# Introduction



# VoiceXML Overview

Since its introduction, VoiceXML has become the standard technology for deploying automated phone systems. The overview of traditional technologies used to develop interactive voice response systems is given to understand the acceptance of VoiceXML by enterprises, carriers, and technology vendors.

## VRU Technologies Limitations

Conventional VRU solutions are not speech enabled, and upgrading to speech recognition on a traditional VRU platform is difficult. These VRU solutions do not allow the choice and flexibility necessary to meet the increasing demands of fast, quality service and a consistent experience across phone and web contact channels. These limitations of conventional VRU solutions are overcome by implementing VoiceXML as one-size-fits-all VRU solution because VoiceXML is flexible and powerful.

## VRU Development Simplification with VoiceXML

VoiceXML is a programming language that was created to simplify the development of VRU 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.

VoiceXML utilizes the same networking infrastructure, HTTP communications, and markup language programming model. 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 VRU provides options when creating, deploying, and maintaining automated customer service applications. By providing the standards-based feature of VoiceXML, organizations are obtaining a number of benefits including:

- **Portability**—VoiceXML eliminates the need to purchase a proprietary, special purpose platform to provide automated customer service. The standards-based feature of VoiceXML allows VRU applications to run on any VoiceXML platform, eliminating customer bound. A VoiceXML-based VRU offers businesses a choice in application providers and allows applications to move between platforms with minimal effort.
- **Flexible application development and deployment**—VoiceXML enables freedom of choice in VRU application creation and modification. Because it is similar to HTML, development of VRU applications with VoiceXML is simple and does not require specialized knowledge of proprietary telephony systems. VoiceXML also 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 VRU based on their own organizational priorities.
- **Extensive integration capability**—VRU applications written in VoiceXML can integrate with and use existing business applications and data, which extends the capabilities of core business systems already in use. A VoiceXML-based VRU 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 consider their VRU like their enterprise applications and fulfill business demands with an integrated customer-facing solution.

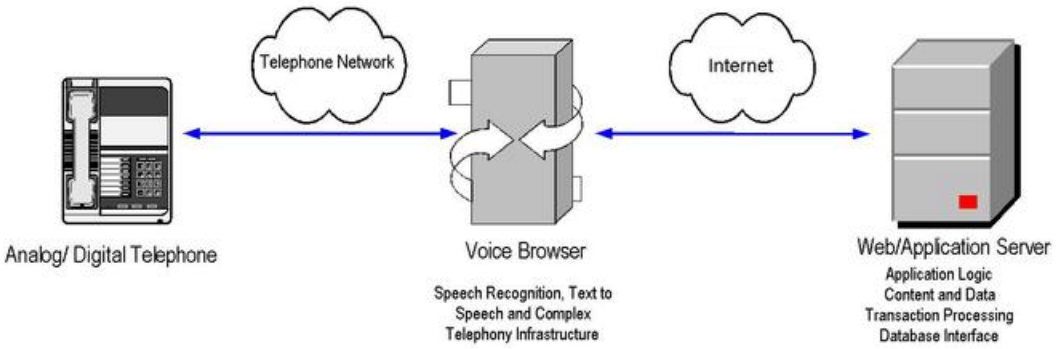
  The wide array of options available allows businesses to maximize existing resources to deliver better service at lower cost.
- **Reduced total cost of ownership**—The options offered by a VoiceXML-based VRU reduces the total cost of ownership in several key areas:
  - **Speech capability is standard**—The VoiceXML architecture directly supports integration with speech recognition, which makes a VoiceXML-based VRU a cost- effective alternative to retrofitting a traditional VRU for speech. Incorporating speech into an VRU 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 VRU applications written in VoiceXML.
  - **Affordable scaling**— In a VoiceXML-based VRU model, application logic resides on a web/application server and is separate from telephony equipment. Businesses can save money 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 VRU solutions for budgets of all sizes. Businesses pay only for necessary application features.

## VoiceXML Use

VoiceXML is designed to leverage web infrastructure. VoiceXML is analogous to HTML, which is a standard for creating web sites. The development of voice applications using VoiceXML is simple and straightforward. Because the complexities of voice applications development 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




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.

## VoiceXML Development Challenges

# Unified CVP Solution

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

Unified CVP consists of three main components: Cisco Unified Call Studio (Call Studio), VXML Server, and Unified CVP Elements. Each of these components is discussed in further detail in this section.

## Call Studio

Call Studio is a development platform for the creation of voice applications. Call Studio provides a framework on which the host of Unified CVP and third-party tools appear with a robust, consistent interface for voice

application designers and developers to use. Call Studio provides a true control panel for developing all aspects of a voice application; each function is implemented as a plug-in to the Call Studio platform.

The most important plug-in for Call Studio is Builder for Call Studio (or Builder), the component that Cisco built to provide 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 percent over the generation and management of flat VoiceXML files.

You can access Call Studio documentation by accessing the Help menu in Call Studio. This guide, however, includes a brief introduction to Call Studio in the Call Studio Introduction section in this chapter.

# VXML Server

VXML Server is a powerful J2EE- and J2SE-compliant run-time engine. VXML Server provides:

- **Robust back-end integration**: VXML Server runs in a J2SE and J2EE framework, giving the developer access to the full collection 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 VXML Server 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 all details about an application can be determined at runtime.
- **System Management**: VXML Server 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**: VXML Server 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.

VXML Server capabilities listed above are discussed in further detail in Administration, User Management, and VXML Server Logging sections.

# Unified CVP Elements

The Unified CVP Elements are a collection of prebuilt, fully tested building blocks to speed application development

- **Browser compatibility**: Unified CVP'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**: Unified CVP Elements encloses the parts of a voice application, from capturing and validating a credit card to interfacing with a database. Unified CVP Elements greatly reduce the complexity of voice applications by managing low-level details.
- **Configurable content**: Unified CVP Elements can be configured by the developer to customize their output specifically to address the needs of the voice application. Prebuilt configurations using proven dialog design techniques are provided to further speed the development of professional grade voice applications.

In Unified CVP, 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. VXML Server 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 Unified CVP, elements are defined as:

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

Many elements in Unified CVP 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 fuction for elements to communicate with each other. The data comes in two forms: element data and session data.

- **Element data**: 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**: 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 to be 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 functions supported by Unified CVP: a fixed

configuration for the element, a Java API to dynamically configure prebuilt elements or to define new ones, and an API accessed with XML-data delivered over HTTP.

- **Fixed configuration**: 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.
- **Java API**: 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
- **XML-over-HTTP (or XML API for short)**: Provides developers with the ability to use 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 VXML Server. 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 VXML Server and the external system.

**Note** HTTPs is not supported with XML-over-HTTP.

# Voice Elements

Almost all voice applications must use a number of dialogs with the caller, playing audio files, interpreting speech utterances, capturing data entered by the user, and so on. 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 covered in *voice 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, and so on. These pre-built components can then be reused throughout the call flow wherever needed.

Voice elements are built using the Unified CVP Voice Foundation Classes (VFCs), which produce VoiceXML compatible with multiple voice browsers (see the *Programming Guide for Cisco Unified CVP VXML Server and Cisco 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**—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**—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. The designer should understand the consequences of modifying these properties.
- **Audio Groups**—Nearly all voice elements involve the use of audio assets, whether in the form of prerecorded 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, and so on. An audio group may contain any number of audio items. Audio items are defined as prerecorded 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.

A voice element's configuration can be either fixed or dynamic.

- **Fixed configurations**—XML files containing the desired settings, VoiceXML properties, audio groups, and variables that are then loaded by VXML Server. The same configuration is applied each time the voice element is called.
- **Dynamic configuration**—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 Unified CVP, refer to Element Specifications for Cisco Unified CVP VXML Server and Unified Call Studio.

# VoiceXML Insert Elements

There are certain situations in a voice application where a developer may want to include prewritten VoiceXML into their Unified CVP application. The developer may want fine-level control over a specific voice function at the VoiceXML tag level without getting involved with constructing a custom configurable element in Java. Additionally, the developer may want to integrate VoiceXML content that has already be created and tested into a Unified CVP 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 can cause the following results:

- 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
- 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.

- **Decision Element**—Comprise 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 end the call.

The results of a decision element are represented as exit states. Although many decisions are boolean in nature, (for example, 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**—Comprises business logic that performs tasks not affecting the call flow (that is, 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. An example of an action element might be one that 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 Unified CVP installation bundles some prebuilt 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.

# Flag Elements

*Flag elements* provides a mechanism that analyze the activities of callers to determine the part of an application that is most popular, ambiguous , and difficult to find. To do these analyses, the developer need to know when the callers reach a certain point in the application call flow. This check can also be done within the call itself, by changing its behavior dynamically if a caller visited a part of the application previously.

- **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.

# Modular Application

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. VXML Server provides two different ways to foster modular application, 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 VXML Server, 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 VXML Server. The caller is not aware that they are visiting a new application, but VXML Server treats it as if it were a separate call with separate logging, administration, and so on. 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 VXML Server 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 VXML Server, they can be hosted anywhere accessible via a URL and does not even need to be a VXML Server application at all.

The VXML Server 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.

To use a subdialog, several special elements are needed in the source and subdialog applications. Visiting a subdialog from a 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 VXML Server 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 VXML Server 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 VXML Server 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.

## Subflows

Subflows are developed in an application as independent modules. Each subflow has a resource and an associated folder to save the subflow information under Call Studio project. Call Studio uses Callflow Editor to open both main call flow and subflows. Subflows are created from the Call Studio Navigator view. Subflows are saved under the subflow folder as pages. Each page contains its associated element configuration. When a new subflow is created from the navigator, a default subflow with a Subflow Start element is created in workspace and opened in the editor. Subflow can be completed by adding additional call flow elements. When the subflow is saved, subflow is saved under the subflow folder in pages. Each page contains element configuration for that page.

Subflows has the following elements:

- **Subflow Start Element**: Subflow Start Element is the entry point of the subflow. Subflow processing starts at Subflow Start Element.
- **Subflow Return Element**: Subflow Return element is the exit point for the subflow processing. When the subflow return is reached, subflow is exited.
- **Call Subflow Element**: Call Subflow Element is used for calling a subflow. Call Subflow element passes the flow control to the Subflow Start element and regains the control when the Subflow Return element is processed. A subflow can be called any number of times in an application.

# Unified CVP Call Studio Introduction

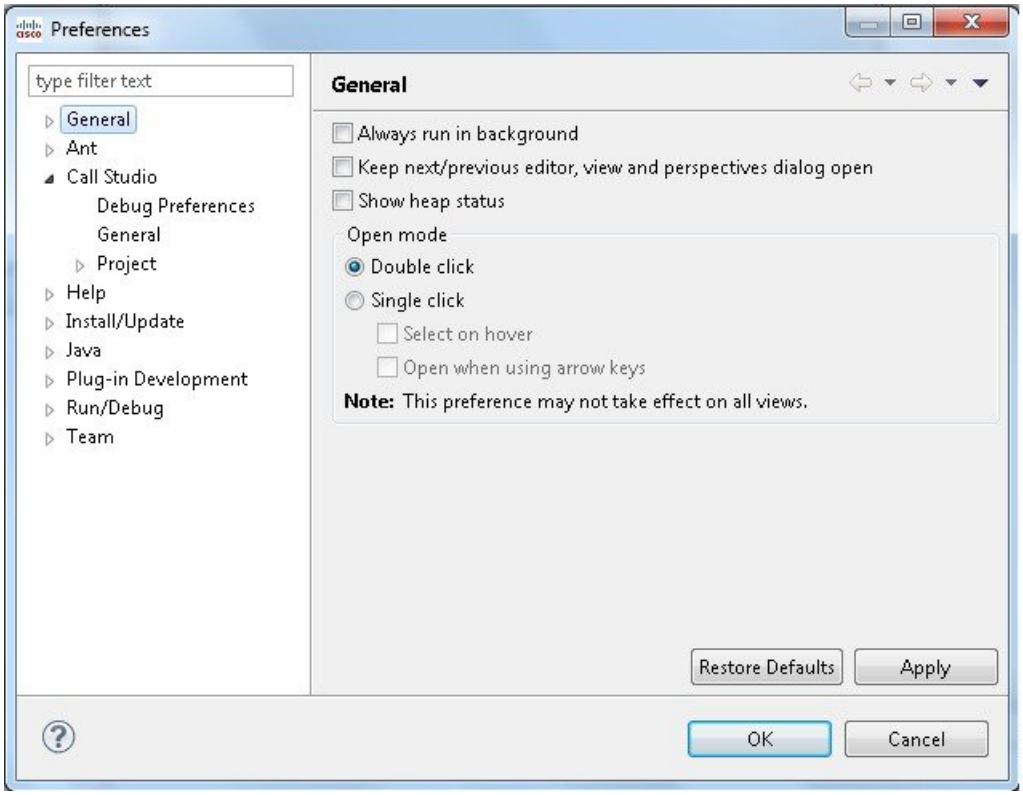
Call Studio is a platform for creating, managing, and deploying sophisticated voice applications. Call Studio is developed using the Eclipse framework, though no knowledge of Eclipse is necessary to work with Call Studio. Call Studio acts as a container in which features - called plug-ins - are encapsulated. 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.

## 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. If modifications are made to any of these settings, we recommend that Call Studio be restarted so that the new settings can take effect.

*Figure 1: Setting Preferences in Call Studio*




The General window includes the following options:

- **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.

Starting from Release 11.5 the following fields have been introduced under **Preferences** > **Call Studio** > **Debug Preferences**.

- **Proxy URL**— Enter the Proxy URL for Context Service network traffic.
- **Timeout**—This setting controls the waiting period of the client for a response from Context Service. It can be set between 1200 and 5000.
- **Test connection**— This button shows the validity of connection data through the Proxy URL.

**Note** For more information refer to *Configure Context Service Connection Data in Call Studio* section in *Administration Guide for Cisco Unified Customer Voice Portal* at https://www.cisco.com/c/en/us/support/customer-collaboration/unified-customer-voice-portal/tsd-products-support-series-home.html.

# Builder for Call Studio

Builder for Call Studio is a graphical user interface for creating and managing voice applications for deployment on VXML Server. VXML Server is the runtime framework for Unified CVP voice applications.

A complete dynamic voice application can be constructed within Builder, including call flow and audio elements. The philosophy behind 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, 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.

Builder works as a flowcharting program tailored specifically for building voice applications. Most of the familiar features of a flowcharting tool are present in 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 of the resources required to build and deploy voice applications that run on VXML Server.

The `callflow` folder contains the XML files that make up that application's call flow and element configurations. The `app.callflow` file opens the Callflow Editor and graphically represents 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.

The `subflow` folder contains of all the XML files related to subflows. Each subflow has a folder inside the subflow folder, and it will contain the files related to subflow.

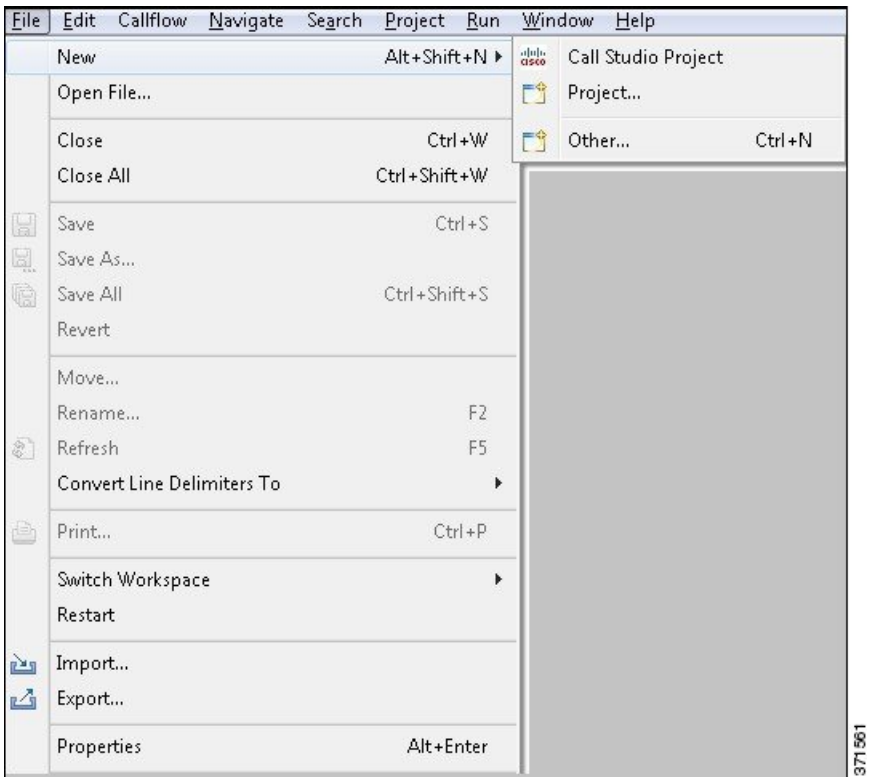
The app.callflow and .subflow file open the Callflow Editor and recreate the call flow based on the information found in the XML files in the callflow and subflow folder. Every time the application is saved, these files are updated.

# Create Call Studio Project

**Procedure**

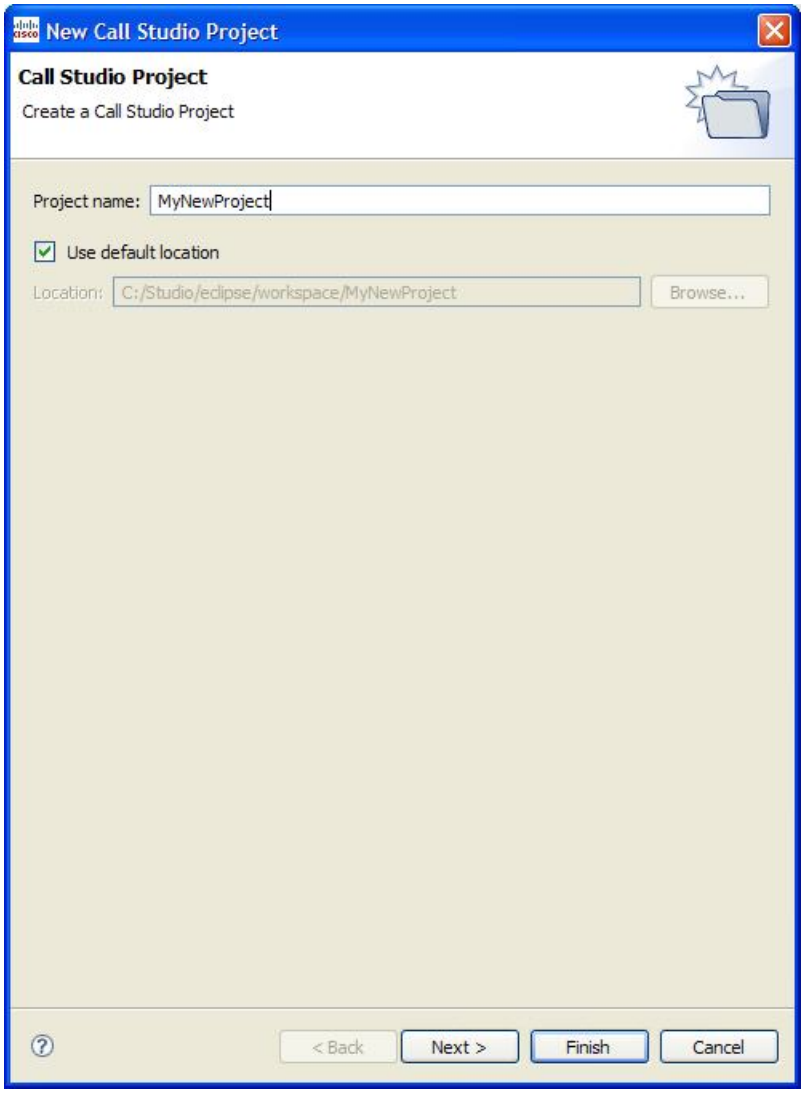
**Step 1** From within the Call Studio program, choose **File** > **New** > **Call Studio Project**.

*Figure 2: Create New Call Studio Project*



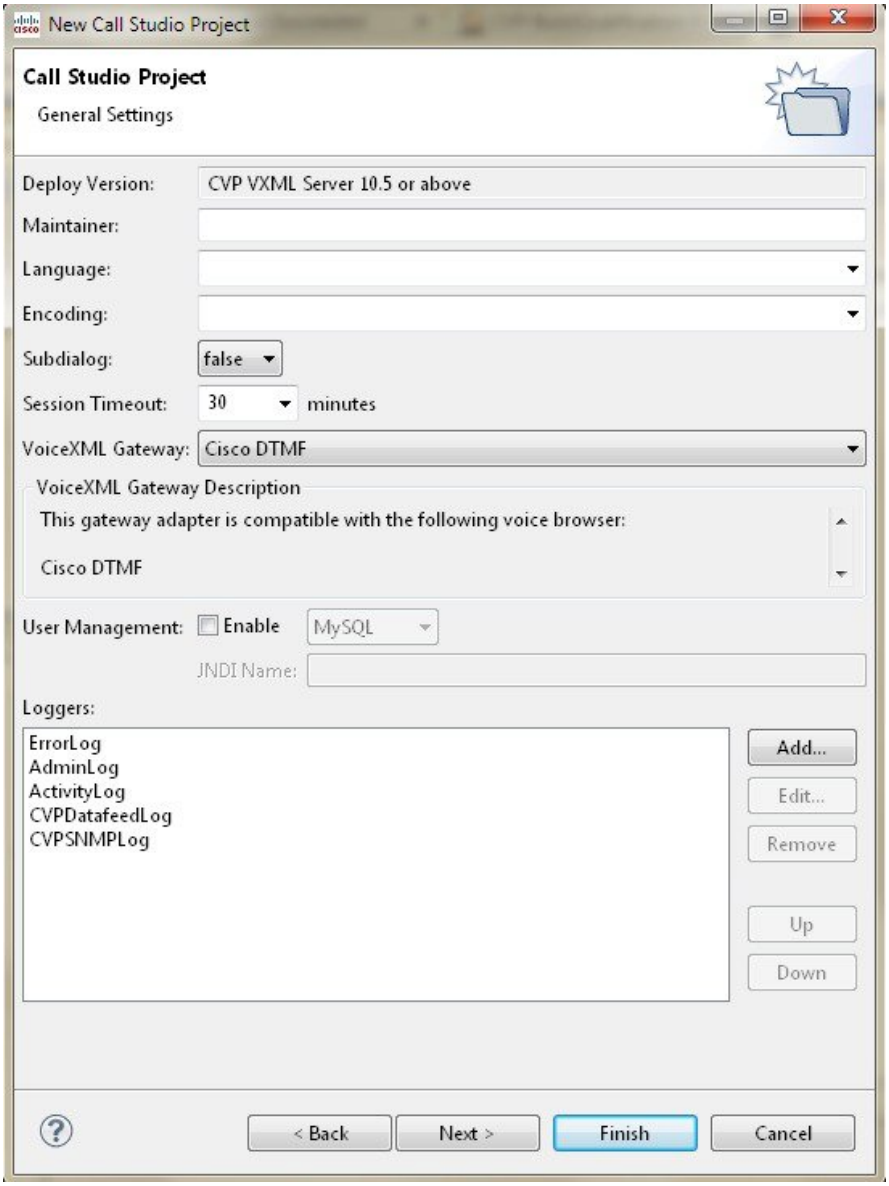

**Step 2** Enter a name for the new Call Studio project and select **Next**. Leave **Use default** location checked to create the new project in the default workspace directory.

*Figure 3: Name the Call Studio Project*



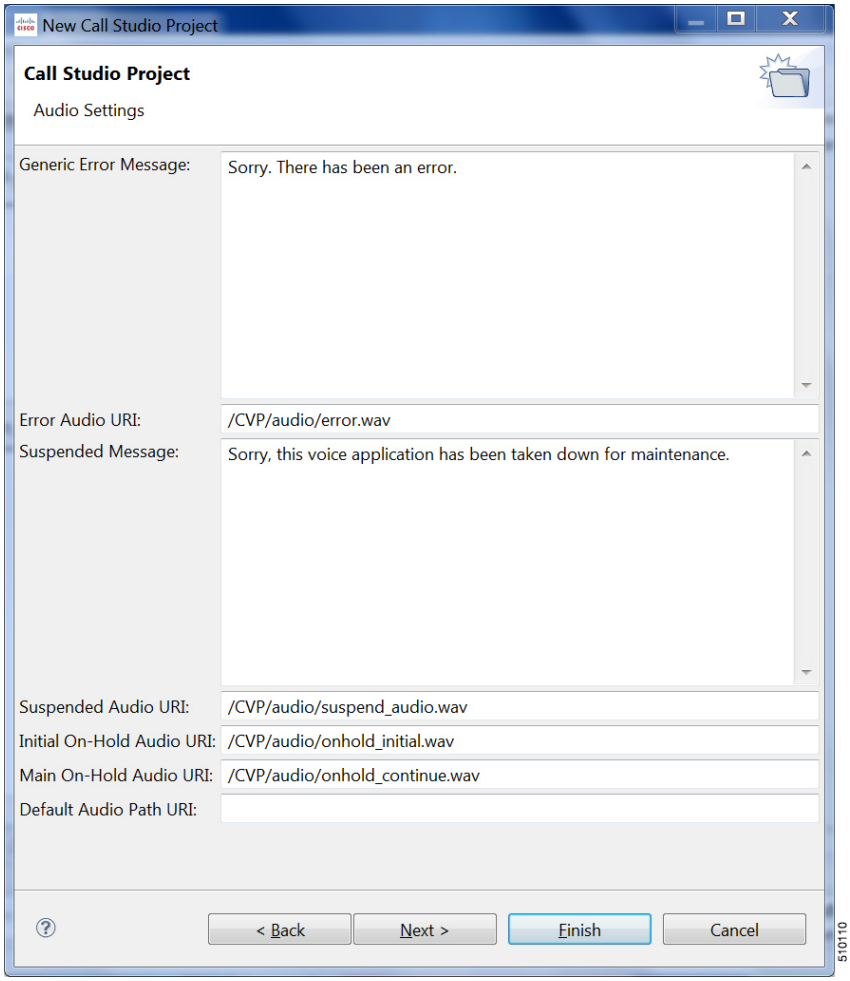

**Step 3** Enter the General Settings for the new Call Studio Project and select **Next**. You can always change General Settings later.

*Figure 4: Call Studio General Settings for New Project*



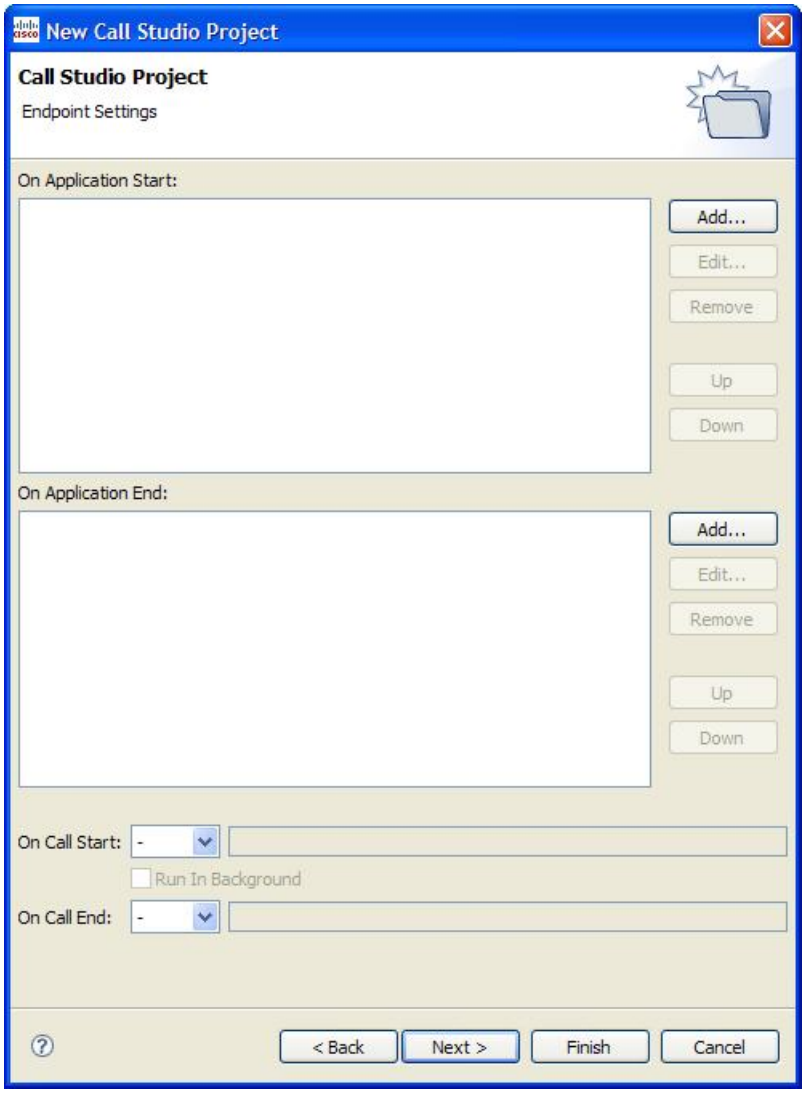

**Step 4** Enter the Audio Settings for the new Call Studio Project and select **Next**. You can always change Audio Settings later.

*Figure 5: Call Studio Audio Settings for New Project*



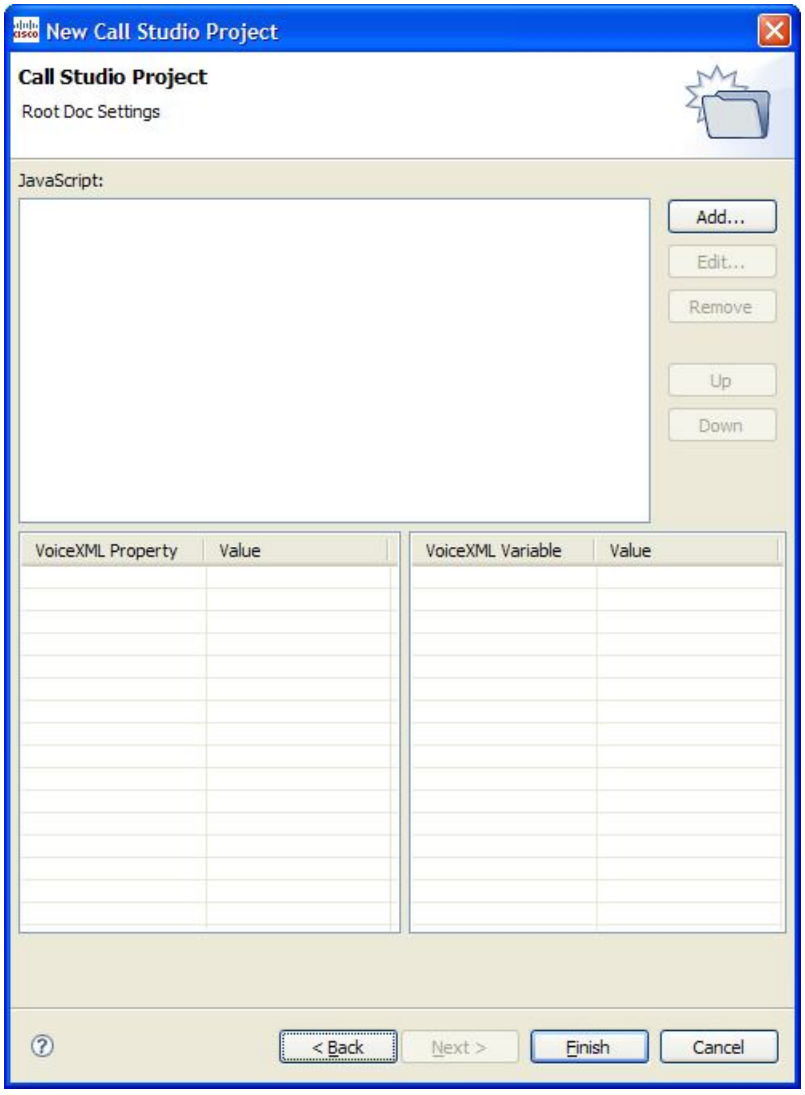

**Step 5** Enter the Endpoint Settings for the new Call Studio Project and select **Next**. You can always change Endpoint Settings later.

*Figure 6: New Project Endpoint Settings*



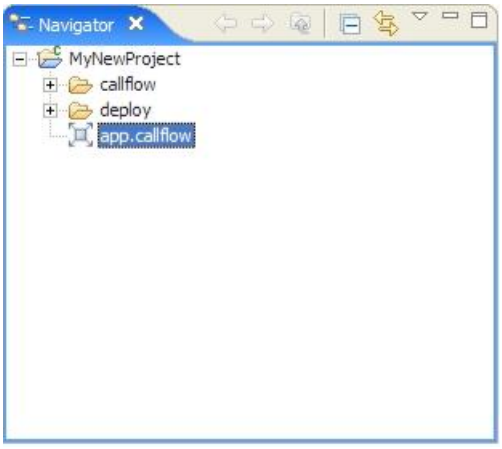

**Step 6** Enter the Root Document Settings for the new Call Studio Project and select **Finish**. You can always change Root Document Settings later.

*Figure 7: Root Document Settings for New Project*




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

*Figure 8: New Project Showing in Navigator View*




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

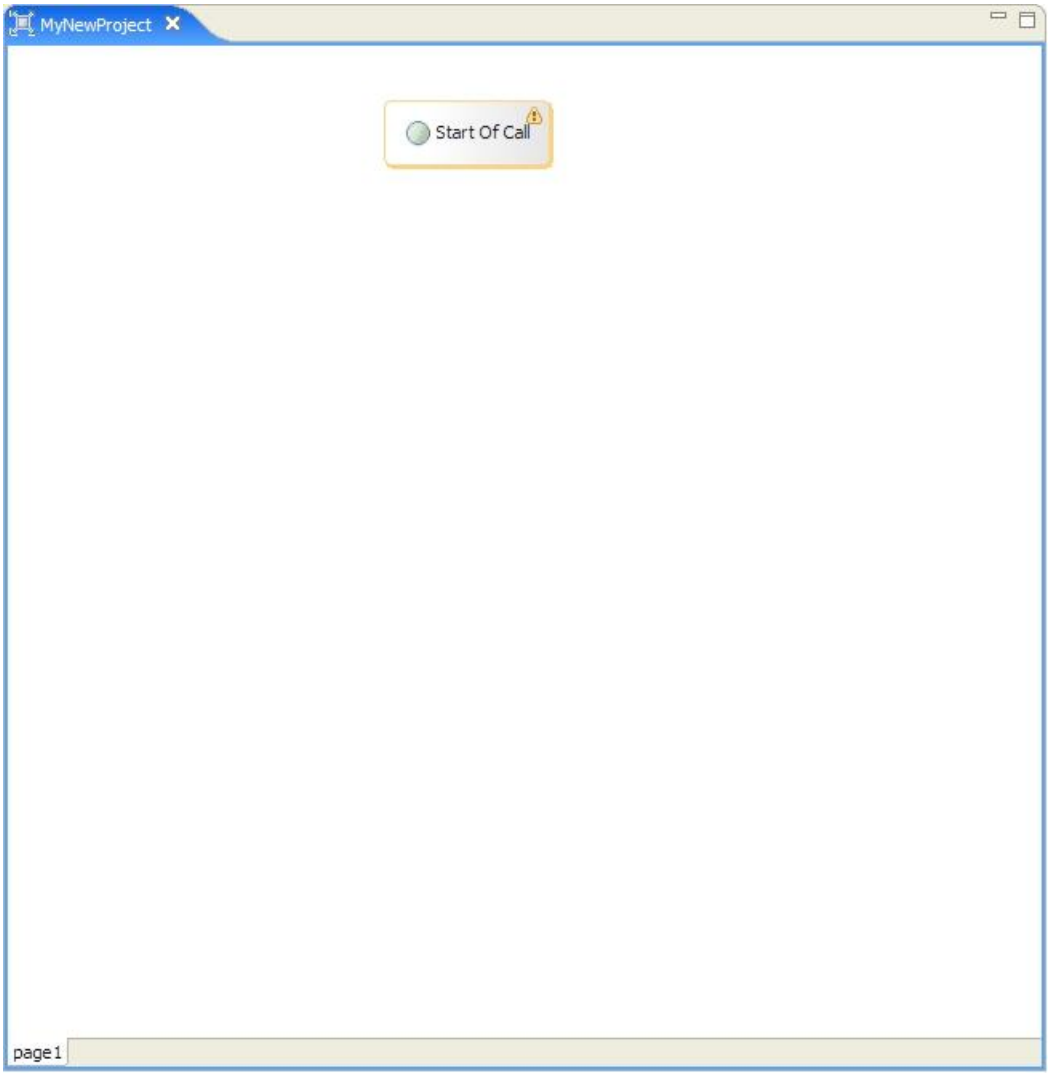
**Note** You can move the Call Studio elements by one pixel with in the Callflow Editor by using the following keyboard shortcut.

- To move left, press Ctrl+Alt+H
- To move right, press Ctrl+Alt+J
- To move up, press Ctrl+Alt+K
- To move down, press Ctrl+Alt+L

You can also select multiple elements and move them by using the same keyboard shortcut.

For the x-coordinate and y-coordinate of a selected element see the left side of the status bar. If multiple elements are selected then the coordinates of the left most element is displayed.

*Figure 9: New Project Automatically Started in Callflow Editor*

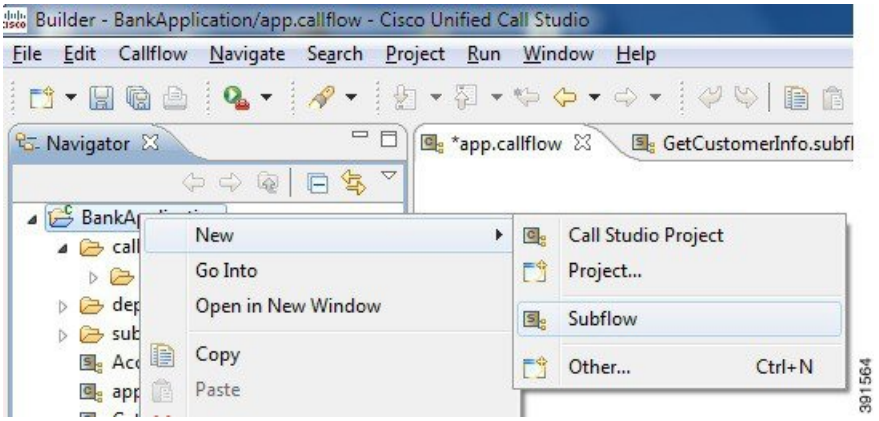



# Create Subflow

**Procedure**

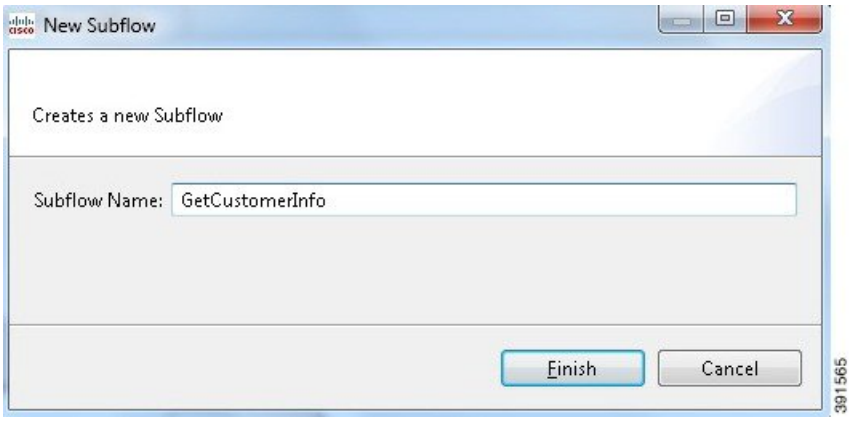
**Step 1** From within the Call Studio program, choose **File** > **New** > **Subflow**.

*Figure 10: Create Subflow*




**Step 2** Enter a name for the new Subflow.

*Figure 11: Name the Subflow*




**Step 3** Click **Finish**.
A new subflow is created with the Subflow Start element in the work space.

*Figure 12: New Subflow Created*

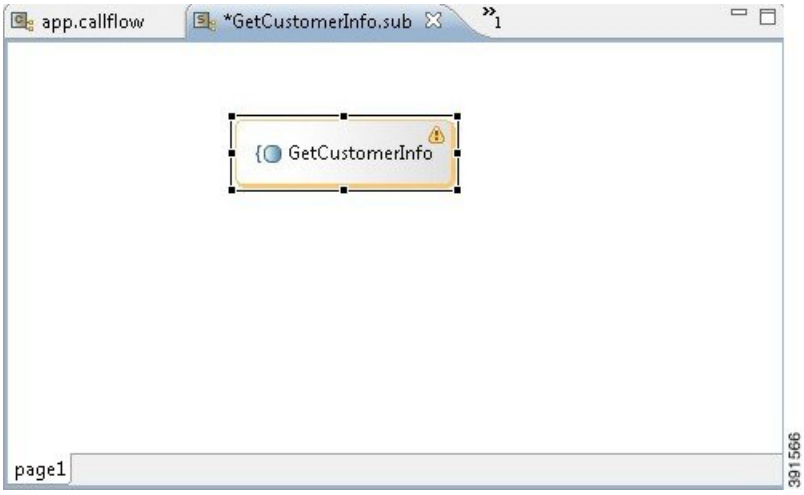
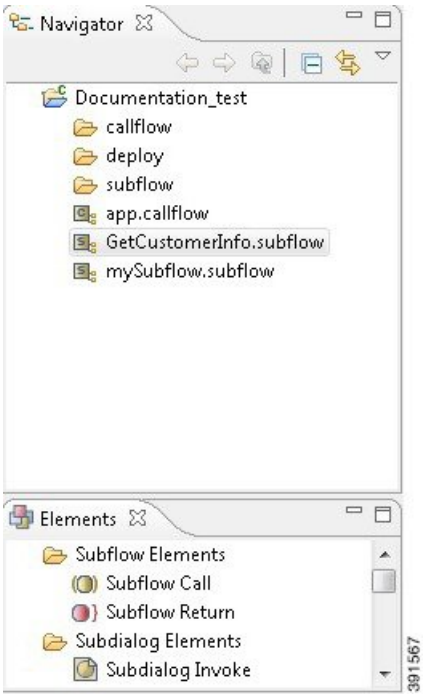



*Figure 13: Subflow Folder Created*




## Refactor Call Flows

Operations such as conversion of main flow to subflow, subflow to mainflow, and moving subflows from one application to another application requires manual process.

### Convert Main Flow to Subflow

**Procedure**

**Step 1** Create a subflow with an appropriate name.

**Step 2** Copy the elements in the main flow and paste them in the subflow.

**Step 3** Add the subflow call element to invoke the converted subflow.
The main flow in the project will still exist; however, the application logic will be migrated to the newly created subflow.

**Note** These steps can be used to function a large application to smaller subflows and remove duplicated call flow logic.

### Copy Subflows

**Procedure**

**Step 1** Find the resource location. Right-click on an application, select properties from the menu, and then select the resource location of the application where the subflow exists.
The properties window shows the resource location.

**Step 2** Go to the resource location and right-click to copy the subflow file that needs to be copied to the target application. (Example, AccountVerification.subflow)

**Step 3** Right-click to paste at the resource location of the target application.

**Step 4** After pasting the subflow, copy and paste the class files, libraries, and media files from the source application to the target application.

# Call Stack History View

This is a new view in Cisco Unified Call Studio. The information that previously appeared in the **Variables** view now appears in a new view called the **Call Stack History** view. The **Call Stack History** view displays information about the variables that are associated with the stack frame that you selected in the **Debug** view. Click an element in the **Editor** view to view the corresponding data variables in the **Call Stack History** view. In addition, Java objects can be expanded to show the fields. The data variables that are displayed in the **Call Stack History** view can be edited.

# Variables View

You can modify data-variable values directly from the **Variables** view, while you debug a call flow. Click a particular element in the **Editor** view to see the name and value of the corresponding data variables in the **Variables** view. The **Variables** view allows you to modify data-variable values for elements in both the main flow and subflow.

## Modify Data Variables in the Variables View

**Procedure**

**Step 1** Click on an element in the Editor view to view the corresponding data variable values in the Variables view.

**Step 2** Select the data value and modify the value directly from the value pane or right click on the data variable and select **Change Value** to modify the value**.**

**Note** The updated data values are valid only for the current debugging session for one single call session.

# 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 by choosing the **Help** > **Help Contents** menu option.

***Figure 14: Call Studio Online Help Access***