Configurable Elements

The large array of Unified CVP Elements bundled with Unified CVP software encapsulate a lot of functionality a typical voice application requires. There are, however, situations where more customized, proprietary, or robust elements are desired. This is certainly the case with action and decision elements, which tend to be highly specialized for interfacing with proprietary backend systems or implementing custom business logic. To a lesser extent this applies for voice elements as well. Unified CVP has tried to provide pre-built voice elements that cover most of what a typical voice application requires. There is always the option of using VoiceXML insert elements to handle a situation that there are no voice elements to manage.

The disadvantages of VoiceXML insert elements, however, are in their inconsistencies across various voice browsers, the size they can be before management becomes burdensome, the difficulty of interacting with backend systems, and their overall performance. Custom voice elements are the best way to achieve these goals as they are fast, use the Unified CVP Voice Foundation Classes (VFCs) that work consistently across multiple voice browsers, and are built with Java, so complex calculations and integrations are simple.

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

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

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

Design

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

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

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

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

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

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

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

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

Common Methods

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

- `String getElementName()`

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

- `String getDescription()`

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

- `String getDisplayFolderName()`

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

- `ExitState[] getExitStates()`

  This method defines the exit states this element can return. It is necessary in order for the Builder to properly render the exit state dropdown menu when the element is right-clicked. The method returns an array of `ExitState` classes. The `ExitState` class encapsulates the real and display name for an exit state. The display name is used only by the Builder and the real name is used everywhere else (within code or XML decisions).

Note For configurable action elements, this method need not be implemented because all action elements automatically have a single exit state named *done*.

- `ElementData[] getElementData()`

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

Note Element data does not have a display name.

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

- `Setting[] getSettings()`

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

  - Its real and display name. The Builder uses the display name in the Element Configuration Pane and the real name is used everywhere else (such as when referring to the setting in the element Java code).

  - The setting data type. The data type determines how the information is entered in the Builder. The following lists the different data types available:

    Boolean – The boolean data type displays a dropdown menu with two options, *true* and *false*.

    Enumerated – The enumerated data type displays a dropdown menu with the options defined in the enumeration.

    Float – The float data type displays a text box that accepts only floating point numbers. The setting can define upper and lower limits for the entered number.

    Integer – The integer data type displays a text box that accepts only integer numbers. The setting can define upper and lower limits for the entered number.

    String – The string data type displays a text box that accepts any input.

    Textbox – The textbox data type displays a button that when clicked produces a large text field that accepts any input. It should be used when text content tends to be long and it is desirable to give more space to enter the setting value than would usually be given with a string data type.

  - Whether the setting is required or repeatable. A required setting is displayed with a red star next to it and the Builder will not allow the application to be deployed or validated if this setting is left blank. A repeatable setting is displayed with a plus sign next to it and can be given multiple values by the application designer. A setting can be both required and repeatable in which case there must be at least one value.

  - Whether the setting allows substitution within it. Substitution is a mechanism for specifying the assembly of dynamic content in the Builder at build time. A setting can be configured to allow substitution or prevent it, for example if that setting's behavior would be too unpredictable if its value were set dynamically. See the User Guide for Cisco Unified CVP VXML Server and Unified Call Studio for information on substitution.

  - The setting default value. When an element is dragged to the workspace for the first time, the element can specify default values for all settings. This allows the application designer to create applications very rapidly by choosing the default values and tweaking them later during the testing phase. A setting need not have a default value, especially if one cannot be predicted (such as a call transfer phone number).

  - Setting dependencies. A setting can specify criteria that determine whether it appears in the Builder's Element Configuration Pane. The criteria involve setting the relationship between the setting and the other setting(s) that it depends on. One can even build complex logic expressions for determining when the setting appears. The main purpose for setting dependencies is to simplify the process of configuring the element in the Builder. By displaying only the settings appropriate for the configuration the designer desires, settings that are not applicable can be safely hidden to avoid confusion and possible conflicting information. For example, a setting for specifying the confidence value of a data capture field would not be required if the input mode of the element were set to DTMF. So one can set the confidence setting to appear only when the input mode setting is not DTMF. Many Unified CVP Elements employ dependencies to simplify their configuration and so are good examples of how dependencies are implemented.

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

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

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

Configuration Classes

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

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

Action Elements

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

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

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

Remote Execution

For remote execution of the Action Elements, the **Remote Execution** checkbox is added in the **General Settings** tab.

On selecting the checkbox, the default URI mentioned below is auto populated.

For HTTP and RPC call→`remote://system/?classurl=<fully_qualified_java_class_path>`

For example:
`remote://system/?classurl=com.cisco.cvp.callstudio.Action.CustomAction`

Where `remote://system` indicates that the configurations will be fetched from the **Remote Url Settings** property tab which is application specific.

Note If a direct remote server URI is provided, then that IP:Port will be used and not fetched from the **Remote Url Settings** property tab.

For example:
`http://<IP>:<Port>/<target_path>/?classurl=<fully_qualified_java_class_path>`

Add Events in the Element Configuration to handle any particular or general exception gracefully.

Decision Elements

A configurable decision element extends the abstract Java class `DecisionElementBase` found in the `com.audium.server.voiceElement` package. This class has default implementations of the abstract configuration methods inherited from the `ElementBase` class. The default implementation sets an empty configuration (null name, folder name, description, element data, settings, and exit states). This was done because this same base class is extended to create generic decision elements as well. What makes a configurable decision element different is the fact that it defines a specific configuration, so the custom decision element must re-implement all the configuration methods rather than relying on the default implementation. Some of the default implementations, though, may be appropriate even for a custom decision element. For example, if the decision element creates no element data, the custom decision element need not implement the `getElementData` method as the default implementation is sufficient.

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

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

Remote Execution

For remote execution of the Decision Elements, the **Remote Execution** checkbox is added in the **General Settings** tab.

On selecting the checkbox, the default URI is auto-populated as follows:

For HTTP and RPC call - `remote://system/?classurl=<fully_qualified_java_class_path>`

For example:
`remote://system/?classurl=com.cisco.cvp.callstudio.Action.CustomDecision`

`remote://system` indicates that the configurations will be fetched from the **Remote Url Settings** property tab which is application-specific.

Note If a direct remote server URI is provided, then that IP: Port will be used and not fetched from the **Remote Url Settings** property tab.

For example:
`http://<IP>:<Port>/<target_path>/?classurl=<fully_qualified_java_class_path>`

Add Events in the Element Configuration to handle any particular or general exception gracefully.

Voice Elements

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

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

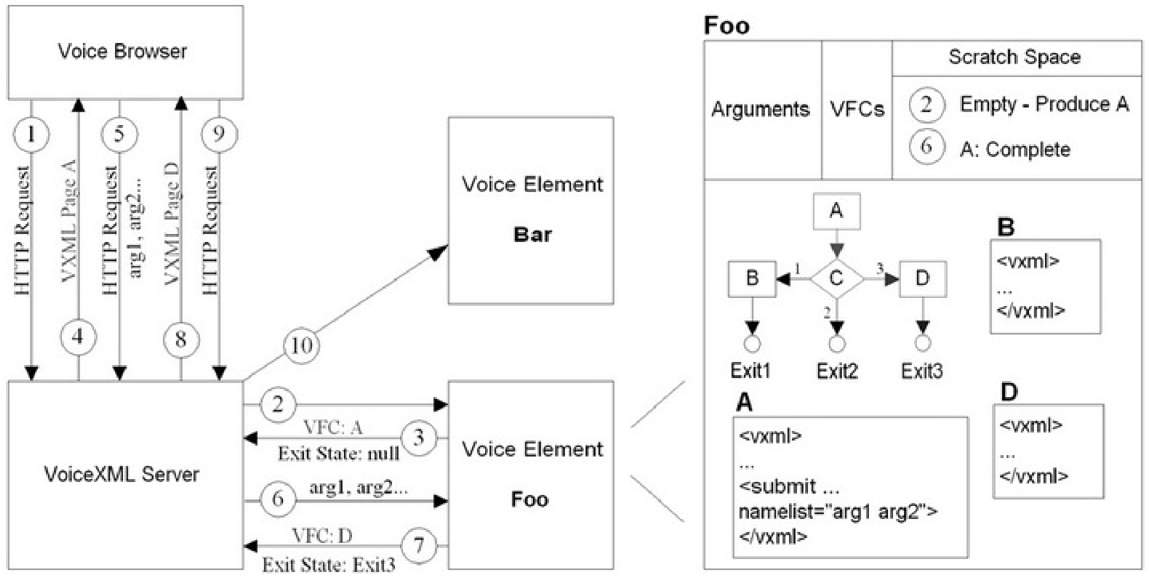
Each VoiceXML page generated by a voice element begins as a *shell* page that contains the VoiceXML VXML Server requires. The voice element then adds to this page any custom VoiceXML content desired before passing it back to VXML Server to send to the voice browser. Most voice elements will require multiple pages, the content of each depending on the actions of the caller. When a voice element is done producing VoiceXML pages, it returns an appropriate exit state so VXML Server can visit the next element according to the call flow. As long as the proper VoiceXML is passed back through the method, the developer is free to do anything allowed by Java, including creating helper classes, accessing backend systems, and so on.

Having a single voice element class produce multiple VoiceXML pages poses a problem. With multiple calls simultaneously accessing the element in various stages of a call, how is the voice element to know where a particular caller is within the element at any one moment in time? Unified CVP helps by providing “scratch space” for custom voice elements to store any data it wishes. VXML Server maintains separate scratch data for each call and makes this data available to the voice element through the Session API. Usually the data stored in the scratch space will be the state of the element for a particular call. Each time the method is called, the element can check the scratch space, determine where the caller is within the element’s internal call flow, and produce the appropriate VoiceXML page. Borrowing web application terminology, scratch space provides “session and state management” functionality to voice elements. Any Java class can be added to the scratch space so the developer can get as complex as desired in handling the state management.

Note When the voice element is complete, the scratch space is automatically cleared by VXML Server, meaning that a voice element that is revisited in the same call will start off with empty scratch space.

The following figure shows a diagram that visualizes the above points concerning how a voice element interacts with VXML Server and the voice browser.

Figure 1: Voice Element Interaction with the VXML Server and the Voice Browser



The diagram shows a typical exchange between the voice browser, VXML Server, and several voice elements as follows:

1. The voice browser makes an HTTP request for a VoiceXML page to VXML Server.
2. According to its record of the application call flow, VXML Server accesses the voice element Foo. The voice element Foo is shown on the right. The element is coded to contain within it a small call flow with three possible exit states. It can produce three separate VoiceXML pages, A, B, and C. In step 2, the scratch space is empty, indicating that the element is being visited for the first time. Foo therefore needs to produce the VoiceXML page A.

Note The A contains in it a submit that points back to VXML Server and includes two arguments *arg1* and *arg2*.

3. Foo produces the VoiceXML page by assembling VFC objects and passes those objects back to VXML Server. Before exiting, it puts in the scratch space information indicating that page A was completed. Foo returns a `null` exit state, indicating to VXML Server that the voice element is not done.
4. VXML Server converts the VFC objects into a complete VoiceXML page that is sent back to the voice browser.
5. The voice browser has parsed the VoiceXML and makes a new request for the next VoiceXML page. The request contains the two arguments specified in the VoiceXML page A.
6. VXML Server knows to go back to Foo because it previously returned a `null` exit state. It revisits Foo, passing along the two arguments. Since VXML Server maintains the session for each call, and the scratch space is stored in the session, Foo can access the Session API to get the scratch space. Foo's scratch space indicates that A is complete. It makes a decision C based on the arguments passed to it and chooses option 3, therefore requiring page D to be produced. Foo also notes that after producing this page, it is done.
7. Foo returns the VFC objects containing page D and returns with the exit state "Exit3".
8. VXML Server produces the VoiceXML page and sends it to the voice browser.
9. The voice browser parsed the VoiceXML page and asks for the next one.
10. Finally, VXML Server notes that Foo returned an exit state of *Exit3* so refers to the application call flow to discover where to go once Foo returns an exit state of *Exit3*. It determines that the next voice element to access is *Bar*. The call continues in this manner until it ends.

Restrictions and Design Considerations

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

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

- The voice element cannot use a `<submit>` or a `<goto>` to link to a VoiceXML page external to VXML Server. When it is time to return to the voice element in a VoiceXML page, a URL pointing back to VXML Server must be used. This is required by the design using VXML Server as the abstraction layer between the voice browser and the voice element. The only exception to this rule allows for the call to continue from a Unified CVP application to an external application but only if steps are taken to properly end the VXML Server session first.
- The VoiceXML `<form>` the developer wishes the browser to visit first must be named *start*.
- The voice element must obtain the `VPreference` object to use in all VFC constructors by calling the `getPreference()` method in `VoiceElementData`. This `VPreference` object contains, among other things, the voice browser choice made by the application designer in Builder for Call Studio (or dynamically in a start of call action). `VPreference` is a VFC and is described in more detail in Voice Foundation Classes. `VoiceElementData` is described in more detail in the following sections.
- The interaction category of the activity log is used to record the activity of callers within the VoiceXML pages produced by voice elements. Since this activity occurs on the voice browser, the only way to record this in the VXML Server logs is to store the appropriate information within the VoiceXML and submit it back to VXML Server. Unified CVP defines an interaction logging convention for recording caller behavior within the VoiceXML. By conforming to this convention, the developer ensures that the activity log will contain the detail expected for application testing and reporting.

  While interaction logging is not required in a custom voice element, not performing logging will reveal nothing about what the callers did within the element. See the section entitled Interaction Logging for more information. Description of the activity log is available in the User Guide for Cisco Unified CVP VXML Server and Unified Call Studio.
- Try to produce VoiceXML that conforms to the VoiceXML specifications, understanding that using browser-specific functionality will prompt that element to function correctly only on that browser.
- Throw an `ElementException` where appropriate so an error logger event can be thrown. Exceptions should indicate the voice element encountered a situation which prevents it from doing its assigned task. Non-fatal exceptions should be caught within the voice element (and possibly linked to exit states) and warnings placed in the error log.
- Do not end the call in the voice element manually using the `<exit>` or `<disconnect>` tags. Instead, exit the voice element with a specific exit state that can then be linked to a end-call element in the call flow. If it must be done, use `<disconnect>` so that VXML Server can detect the end-call and run the on end call action. Using `<exit>` would cause VXML Server to not know the calls had ended, using up an VXML Server port until it times out on its own. This is obviously not a desirable situation.
- A voice element can add any Java object to scratch space via the `setScratchData()` method in `VoiceElementData`. Scratch data is used for voice element internal uses only since the scratch space is cleared when the voice element ends, even if it is revisited in the same call. Additionally, the data in the scratch space is stored in the session managed by VXML Server. To minimize performance issues, do not place large Java objects in the scratch space.

VoiceElementBase Methods

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

Method to Run

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

This method must be run for voice elements. The arguments to the method are:

- **vxml** – The `VMain` object is the container VFC object in which all VoiceXML content is added (by adding other VFC objects to it). The desired VoiceXML page to produce must be assembled using the VFCs and added to this object. If no VFCs are added, an incomplete VoiceXML page will be produced, causing the voice browser to encounter an error. Simply add all `VForm` objects to this object. VXML Server will take care of the rest.

Note The form you wish to be visited first must be named *start*. **This is very important!**

- **params** – The `params` object is a `Hashtable` that contains all HTTP arguments passed by the voice browser through VXML Server. For example, if the voice element produces a VoiceXML page with a variable named *dataToCollect* that is then included in a submit argument list, the `params Hashtable` will contain an entry with the key *dataToCollect* and the value as a `String`. To access it, the developer would write:

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

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

The `String` return value of the method must refer to the real name of the voice element's exit state. The real name of the exit state must match one of the names given in the `getExitStates()` configuration method. Since voice elements can span multiple VoiceXML pages, returning `null` indicates that the voice element is not done and the method should be visited again when the voice browser sends its next request to VXML Server. Only when the method returns a proper exit state will VXML Server follow the application call flow and proceed to the next element. The method throws an `ElementException` that is used to indicate an error occurred within the method that prevented the voice element from doing its assigned task. The error message will be logged in the application's error log and the error element (if applicable) will be visited. This method additionally throws a `VException`, which is generated by incorrectly configured VFCs and does not need to be thrown explicitly by the element method itself.

- `Striung getSubmitURL()`

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

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

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

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

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

  See Voice Foundation Classes for a full description of the VFCs.

- `VoiceElementResult createSubVoiceElement(VoiceElementBasemainVoiceElement, VMain vxml, Hashtable params, VoiceElementData data)`

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

  This method is intended to be called *from an instance of a sub-element*, not the super-element. The super-element first creates an instance of the sub-element and then calls this method from that instance. This runs a sub-element within the super-element using the same context. The sub-element runs, it reads from a configuration object and creates VFC classes. The difference is that the super-element can take what the sub-element produced and modify it or add to it.

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

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

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

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

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

- The element data namespace is shared across the super-element and all sub-elements. Again, VXML Server uses a separate namespace for each element it is aware of, which means the super-element. The super-element must ensure that element data created by one sub-element does not overwrite the element data created by another sub-element.
- The super-element must handle all the *internal call flow* between sub-elements. Essentially, the super-element must perform the tasks VXML Server usually performs to handle the call flow between sub-elements. This means keeping track of state, managing the exit states of sub-elements, and knowing when to leave the super-element with its own exit states.

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

Configuration Methods

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

- `HashMap getAudioGroups()`

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

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

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

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

- `String[] getAudioGroupDisplayOrder()`

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

Interaction Logging

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

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

Once a VoiceXML page is visited and the `audium_vxmlLog` variable is filled with content, it must be passed back to VXML Server for parsing to place in the activity log. This means that every submit back to VXML Server must include this variable as an argument. If the voice element uses the `getSubmitURL()` method to obtain the submit URL, `audium_vxmlLog` must be explicitly added to the argument list in order to log correctly. Another advantage of using the `getSubmitVAction()` methods are that they already add this variable to the submit.

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

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

Where:

- ACTION is the name of the action. There are currently seven different actions that can be logged (the names are exactly as listed, all lowercase):
    - **audio_group** – This is used to indicate that the caller heard an audio group play. The value is the name of the audio group.
    - **inputmode** – This is used to report how the caller entered their data, whether by voice or by DTMF keypresses. The value is stored in the inputmode VoiceXML shadow variable.
    - **utterance** –This is used to report the utterance as recorded by the speech recognition engine (available using a VoiceXML shadow variable). The value is the actual utterance.

    - **interpretation** – This is used to report the interpretation as recorded by the speech recognition engine. The value is the actual interpretation.
    - **confidence** – This is used to report the confidence as recorded by the speech recognition engine (available using a VoiceXML shadow variable). The value is the confidence value.
    - **nomatch** – This is used to indicate the caller entered the wrong information, incurring a nomatch event. The value is the count of the nomatch event.
    - **noinput** – This is used to indicate the caller entered nothing, incurring a noinput event. The value is the count of the noinput event.

- VALUE is the value (description) to put in the log.
- ELAPSED is the number of milliseconds since the VoiceXML page was entered. This is required in order to keep an accurate timestamp in the activity log. Luckily, the VXML Server-generated root document provides a Javascript function named `application.getElapsedTime(START_TIME)` that returns the number of milliseconds elapsed since the time specified in `START_TIME`. A VoiceXML variable is declared in the root document that holds the time the VoiceXML page was entered and should be passed as input to this method. The variable name is `audium_element_start_time_millisecs`, though just as with `audium_vxmlLog`, a Java constant defined in `VoiceElementBase` named `ELEMENT_START_TIME_MILLISECS` should be used to refer to this variable.

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

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

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

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

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

Note The `audium_vxmlLog` and a`udium_element_start_time_millisecs` variables are not mentioned by name, the `VXML_LOG_VARIABLE_NAME` and `ELEMENT_START_TIME_MILLISECS` Java constants are used instead. Where Java constants are used, they must appear outside the double quotes defining the string.

Note `pref` is expected to be a valid `VPreference` object.

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

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

and the `VAction` object would be configured like:

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

See Element Specifications for Cisco Unified CVP VXML Server and Unified Call Studio for details about the different logs VXML Server records and the data that can appear in the logs.

Remote Execution

For remote execution of the Voice Elements, the **Remote Execution** checkbox is added in the **General Settings** tab.

On selecting the checkbox, the default URI is auto-populated as follows:

For HTTP and RPC call, `remote://system/?classurl=<fully_qualified_java_path>`

Where `remote://system` indicates that the configurations will be fetched from the **Remote Url Settings** property tab which is application-specific.

Note If a direct remote server URI is provided, then that IP:Port will be used and not fetched from the **Remote Url Settings** property tab.

For example:
`http://<IP>:<Port>/<target_path>/?classurl=<fully_qualified_java_path>`

Add Events in the Element Configuration to handle any particular or general exception gracefully.