is written in and that can be used by developers to extend the functionality provided by Universal Edition.
- **JMX.** Stands for Java Management eXtensions and is a standard Java technology used to provide visibility into an application and the JVM itself. This is used by administrators to gauge the health of a system. Call Services exposes much information about itself as well as the applications running on it for access and even exposes functions to allow administrators the ability to alter how it runs.
- **Maintainer**. The maintainer of an application is an e-mail address that is sent messages by the voice browser when issues occur that require attention such as missing audio files, incorrectly formatted VoiceXML, etc.
- **MBean**. An MBean is a single unit that a system using JMX defines to provide information on that unit or functions to execute on that unit. A JMX console will typically render all MBeans exposed by a system into a tree structure to allow the administrator to navigate

though the information available. Call Services creates MBeans for metrics it exposes, functions it allows administrators to execute, and for each application deployed on it to access information and commands relative to that application.

- **Say It Smart.** This Universal Edition technology is used to take formatted data such as dates, a state abbreviation, a currency value, etc. and render it as a series of audio files played one after the other. Using Say It Smart allows an application to use pre-recorded audio files for dynamic data, providing a consistent experience to the caller, without resorting to using TTS. Universal Edition provides an API that allows developers to create their own Say It Smart types.
- **Standard Elements.** Standard elements differ from configurable elements in that they have a singular, application-specific purpose and are not expected to be very reusable since they do not have configurations. Due to the fact that they do not have configurations, standard elements are easier to construct. Action and decision elements are the two types of standard elements. An example of a standard element would be a mortgage calculator that would only be used for a specific application.
- **UID**. A UID is a user ID used by Call Services to identify users in the user management database. A developer can associate a UID with a call so that the application can dynamically make decisions based on user information and historical performance.
- **Voice browser.** A voice browser is software that is responsible for bridging the telephony system and its related hardware with VoiceXML. The voice browser connects to telephony hardware and is responsible for handling the phone calls. It is the voice browser that interacts with the ASR engine and the TTS engine, listens to the caller for input and plays audio back to the caller. It is responsible for requesting VoiceXML pages from an external system (Call Services) and parsing the pages that come back. Voice browsers typically run on a separate machine from the one on which Call Services is installed.
- **VoiceXML Gateway Adapter.** Gateway Adapters are small plugins installed on Call Services that provide compatibility with a particular voice browser. Once installed, all Universal Edition voice elements (and all custom voice elements not utilizing browser-specific functionality) will work on that voice browser.
- **XML-over-HTTP**. The XML-over-HTTP API (also known as the XML API) is developed to allow the use of non-Java programming languages to extend Universal Edition software. This API works by sending XML content over a standard HTTP connection and receiving XML in response. Using this API, any programming language that can handle HTTP can be used to extend Universal Edition software.