Build VoiceXML Applications from a Windows Desktop

a report by
John Calvit

Cambridge VoiceTech on behalf of VoiceXML Italian Users Group

Speech recognition will soon allow anyone to access the Internet from any mobile device using vocal orders, through phones or even in the car. Voice XML was first introduced in 1999 and has achieved wide acceptance in the industry.

The specification builds on existing mark-up languages such as HTML, XML and WML, and the introduction and development of the voice Web gives service providers a simple and cost-effective means of deploying innovative speech applications for end-users. Companies will soon be able to use this technology to satisfy their users' need to be informed anywhere, anytime.

An open-source standard – VoiceXML – allows voice applications to be developed in the same manner as Web applications, and connect to a server and database. The availability of differentiated, advanced voice services based on Internet Protocols (IPs) will enable service providers to strengthen links with the Internet and bring interactivity closer still.

This article will introduce the essentials for building a VoiceXML application. Following is the code for the obligatory 'Hello World!' application, an application that indicates that it is the user's first VoiceXML application.

```
1. <?xml version="1.0"?>
2. <vxml version="1.0" mode="TTS">
3. <!–
4. This example shows the basic
   requirements for a
5. VoiceXML Application.
6. –>
7. <form>
8. <block>
9. <prompt>Hello World! This is my first
   VoiceXML
10. Application</prompt>
11. </block>
12. </form>
13. </vxml>
```

Tags, Elements and Attributes

The first two lines of the application:

```
<?xml version="1.0"?>
<vxml version="1.0" mode="TTS">
```

are mandatory for any VoiceXML application. Line 1 is important as it indicates to people and XML parsers that this is indeed an XML file. Line 2 is an opening tag that identifies the remainder of the file as a VoiceXML file. In XML and VoiceXML, all inform-ation is enclosed by a pair of tags or is embedded in a tag. A tag is identified by its angle brackets (< and >). Typically, an opening tag, such as in line 2, is paired with a closing tag. In this case, the closing tag is on line 13:

```
<vxml version="1.0" mode="TTS">
...
</vxml>
```

Closing tags are identified by the backslash (/). The opening and closing tags, taken together, form an element. The VoiceXML opening tag on line 2 contains two attributes: `version` and `mode`. Attributes are predefined variables that apply to an element. In XML and, by extension, VoiceXML, all attributes are declared as strings but they do not necessarily remain so. The `version` attribute is assigned the value `1.0`. The `mode` attribute identifies the mode in which the interpreting browser is to present the VoiceXML. In this case, the mode is text to speech (`TTS`). Therefore, the browser will convert voice prompts to sounds using the available TTS engine.

Comments

Lines 3–6 begin with <!– and end with –>. These indicate a comment block: essentially, notes left by the developer. Comments are usually ignored by the browser. However, certain symbols, such as the greater-than sign (>), can fool the browser into thinking the comment is complete. Therefore, care must be exercised when using standard XML comments.

Prompts

The real action (what little there is of it) happens inside the block element that opens on line 8 and

John Calvit is Product Manager at Cambridge VoiceTech (VoiceXML Italian Users Group) with over a decade of software experience as a technical writer, editor and software developer. Mr Calvit's writing has won awards from the Society of Technical Communication for design and content.

closes on line 11. This is where the prompt element at lines 9 and 10 is executed. A prompt element outputs synthesised speech and prerecorded audio to the user. (Note that the prompt text is actually between the prompt tags in character data form.)

Element Hierarchy

VoiceXML is a hierarchical language. Each element type is allowed only certain elements as 'children' and other elements as 'parents'. The prompt on lines 9 and 10 is contained within a block element that, in turn, is contained in a form element.

```
<form>
<block>
<prompt>Hello World! This is my first
VoiceXML
Application</prompt>
</block>
</form>
```

In VoiceXML, most procedures take place within form elements or menu elements. Therefore, to prompt the application to do anything, a form or menu element must be included. However, the form element cannot have a prompt element as a child, but the block element can be a child of the form element. Block elements provide execution for non-interactive code.

A prompt element is non-interactive (i.e. there is no user input) and therefore is allowed in the block element. Thus, following the permissible hierarchy, the prompt element is nested within the block element, and the block element within the form element. This element hierarchy may seem somewhat confusing but it does provide a number of advantages, including referencing and variable scoping.

Lesson – Menus and Fields

An example is used here of building a scheduling application for conference rooms. The conference room scheduler will be able to incorporate the date, time and number of people and then inform the attendees of the scheduled meeting via e-mail.

Before examining the conference room features, the menu elements must be considered. A menu element combines a prompt with input capture and a 'go-to' or 'switch' capability to route execution of the application. When the phrase, "Say or press one to [do something], say or press two to [do something else]" is heard, it indicates a menu.

This lesson will demonstrate how a menu captures user input and then evaluates that input to route the control. Also included are examples of how to deal with instances when users fail to input valid choices or do not provide any input at all.

```
1.  <?xml version="1.0"?>
2.  <vxml version="1.0">
3.  <!–
4.  This example shows the basic
    requirements for a
5.  menu in a VoiceXML Application
6.  –>
7.
8.  <menu>
9.  <prompt>Welcome to the Cambridge
    Conference Room
10. Scheduling
    System.<enumerate/></prompt>
11. <choice nextitem = "databaseForm
    ">database
    </choice>
12. <choice nextitem =" withoutForm">
    without </choice>
```

Line 8 is the opening tag for the menu. The menu starts with the prompt: "Welcome to the Cambridge Conference Room Scheduling System." However, there is an additional element: the enumerate element. The enumerate element causes a number option to be added automatically to each verbal choice. Therefore, the prompt output appears in the browser as:

```
CON: Welcome to the Cambridge Conference
Room Scheduling System.
CON: For database, say database or press 1.
CON: For without, say without or press 2.
```

(Note the unusual syntax of the enumerate element, which is a 'self-closing tag'.) Looking closely at the choice option, we see:

```
<choicenextitem = "databaseForm">
database </choice>
```

The character data between the tags – database – is added to the prompt text. The nextitem attribute – databaseForm – acts, in effect, as a go-to label. If the response to the prompt matches either database or without, the application control will jump to the databaseForm form or the withoutForm form, respectively. Similarly, pressing 1 or 2 will also jump to the appropriate forms.

Self-closing Tags

Also worth mentioning is the enumerate element on line 10. Unlike the elements seen so far, enumerate has no closing tag, instead appearing as:

```
<enumerate/>
```

The slash (/) before the closing angle bracket acts as the closing tag. This is common shorthand for many elements that contain no children, such as exit,

disconnect, enumerate and reprompt, etc. However, it cannot be used on all childless elements and not on elements with children.

No Match and No Input

The user may answer with an utterance that is not a choice, press an inappropriate number or simply not reply at all. Menus must be designed to deal with these contingencies.

VoiceXML provides two elements for handling such situations: the nomatch element and the noinput element. The nomatch element is triggered in response to sounds that do not fit any choice in a menu. The noinput element is triggered if a certain time elapses (typically, five seconds) without the user making a response of any kind.

The code as follows continues the menu element started previously (the closing tag for the menu appears on line 36):

```
13. <!–
14. nomatch and noinput elements handle
    invalid user input
15. –>
16. <nomatch count=”1”>That was an invalid
    choice. Please try
17. again.
18. </nomatch>
19. <nomatch count = ”2”>That was an
    invalid choice. Please try
20. again.
21. </nomatch>
22. <nomatch count=”3”>That was your third
    try. The program will
23. now exit.
24. <goto nextitem=”end”/>
25. </nomatch>
26. <noinput count=”1”>No input was
    received. Please try
27. again.
28. </noinput>
29. <noinput count=”2”>No input was
    received. Please try
30. again.
31. </noinput>
32. <noinput count=”3”>That was third try.
    The program will
33. now exit.
34. <goto nextitem=”end”/>
35. </noinput>
36. </menu>
```

The nomatch and noinput element structures are almost identical. Each has a count attribute indicating the appropriate response for the occurrence. Between the opening and closing tags of the nomatch and noinput elements is the prompt text to be played to the user. Though this example shows three nomatch and three noinput elements, any number of each could be coded. A menu element should always have at least one nomatch and one noinput element. In the third nomatch and third noinput elements, the nextitem attribute appears again, this time in a goto element. Just as with the nextitem attribute in the choice elements, the nextitem attribute indicates that flow should jump to another form, this time called end.

Form Identifications

The menu choice at line 11 reads:

```
<choice nextitem = ”databaseForm”>
database </choice>
```

indicating a jump to the databaseForm. Below is the databaseForm form, which utilises the id attribute to set the form’s identification (ID). The form also prompts the user and sends control to the end form.

```
37. <form id=”databaseForm”>
38. <block>
39. <prompt>This is the form database.
40. You will now be sent to the next form
    without.</prompt>
41. <goto nextitem=”end”/>
42. </block>
43. </form>
44.
```

Input Fields and Grammar

One of the most important aspects of voice interactive applications is the ability to capture voice input. If there is a limited number of valid inputs, a grammar can be defined. A grammar is an enumerated set of acceptable responses. When the user makes a valid response, the value can be written to a field reliably and used later in a prompt. This is different fro, a menu choice, where the response acts essentially as a switch statement, immediately directing the flow of execution.

The code as follows will capture the name of the user in a field. If that name is part of the established grammar, the application will speak it back in verification.

```
45. <!–
46. In-line Grammars would need to be
    built based on your company
47. directory. You must enter one of the
    names listed in the
48. in-line grammar shown or replace the
    grammar with your own.
49. –>
50. <form id=”without”>
51. <field name=”you”>
```

```
52. <grammar>Casey|Eric|Jan|John</grammar>
53. <prompt>Please say or enter
aname</prompt>
54. <prompt>Your choices are: Casey,
    Eric, Jan or John.</prompt>
55. <nomatch count="1">I don't understand.
    The program will
56. now exit.
57. <goto nextitem="end"/>
58. </nomatch>
59. <noinput count="1">I have received no
    input. The program will
60. now exit.
61. <goto nextitem="end"/>
62. </noinput>
63. <filled>
64. <prompt>I heard<value expr = "you"/>
    </prompt>
65. </filled>
66. </field>
67. </form>
```

Line 50 is arrived at via the menu choice jump at line 12. The `field`, named you, is declared on line 51. The `grammar` is enumerated at line 52 within the grammar element. Each value is separated by a bar (|). The prompts at line 53 and 54 are played. If the user responds with one of the defined grammar values, the `field` is considered `filled` and moves to the filled element on line 63. This executes a `prompt` that repeats the user name using the value for the field. (Note that the browser does not play back a recording of the user's spoken response but, instead, in this case, synthesises the name.) The field can also be accessed as a variable.

If the user responds with anything besides a value from the grammar or does not respond at all, the `nomatch` element or `noinput` element moves execution to the `end` form.

Exit

The final form of the lesson simply provides a closing goodbye `prompt` and exits the application.

```
68. <form id="end">
69. <block>
70. <prompt>Thank you for using the
    Cambridge Conference Room
71. Scheduling System.</prompt>
72. <exit/>
73. </block>
74. </form>
75. </vxml>
```

Though the application would end on its own at line 75, an `exit` element at line 72 quits the application earlier. ■