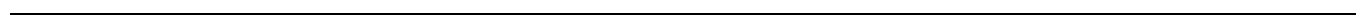


VoiceXML Tutorial



BeVocal, Inc.
685 Clyde Avenue
Mountain View, CA 94043

Part No. 520-0002-02

Copyright © 2005. BeVocal, Inc. All rights reserved.

Table of Contents

Preface	1
Prerequisites	1
Conventions	1
How to Use This Tutorial	2
Other Sources of Information	3
If You Need Help	3
1. Introduction	5
A Typical Application	5
Voice Access to the Web	6
First Steps	6
Application Development	6
Basics of VoiceXML	7
Basic Syntax	7
Header	8
Main Body	8
2. A Basic Dialog	11
Example Application.....	11
Initialization and Debugging Features	12
Initialization	12
Maintainer	12
Comments.....	12
Document Reload	13
Help and Error Handling	13
Forms and Fields	14
Block	14
Basic Field Format	14
Variables	15
Field Computation	16
Form Reset	17
Summary	17

3. Variables, Expressions, and JavaScript	19
JavaScript in VoiceXML	19
JavaScript Expressions in Attributes	19
Script Elements	20
Special Characters in JavaScript	20
Functions	21
Text Manipulation in Functions	23
More on Variables	23
Variable Scopes	23
Properties	24
Applications and Root Documents	25
4. Forms	27
Forms and Form Items	27
Fields	28
Field Types	28
Interpretation Order	29
Mixed-Initiative Forms	29
5. Prompts	31
Prerecorded Audio Data	31
Barge-in	32
Error Prompts	32
Reprompts	33
Tapered Prompts	34
Speech Timeouts	35
6. Interaction with Servers	37
Data Submission	37
Data in URIs	38
7. Other VoiceXML Elements	39
Links	39
Menus	40
DTMF in Menus	40
Automatic Enumeration of Choices	41
Next Document, Form, or Field	41
Exit	42
User Input Recorded on the Fly	42

8. Subdialogs	45
Transfer of Control	45
Subdialog Invocation	45
Value Return from a Subdialog	46
9. More on Grammars	47
Basic Grammar Rules	47
Grammar Slots	49
Grammars for Mixed-Initiative Forms	50
10. Design and Performance Considerations	53
File Retrieval and Caching	53
User-Friendliness	54
Maintainability and Debugging Aids	54
11. Example Code	55
Calculator	55
Factorial Function	58
Mixed-Initiative Form	59
Recording Audio	60
Subdialog—Main Dialog	61
Subdialog—The Subdialog	62



Welcome to the world of VoiceXML application development!

BeVocal Café provides software and services that make it easy to create interactive voice-based applications. Your customers, staff, or other users can access the application with an ordinary telephone. They can provide information by pressing touch-tone keys or simply by speaking.

To create an application, you write documents in the VoiceXML language. VoiceXML allows your document to access the Internet, to exchange information with web sites and other servers.

Prerequisites

Besides interacting with the user, a complete VoiceXML application usually accesses web pages or other Internet server functions. Consequently, to create a complete application, you will probably need some knowledge of HTML and related technologies.

To understand this tutorial, it will be helpful if you have already written some web pages using HTML, or if you have used any type of XML, or have any other programming background. For additional information, see some of the on-line resources listed below.

Conventions

Italic font is used for:

- Introducing terms that will be used throughout the document
- Emphasis

Bold font is used for headings.

Fixed width font is used for:

- Code examples
- Tags and attributes
- Values or text that must be typed as shown
- Filenames and pathnames

Italic fixed width font is used for:

- Variables
 - Prototypes or templates; what you actually type will be similar in format, but not the exact same characters as shown
-

How to Use This Tutorial

The first two chapters of this tutorial introduce VoiceXML. A new application developer typically reads these chapters completely and in order.

- Chapter 1, “Introduction” gives an overview of the BeVocal Café and the BeVocal VoiceXML language. It uses a very simple document to illustrate the basic principles of VoiceXML.
- Chapter 2, “A Basic Dialog” fills in more VoiceXML details to build a complete, useful document.

The rest of this tutorial consists of a “cookbook” of VoiceXML techniques. Application developers typically do not read these chapters from start to finish. Read these chapters in any order that seems most useful for your intended application.

- Chapter 3, “Variables, Expressions, and JavaScript” gives details on types of data manipulation that are available in VoiceXML.
- Chapter 4, “Forms” discusses forms, the most common dialog element. It describes types of data that forms can collect and advanced features for controlling form execution.
- Chapter 5, “Prompts” discusses prompts, the method for creating output for the user to hear. It describes features such as using recorded audio files for output and handling errors gracefully.
- Chapter 6, “Interaction with Servers” describes ways to pass data from VoiceXML documents to web servers.
- Chapter 7, “Other VoiceXML Elements” discusses VoiceXML elements, such as links and menus, that provide additional features for your documents.
- Chapter 8, “Subdialogs” describes how to write VoiceXML documents that interact with each other.
- Chapter 9, “More on Grammars” discusses how to construct grammars for recognizing various types of speech and DTMF input.
- Chapter 10, “Design and Performance Considerations” gives a collection of techniques and tips to make your applications efficient and easy to use.
- Chapter 11, “Example Code” contains the text of the long examples used in other chapters.

Other Sources of Information

Besides this tutorial, BeVocal Café provides a variety of other resources, including:

- *Getting Started with Café* provides an introduction to using the BeVocal Café to develop your VoiceXML applications.
- *VoiceXML Programmer's Guide* provides detailed information about the VoiceXML language.
- *VoiceXML Samples* provides several sample documents, similar to the ones in this tutorial.
- *Grammar Reference* provides detailed information on how to construct grammars for speech recognition.
- *JavaScript Quick Reference* provides a summary of the JavaScript features that are available with BeVocal VoiceXML.
- *The Audio Library* contains audio files with commonly used prerecorded prompts and other sounds that you can use in your applications.

For more detailed information on VoiceXML, see the World Wide Web Consortium's on-line specification at <http://www.w3.org/TR/voicexml20/>.

VoiceXML is based on the XML standard. For information on XML, a variety of resources are available at these sites:

<http://www.xml.org/>

<http://www.w3.org/XML/>

If You Need Help

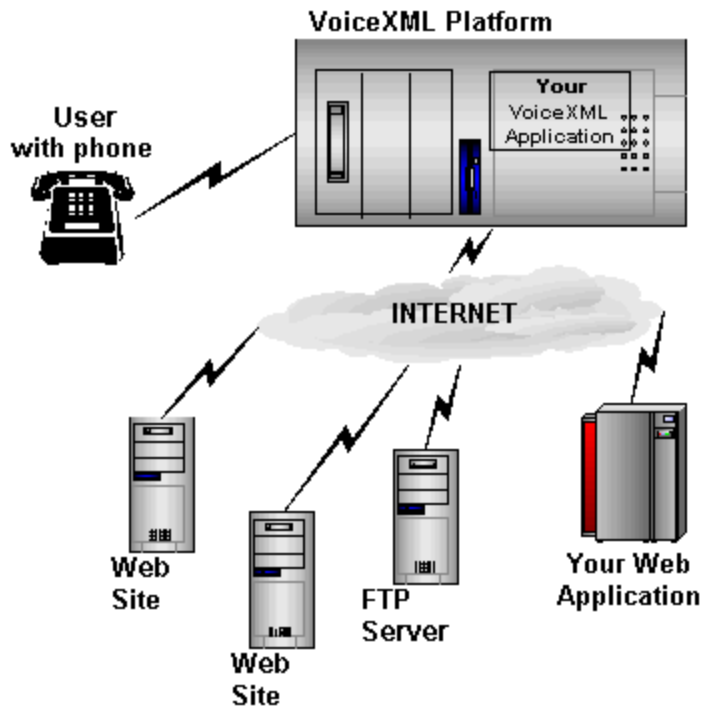
Support for this tutorial is available by email at the *BeVocal Café news groups*.

This chapter gives an overview of how VoiceXML applications work and how to write VoiceXML documents. It describes:

- A Typical Application
- Voice Access to the Web
- First Steps
- Application Development
- Basics of VoiceXML

A Typical Application

The following diagram shows a typical VoiceXML application at work:



A user connects with your application by dialing the appropriate phone number. The VoiceXML interpreter answers the call and starts executing your VoiceXML document. Under the document's control, the interpreter may perform actions such as:

- Sending vocal prompts, messages, or other audio material (such as music or sound effects) to the user.
 - Accepting numeric input that the user enters by DTMF (telephone key tone) signals.
 - Accepting voice input and recognizing the words.
-

- Accepting voice input and simply recording it, without trying to recognize any words.
- Sending the user's information to a web site or other Internet server.
- Receiving information from the Internet and passing it to the user.

In addition, VoiceXML documents can perform programming functions such as arithmetic and text manipulation. This allows a document to check the validity of the user's input. Also, a user's session need not be a simple sequence that runs the same way every time. The document may include "if-then-else" decision making and other complex structures.

Voice Access to the Web

A complete VoiceXML application usually requires some resources outside the server hosting the VoiceXML application. You may need a web site or other server that is able to accept and process information from users and to send reply information back to them.

This is an advantage of using VoiceXML. A VoiceXML document can access the World Wide Web, acting as a sort of voice-controlled browser. It can send information to web servers and convey the reply to the user. Also, by using VoiceXML as a "front end" to a web application, you minimize the amount of VoiceXML coding that is required. The rest of your application can be based on familiar protocols, such as HTTP and CGI, for which powerful programming tools are available.

First Steps

To begin developing and deploying BeVocal VoiceXML applications, you must register with BeVocal Café and arrange:

- A username and password for your Café account.
- The phone number that users will dial to access your application.
- The location where your VoiceXML document files will be stored.

To get started, you can set up an account yourself over the Web, by using the BeVocal Café web site at <http://cafe.bevocal.com>.

You may store your document and related files on the Café server. In that case, you can use the File Management tool to upload, view, update, check, and delete them. However, you may also keep these files on your own server, if that makes setup and file management easier. In that case, you specify the URI of the main document by entering it in a form on the File Management page.

Application Development

Because VoiceXML documents consist of plain text, you can create and edit them with any simple text editor such as Notepad on Windows, or EMACS or VI on UNIX systems. You place the documents on your web server, along with related files such as audio data. You test them by calling the appropriate phone number and interacting with them.

BeVocal Café provides several tools to help you develop your applications. Among these tools are the following:

- The *File Management* page lets you upload, view, and delete files that you have stored on the Café server.
- The *VoiceXML Checker* can check the syntax of your documents. Although it does not guarantee that they will run correctly, it does check for correct syntax, properly nested elements, missing quote characters, and so on.
- The *Vocal Player* lets you replay calls that have been made to your application.
- The *Log Browser* shows you detailed log files of calls to your application, showing what files were fetched, values were computed, and so on.
- The *Trace Tool* lets you monitor a call while a user is on the phone.

These and other tools are all available at the BeVocal Café web site.

For an introduction to using the BeVocal Café to develop your application, see *Getting Started with Café*.

Basics of VoiceXML

At this point, let's look at a simple VoiceXML document (the line numbers are not part of the document):

```

<?xml version="1.0"?> 1
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN" 2
    "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">
<vxml version="2.0" 3
    xmlns="http://www.w3.org/2001/vxml"
    xml:lang="en-US">
    <form> 4
        <field name="selection"> 5
            <prompt> 6
                Please choose News, Weather, or Sports. 7
            </prompt> 8
            <grammar type="application/x-nuance-gsl"> 9
                [ news weather sports ] 10
            </grammar> 11
        </field> 12
        <block> 13
            <submit next="select.jsp"/> 14
        </block> 15
    </form> 16
</vxml> 17

```

This document causes the request, “Please choose News, Weather, or Sports,” to be spoken to the user. Then it accepts the user's response and passes the response to another document—actually a server-side script named `select.jsp`—which presumably provides the service that the user selected.

Basic Syntax

This document illustrates a number of basic points about VoiceXML. It consists of plain text as well as *tags*, which are keywords or expressions enclosed by the angle bracket (< and >) characters. In this example, everything is a tag except for lines 7 and 10.

A tag may have *attributes* inside the angle brackets. Each attribute consists of a *name* and a *value*, separated by an equal (=) sign; the value must be enclosed in quotes. Examples are the `version` attribute in line 3 and the `name` attribute in line 5.

Most of the tags in this example are used in pairs. The `<vxml>` tag is matched by `</vxml>`; tags such as `<form>` and `<block>` have similar counterparts. Tags that are used in pairs are called *containers*; they contain content, such as text or other tags, between the paired tags.

Some tags are used singly, rather than in pairs; these are called *stand-alone* or *empty* tags. Empty tags do not have any content (text or other tags), although they may have attributes. The only one in this example is the `<submit... />` tag. Notice that in an empty tag, there is a slash (/) just before the closing `>` character.

Caution: Compared to HTML, VoiceXML is much stricter about using correct syntax. If you have used HTML, you may be used to taking shortcuts, such as writing attribute values without quotes or omitting the ending tag of a container. In VoiceXML, you must use proper syntax in all documents.

The term *element* is sometimes used in talking about languages such as HTML and VoiceXML. An element is either:

- A stand-alone tag, including its attributes, if any; or
- A container tag, including the start tag and its attributes, as well as the matching end tag, and any other text or tags contained between them.

If an element contains another element, it may be referred to as the *parent* element of the one it contains. Conversely, a contained element is said to be a *child* element of its container.

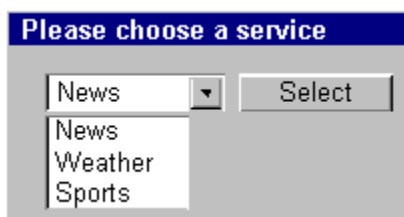
Header

Lines 1, 2, and 3 are required at the beginning of all VoiceXML documents. For now, always use the lines just as shown above. These lines work with standard VoiceXML (as described in this tutorial), but also work if you use any of the extensions to the language that are available in BeVocal VoiceXML. If you use only standard VoiceXML and none of the extensions, you can use different lines. For more information, see the *VoiceXML Programmer's Guide*.

The last line (17) is required at the end of all documents; it matches the `<vxml>` tag in line 3.

Main Body

After the header information comes the main body of the VoiceXML document. Typically, the body contains at least one `<form>` tag. The `<form>` tag defines a VoiceXML form, which is a section of a document that can interact with the user. This is the VoiceXML equivalent of an HTML form. For instance, you could write HTML to display a form in a web page:



In HTML, you would use the `<form>`, `<select>`, and `<input>` tags to create this form with a selection box and a button. In VoiceXML, you use `<form>`, `<field>`, and `<submit>` in a very similar structure.

A form usually contains *fields* which accept pieces of information from the user. In this simple example, the form has a single field named `selection` (lines 5-12) that stores the user's choice.

Inside the field is a `<prompt>` element (lines 6-8), which contains a line of plain text. The VoiceXML interpreter will use its text-to-speech (TTS) translation engine to read this text and send the resulting audio to the user. As we'll see later, instead of relying only on the TTS engine, you can also use `<audio>` tags to play prerecorded audio files.

After the `<prompt>` is a `<grammar>` element (lines 9-11) that defines the three responses the interpreter will accept for this field. Other forms of the `<grammar>` tag allow you to access more complex grammars

stored in separate files. Grammars can define spoken words or DTMF code sequences; you can set up a document to accept either type of input. VoiceXML allows you to write grammars using several formats; this tutorial uses the Nuance GSL format (as indicated by the `type` attribute of the `<grammar>` tag). For more on grammars, see Chapter 9, “More on Grammars” in this tutorial and the *Grammar Reference*.

After the interpreter sends the prompt to the user, it waits for the user to respond and tries to match the response to the grammar. If the response matches, the interpreter sets the field's `selection` variable to the response. This value is then available throughout the rest of the form.

Finally, the form ends with a `<block>` element containing a `<submit>` tag (lines 13-15). These lines transfer control to `submit.jsp`, passing the `selection` variable for use by that script.

In the first chapter, we looked at the fundamentals of VoiceXML as they applied to a very simple script with only one field. In this chapter, we take a closer look at interacting with the user—how to make a voice-response application that is powerful and easy to use.

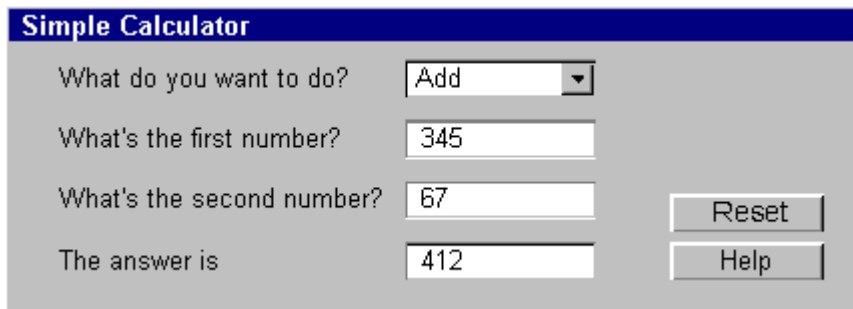
This chapter describes:

- Example Application
- Initialization and Debugging Features
- Forms and Fields
- Summary

Example Application

Here we'll look at something more like a real-world application. We'll introduce a larger document that accepts several inputs from the user, and also provides some error handling. This document will function as a voice-operated calculator.

The user names an operation (add, subtract, multiply, or divide) and then says two numbers; the script replies with the result. As with the first example, it may be helpful to compare this to an HTML form on a web page, which might look about like this:



The image shows a web form titled "Simple Calculator" with a blue header. The form contains four rows of input fields and two buttons. The first row is "What do you want to do?" with a dropdown menu showing "Add". The second row is "What's the first number?" with a text input field containing "345". The third row is "What's the second number?" with a text input field containing "67". The fourth row is "The answer is" with a text input field containing "412". To the right of the input fields are two buttons: "Reset" and "Help".

A VoiceXML equivalent to this could be made with a `<form>` containing three `<field>` elements. But to make it easy to use, we'll add some additional features that function similarly to the **Reset** and **Help** buttons.

This document is longer than our first example; we'll examine it one part at a time. To see the entire document in a separate window, click *here*. (This example is also listed in Chapter 11, "Example Code".)

Initialization and Debugging Features

Initialization

Our calculator starts with the usual standard elements:

```
<?xml version="1.0" ?>

<!DOCTYPE vxml
  PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
  "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0"
  xmlns="http://www.w3.org/2001/vxml"
  xml:lang="en-US"
>
```

Maintainer

Next, we have some code that is helpful for debugging and managing a document. First, the `<maintainer>` meta-tag:

```
<!--
  <meta name="maintainer" content="you@yourcompany.com"/>
-->
```

Just for a moment, let's ignore the `<--` and `-->` lines. This `<meta>` tag has the name `maintainer` and its content is an email address. The BeVocal VoiceXML interpreter will send a message to this address each time the document is executed, containing the log of that call, as an XML file. This allows you to keep track of your application's activity. For a description of the XML file, see *Log Access Service*. Of course, if your document is being used by hundreds of callers every day, you will get a lot of email...which brings us back to the `<--` and `-->`.

Comments

VoiceXML comments use the form you are familiar with from HTML. Comments start with the characters:

```
<!--
```

and end with:

```
-->
```

Anything between these delimiters is ignored by the interpreter. Comments can be used to put descriptive or historical information into a script, such as:

```
<!-- Test script by Mike Rowe, October, 2000 -->
```

Also, comments can be used to disable a section of a document without actually removing it from the file. For instance, you may use the `maintainer` meta-tag while you are developing a document. When you are ready to deploy it, instead of deleting the tag, you simply comment it out as shown above. Now you can leave the tag in place; the next time you do some work on the document, you can reactivate the tag by simply removing the comment delimiters.

Caution: Do not use strings of consecutive dash characters (`---`) inside comments.

Document Reload

Next, we'll show you a piece of code that is helpful for debugging a new document. It allows you to completely reload and restart it at any time, without having to hang up the phone and call in again.

```
<link
  maxage="0"
  next="http://yourcompany.com/this.vxml">
<grammar type="application/x-nuance-gsl">
  [
    (bevocal reload)
    (dtmf-star dtmf-star dtmf-star)
  ]
</grammar>
</link>
```

A `<link>` tag is like a hyperlink on a web page—it defines an option that is always active while the interpreter is executing the element that contains the link. In this case, the link is a child of the `<vxml>` element; that means it is active throughout the whole document.

The `next` attribute specifies the document to run when the link is activated. We have specified the URI of this document itself; when the link is activated, it has the effect of restarting the document from scratch, as if you had just dialed in.

The `maxage="0"` attribute ensures that the interpreter will always get the latest version of a document, rather than re-using a saved copy; this is probably necessary for debugging a new document. (For more on caching, see “File Retrieval and Caching” on page 53.)

The `<grammar>` tag contained in the `<link>` tag defines two ways that the link can be activated—by speaking the words “BeVocal reload” or by pressing the telephone’s * key 3 times.

Help and Error Handling

As the last part of our document’s initialization, we’ll include some tags, called *event handlers*, that provide help and error handling for user input. First:

```
<noinput>
  I'm sorry. I didn't hear anything.
  <reprompt/>
</noinput>
```

`<noinput>` specifies what the document should do if the user fails to provide any voice or keypad input. In this case, the tag specifies a text message and uses the `<reprompt>` tag to cause the interpreter to repeat the `<prompt>` for the current form field.

```
<nomatch>
  I didn't get that.
  <reprompt/>
</nomatch>
```

`<nomatch>` is similar to `<noinput>`; it specifies what to do if the user’s input doesn’t match any active grammar.

```
<property name="universals" value="all" />
<help>
  I'm sorry. There's no help available here.
</help>
```

A *universal* command is one that is available anywhere in your document. VoiceXML has a small set of predefined universals that you can use. The `<property>` element says to turn on these predefined universals.

For now, the most interesting of these universal commands is "help". Once you have turned on the help universal, you use the `<help>` element to specify what to do if the user asks for help. In this case, the message is not very helpful! But this is the default help message for the entire document. As we'll see shortly, you can create specific `<help>` messages for various parts of your dialog.

Forms and Fields

Now that the initial setup tags are written, we can begin constructing the form that controls the dialog.

Block

We start with the `<form>` tag, followed by a `<block>` containing an opening message:

```
<form>
  <block>
    This is the BeVocal calculator.
  </block>
```

In general, the `<block>` tag is used to contain a section of executable content that does not expect to interact with the user. For example, you can use it, as here, to play a message that doesn't expect a response. Or, you can use it to contain some program logic, such as the `<submit>` tag that ends the example in Chapter 1, "Introduction".

When a `<block>` tag contains simply text, as in this example, the interpreter treats it as shorthand for having included a `<prompt>` tag, for example:

```
<block>
  <prompt>
    This is the BeVocal calculator.
  </prompt>
</block>
```

Note: Most examples in this tutorial will use prompts that are written as plain text, to be played by the interpreter's TTS engine. This is very convenient for development, but the sound quality is not as high as a recorded human voice. When you prepare an application for use by the public, you should record all the necessary prompts and convert your script to play them by using the `<audio>` tag. (See "Prerecorded Audio Data" on page 31 for details.)

Basic Field Format

Now, let's write the first field, where the user chooses the type of calculation:

```
<field name="op">
  <prompt>
    Choose add, subtract, multiply, or divide.
  </prompt>
  <grammar type="application/x-nuance-gsl">
    [add subtract multiply divide]
  </grammar>
  <help>
    Please say what you want to do. <reprompt/>
  </help>
  <filled>
    <prompt>
      Okay, let's <value expr="op"/> two numbers.
    </prompt>
```

```

    </filled>
</field>

```

This field is a bit more complex than the one in our first example. It shows the primary parts of a field—prompts to play, grammars to use for recognizing a user’s utterances, and actions to perform on successful recognition.

This field has the name `op`, which can be used in VoiceXML expressions to reference the field’s *input* value, that is, what the user selects. The field contains a `<prompt>` tag to tell the user what’s expected and a `<grammar>` tag to listen for the user to say the operation he wants to perform. The grammar, a set of four words enclosed in square brackets, specifies that any one of these words is an acceptable value for this field.

Next is a `<help>` tag; this provides a specific message that the user will hear if he requests help for this field. Part of the help message (supplied by the `<reprompt>` tag) repeats the field’s prompt. You can also add `<noinput>` and `<nomatch>` tags for each individual field, if that helps to make your script more user-friendly.

The last part of the field is a `<filled>` tag. This specifies the action to take once the field is “filled in”; that is, once the user has provided a valid input. In this case, the document plays a message that confirms the user’s choice. The `<value>` tag is used to include the input value in the message by referencing the field name, `op`.

Let’s go on to the next field, which will hold the first number for the calculation:

```

<field name="a" type="number">
  <prompt>
    Whats the first number?
  </prompt>
  <help>
    Please say a number.
    This number will be used as the first operand.
  </help>
  <filled>
    <prompt> <value expr="a"/> </prompt>
  </filled>
</field>

```

This field has the name `a`. It also has a `type` attribute that defines it as a numeric field. This is a useful shortcut—it means that we do not need to specify a grammar for this field. The VoiceXML interpreter has a built-in grammar for numbers. The field contains `<prompt>`, `<help>`, and `<filled>` tags that act like the ones described for the `op` field. Notice this time the help prompt does not repeat the initial prompt.

Variables

Next, we need a field to accept the second number for the calculation. The `<filled>` tag in this field is the one that does the calculation and reports the answer to the user.

To have the `<filled>` element do this work, though, we need to include one more tag before the field:

```

<var name="result"/>

```

The `<var>` tag is used to declare a *variable*. VoiceXML variables can hold numbers, text, and several other types of data. As we’ve seen, every field has an associated variable, based on its `name` attribute. Because we have declared fields for the numbers to use for the calculation, they already have associated variables. However, since the script will compute the result, there is no field for it; we must explicitly declare a variable to hold the result.

Field Computation

With that done, we can write the code for the last field. It's going to be a long one, so let's start with the first few tags:

```
<field name="b" type="number">
  <prompt> And the second number? </prompt>
  <help>
    Please say a number.
    This number will be used as the second operand.
  </help>
  <filled>
    <prompt> <value expr="b"/> Okay. </prompt>
```

This field is named `b`, and its type is `number`, like the previous field. It has a `<help>` message, and a `<filled>` tag that confirms the user's entry. After that, things will get more complex.

We need to write code that actually chooses the type of operation to perform, does the calculation, and reports the result. The `<filled>` element is going to need some more content.

```
<if cond="op=='add'">
  <assign name="result" expr="Number(a) + Number(b)"/>
  <prompt>
    <value expr="a"/> plus <value expr="b"/>
    equals <value expr="result"/>
  </prompt>
```

The `<if>` tag is used for decision-making. Its `cond` attribute specifies the condition to be tested. If the condition is true, the code following the `<if>` will be executed. If the condition is false, the interpreter will skip over the following text and tags, until it encounters an `<else>`, `<elseif>`, or closing `</if>` tag. The `cond` value, `op=='add'`, is an expression that is true if the `op` field contains the word `add`.

If the condition is true, the interpreter will execute the `<assign>` tag, which computes the expression `Number(a) + Number(b)` and places the sum of the two numbers in the `result` variable (specified by the `name` attribute). You might have been tempted to write the expression as `a + b`; however, the `+` symbol can indicate string concatenation as well as addition. Using the `Number()` functions guarantees the arguments are numbers and not strings. If a user tries to add 30 and 14, the result should be 44, not 3014!

The `<assign>` is followed by a `<prompt>` to report the result to the user. Again, the `<value>` tag is used to include the values of both input numbers, as well as the result value, in the prompt.

That takes care of addition. Next, we need another tag to handle subtraction:

```
<elseif cond="op=='subtract'"/>
  <assign name="result" expr="a - b"/>
  <prompt>
    <value expr="a"/> minus <value expr="b"/>
    equals <value expr="result"/>
  </prompt>
```

These tags are very similar to the ones for addition. Instead of `<if>`, we started with `<elseif>` because we are building a series of choices, only one of which can be executed each time the script is run. In the `<assign>`, we can write `a - b` without using `Number()`, because the `-` symbol only has one possible meaning; there is no need to clarify it.

The tags to handle multiplication are largely the same:

```
<elseif cond="op=='multiply'"/>
  <assign name="result" expr="a * b"/>
  <prompt>
    <value expr="a"/> times <value expr="b"/>
```

```
    equals <value expr="result"/>
  </prompt>
```

Finally come the tags to handle division:

```
<else/>
  <assign name="result" expr="a / b"/>
  <prompt>
    <value expr="a"/> divided by <value expr="b"/>
    equals <value expr="result"/>
  </prompt>
</if>
```

Here we have used `<else/>` instead of `<elseif>`. We have ruled out all the other choices; only division is left. After the prompt is the `</if>` tag that closes the entire `<if>...<elseif>...<elseif>...<else/>` sequence.

Form Reset

At this point, we have just about finished building our dialog; there are just a few “housekeeping” items to take care of:

```
<clear/>
```

The `<clear>` tag is used to reset the form to its initial state; that is, it clears the values of all the fields in the form. When appropriate, you can also use the `<clear>` tag to reset only some of the variables and fields. Resetting the fields causes execution to resume at the top of the form, allowing the user to make another calculation.

Finally, we have tags to close the `<filled>` element, the field that contains it, the entire form, and the VoiceXML document:

```
    </filled>
  </field>
</form>
</vxml>
```

Summary

This chapter has introduced the following main points:

- Use comments to document and maintain your script.
- Use the `maintainer` meta-tag and a “reload” link to help with debugging.
- Use `<help>`, `<noinput>`, and `<nomatch>` to guide your users.
- Use `<filled>` fields to take action when the user provides input to them.
- Declare variables with `<var>`, perform calculations with `<assign>`, and place results into prompts with `<value>`.
- Make decisions with `<if>`, `<elseif>`, and `<else/>`.
- Reset a form with `<clear/>`.

In the calculator script example, you saw some basic uses of variables and expressions, such as:

```
<var name="result"/>
...
<assign name="result" expr="a - b"/>
```

VoiceXML uses the JavaScript language (also called ECMAScript) for manipulating data.

This chapter discusses:

- JavaScript in VoiceXML
- Functions
- Text Manipulation in Functions
- More on Variables
- Applications and Root Documents

JavaScript in VoiceXML

JavaScript is very powerful—it provides math and text-manipulating functions, as well as advanced programming features.

- For complete information about JavaScript, a variety of resources are available on the Web. For example, you can download the *ECMAScript Language Specification*.
- For a list of the specific features supported by the BeVocal VoiceXML implementation of JavaScript, see the *JavaScript Quick Reference*.

There are two ways to use JavaScript in VoiceXML:

- In attributes of VoiceXML tags.
- Inside a `<script>... </script>` container.

We'll discuss the simpler way first.

JavaScript Expressions in Attributes

VoiceXML has very few programming features of its own; JavaScript is, in a sense, the “native” programming language. For that reason, VoiceXML has many attributes that let you use JavaScript expressions directly. For example, the `expr` attribute of tags such as `<assign>`, `<field>`, and `<param>`, or the `cond` attribute of tags such as `<if>` and `<field>` all expect JavaScript expressions as their values. We already saw this tag:

```
<assign name="result" expr="a - b"/>
```

The value of the `expr` attribute can actually be any valid JavaScript expression. For instance, if we wanted to calculate square roots, we could take advantage of JavaScript's math functions by writing:

```
<assign name="result" expr="Math.sqrt(a)"/>
```

Script Elements

Using the `<script>` tag allows you to go beyond expressions and write JavaScript functions and other executable statements. To put JavaScript code in a document, you enclose the script with `<script>` and `</script>` tags as you would for HTML. Because VoiceXML is based on XML, it has various restrictions on where special characters can be used. For safety, it is helpful to always enclose the code with `<![CDATA[and]]>` inside the `<script>` tag, as in:

```
<SCRIPT>
  <![CDATA[
    ... your JavaScript code ...
  ]]>
</SCRIPT>
```

Without the `<![CDATA[and]]>`, you would need to replace some special characters with their entity equivalents, such as `<` and `>`; instead of the `<` and `>` characters. You'd also need to know exactly which characters to replace. Using CDATA simplifies all this for you.

A `<script>` may be handled in one of two ways, depending on where it is placed in the document:

- A `<script>` may be at the top level, that is, contained by the `<vxml>` element. In that case, it is executed only once, when the document is first loaded into the VoiceXML interpreter. This is a good place to put statements that define functions or initialize variables.
- A `<script>` may be in an executable element: `<filled>`, `<if>`, `<block>`, or an event handler such as `<help>`. A script located in one of these is executed every time the containing element is executed.

Special Characters in JavaScript

There's one problem putting JavaScript code in your VoiceXML documents. There are 3 characters ("`<`", "`>`", and "`&`") that have special meaning in both JavaScript and in VoiceXML and that have *different* meaning in the 2 languages:

Character	VoiceXML Meaning	JavaScript Meaning
<code><</code>	First character of a tag, as in <code><form></code> or <code></form></code>	Arithmetic less-than operator
<code>></code>	Last character of a tag	Arithmetic greater-than operator
<code>&</code>	Start of an "entity" (a way of providing some esoteric information)	Part of several operators concerning conjunction (AND), as in <code>&</code> or <code>&&</code>

Whenever JavaScript inside your VoiceXML document contains one of these characters, you must somehow "escape" the character—that is, tell the VoiceXML interpreter to use the JavaScript meaning for the character, not the VoiceXML meaning.

For a JavaScript expression that is the value of an attribute, you *must* escape the characters using the corresponding *escape sequence*. For example, assume you wanted to have the following JavaScript expression as the value of the `cond` attribute of the `<if>` tag:

```
balance < minbalance
```

You would write the tag as follows:

```
<if cond="balance &lt; minbalance">
```

Notice that the & in this expression is the VoiceXML special character! It says to use the JavaScript less-than operator.

On the other hand, if you're writing JavaScript code inside a `<script>` element, the code will be easier to read if you wrap all of the code in a CDATA section. For example, a script containing the factorial function would be written as follows:

```
<script>
  <![CDATA [
    function factorial(n) {
      return (n <= 1) ? 1 : n * factorial(n-1);
    }
  ]]>
</script>
```

The appropriate escape sequences are as follows:

Character	Escape Sequence
<	<
>	>
&	&

Functions

Let's look at another type of calculator. This document will use JavaScript to define a function that computes factorials. A VoiceXML form accepts a number from the user; the JavaScript function computes the answer.

To simplify the code in the tutorial, we'll omit some items, such as `<noinput>` and debugging aids, that should be included in most real-world documents. See Chapter 2, "A Basic Dialog" for details on these.

As in that chapter, we'll present the document in sections. To see the entire document in a separate window, click [here](#). (This example is also listed in Chapter 11, "Example Code".)

First comes the required initialization code:

```
<?xml version="1.0" ?>

<!DOCTYPE vxml
  PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
  "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0"
  xmlns="http://www.w3.org/2001/vxml"
  xml:lang="en-US">
```

After the usual `<?xml>` and `<vxml>` tags is the `<script>` element that defines the function:

```
<script>
  <![CDATA [
    function factorial(n) {
      return (n <= 1) ? 1 : n * factorial(n-1);
    }
  ]]>
</script>
```

As mentioned before, the JavaScript code is enclosed by the delimiters:

```
<![CDATA [  
and:  
]]>
```

Remember that putting the `<script>` element directly as a child of the `<vxml>` element assures that it executes once, when the document is initially entered. You don't want your functions to be redefined every time the form is entered.

The function itself is called `factorial`. It accepts a single argument, `n`; it uses the `return` statement to return a value back to the expression that calls it.

The next section of code defines the form. First, it defines a variable named `lastresult`, which will be used to store a message about previous uses of the `factorial` function. This variable is given an initial string value. It then declares the field as a `number` and plays a prompt to request input from the user:

```
<form id="foo">  
  <var name="lastresult"  
    expr="'This is the first factorial you have computed'" />  
  <field name="n" type="number">  
    <prompt> Say a number. </prompt>
```

Next comes a `<filled>` item. It first reports the result, using the `expr` attribute of a `<value>` tag to call the `factorial()` function and also reporting the `lastresult` message:

```
<filled>  
  <prompt> The factorial of <value expr="n"/> is  
    <value expr="factorial(n)"/>  
    <value expr="lastresult"/>  
  </prompt>
```

The `<filled>` tag then uses the `<assign>` tag to update the value of the `lastresult` variable with a new message, indicating what factorial you just computed. Finally, we see a new use of the `<clear>` tag, using the `namelist` attribute to clear only the variable `n` and not the variable `lastresult`. This assures that the next time through the form, we'll get the new factorial message.

```
    <assign name="lastresult"  
      expr="'The last factorial you computed was for '  
        + n" />  
    <clear namelist="n"/>  
  </filled>  
</field>  
</form>  
</vxml>
```

An interaction with this application would be something like this:

Application: Say a number.

User: 3.

Application: The factorial of 3 is 6. This is the first factorial you have computed.

Application: Say a number.

User: 5.

Application: The factorial of 5 is 120. The last factorial you computed was for 3.

Application: Say a number.

Text Manipulation in Functions

JavaScript functions can operate on pieces of text (sometimes called *strings* in computer jargon) as well as numbers. This can help simplify your documents. If your application uses a lot of recorded audio for prompts, you may have a large number of tags that look something like the following, with only the filename part being different in each one:

```
<audio
  src="http://www.yourcompany.com/appName/audio/thankyou.wav">
```

It would be more convenient if you could just write:

```
<audio expr="say(thankyou)">
```

You can do this by defining a JavaScript function such as:

```
function say(filename) {
  return(
    "http://www.yourcompany.com/appName/audio/"
    + filename + ".wav")
}
```

In JavaScript, the `+` operator can be used to combine or *concatenate* two strings. Using a function like `say()` in all the documents of a large application can save you a lot of typing and editing, which also cuts down on debugging time.

Note that, to use a JavaScript function, the `<audio>` tag must use the `expr` attribute, rather than `src`. There may be a slight performance trade-off here, since caching cannot be done when `expr` is used (see “File Retrieval and Caching” on page 53).

More on Variables

Variables can be created, or *declared*, in two ways:

- You can declare them explicitly with the `<var>` tag, as we did the `lastresult` variable in the factorial example.
- When you declare a field in a form, this has the effect of creating a variable with the name you specify in the `<field>` tag. For example, in the calculator document in Chapter 2, “A Basic Dialog”, the input variables named `a` and `b` are used in JavaScript expressions that calculate the result.

Note: In a document, the tag that declares a variable must be located **before** any expressions that use the variable.

As we saw in the factorial example, you may want to declare a variable and also assign it a value. You can do this in one step by writing an expression in the `expr` attribute of the `<var>` tag, such as:

```
<var name="squarerootoftwo" expr="Math.sqrt(2)">
```

The following sections describe some additional features of VoiceXML variables.

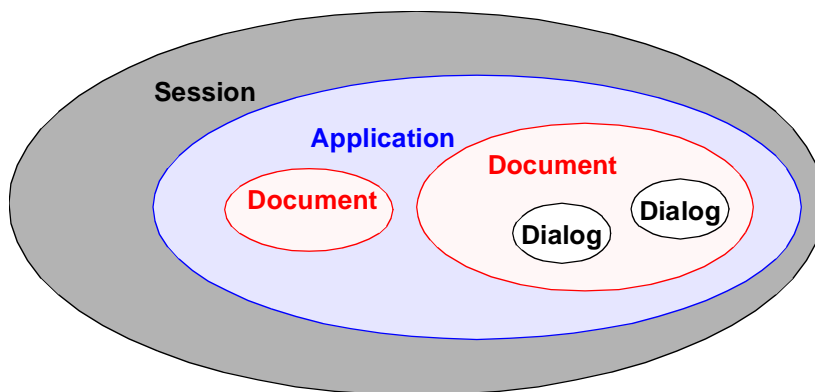
Variable Scopes

A variable’s *scope* is the range or “territory” in which it can be used. Scopes are useful because they allow one name, like `a` or `b` or `zipcode`, to be used for different purposes in different parts of an application.

However, scoping may result in errors in your document, if you try and use a variable in a location outside its scope. It's important to understand how they work.

- Variables declared inside a `<block>`, `<filled>`, or event handler element (such as `<noinput>`) are *local* to that element. They cannot be used in expressions outside that element. This is called *anonymous scope*.
- Variables declared inside a dialog (form or menu), but outside of any `<block>`, `<filled>`, or event handler elements, are local to the dialog. They can be used in expressions anywhere inside the dialog element, but not elsewhere in the document. This is called *dialog scope*.
- Variables declared in a document, but outside of any dialogs, are local to the entire document. They can be used in expressions anywhere in the document. This is called *document scope*.
- Variables declared in the application root document (see “Applications and Root Documents” on page 25) are local to the application. When one document loads another, these variables are preserved as long as the new document is part of the same application. This is called *application scope*.
- Session variables are created as soon as a user dials into your service. These variables are set by the VoiceXML interpreter and are described in Chapter 12, “Properties” of the *VoiceXML Programmer's Guide*. Session variables are preserved throughout the call, even if the application's root document changes. This is called *session scope*.

Notice how the various scopes form a nested structure:



Under these rules, you can have two or more variables that have the same name. If their scopes do not overlap, it's easy for the VoiceXML interpreter to tell which expressions refer to which variables.

If an expression uses a name in a place where two variables' scopes do overlap, the interpreter uses the *more local* variable, that is, the one in the smaller scope. For instance, suppose you have an application variable named `zipcode` and a local variable named `zipcode` in a particular document. If you write `zipcode` in an expression inside the document scope, the interpreter will use the document's variable, not the application's variable. If you want to override this behavior, you can explicitly reference the application variable by writing `application.zipcode`.

Similarly, you can use the prefixes `session.`, `document.`, and `dialog.` to specify those scopes. There is no prefix for the anonymous scope—because it is the most local of all scopes, there is no need for a way to explicitly refer to it.

Properties

VoiceXML `properties` are predefined variables that control various features of the interpreter. You set them with the `<property>` tag, as in:

```
<property name="timeout" value="5s">
```

which sets the value of the `timeout` property to 5 seconds. This property controls how long the interpreter will wait for user input before it executes a `<noinput>` event handler.

The scope of a `<property>` tag is the element that contains it. For instance, suppose you use the above tag to set a `timeout` of 5 seconds for an entire document. You can also write another `<property>` tag inside a form, or even inside a single field, to define a different `timeout` value for that particular element.

Applications and Root Documents

Sometimes a complete VoiceXML application is large enough that it will be advisable to divide it into several separate documents. Normally, when one document transfers control to another, all variables are destroyed and the new document starts from scratch. However, it's often helpful to have some variables that are preserved from one document to the next.

This is where the concept of an *application* comes in. A VoiceXML application is a set of documents that are written to work together. The steps to create such a set of documents are as follows:

- Create an *application root document*, where you define application variables and other information to be shared between documents.
- Specify the root document in all the other documents, by writing its name or URI as the value of the `application` attribute of the `<vxml>` tag.

When the VoiceXML interpreter loads a document, if it specifies a root document, and the root is not already loaded, the interpreter loads the root. Once the root is loaded, it remains active until either the session is complete (the user hangs up) or a document with a different root is loaded.

When one document transfers control to another that is part of the same application, all variables declared in the root document are preserved. This provides a convenient way to share information across documents.

An application can also define shared grammars. For instance, a `<link>` tag in the root document can define a help request or other function, which is then active throughout all documents in the application. Also, JavaScript functions that are defined in the root document are available to all other documents in the application.

In the calculator example in Chapter 2, “A Basic Dialog”, you saw a basic VoiceXML form that collected three input values from the user. The form is the basic mechanism for controlling interaction with the user. In this chapter, we will take a closer look at forms. We’ll see how to collect different types of data and also more advanced ways of controlling how the user passes through the dialog.

This chapter discusses:

- Forms and Form Items
- Fields
- Interpretation Order
- Mixed-Initiative Forms

Forms and Form Items

There are two types of forms. One type, called a *directed* form, is simpler; the forms we have looked at already are of this type. When executing a directed form, the interpreter processes the fields in sequence, from the first to the last. There are some exceptions that we’ll discuss below, but in general, directed forms guide the user through a series of prompts and responses.

The other type of form, called a *mixed-initiative* form, is more flexible. In a mixed-initiative form, the interpreter can fill in several fields from a single user input. See “Mixed-Initiative Forms” on page 29.

A VoiceXML form is contained by the `<form>` and `</form>` tags. Inside the form, you may write other elements called *form items*. There are two kinds of form items—*input items* and *control items*. Input items include `<field>`, as well as several other elements that are like `<field>` in that they can hold a value for use in expressions or for submission to a server. The input items include:

- `<field>` defines fields for user input. Fields may contain other elements in a nested structure.
- `<record>` makes an audio recording of the user’s input.
- `<transfer>` transfers the telephone caller to another destination, such as placing a call to a customer service representative.
- `<subdialog>` transfers control to another document, which can then return control to the first document using `<return>`.

There are two control items. They do not hold values, but they perform functions that control the interpretation of the form or the execution of the entire document. The control items include:

- `<block>` defines a section of executable code, such as audio prompts or calculations.
 - `<initial>` defines the starting point for a mixed-initiative form (see “Mixed-Initiative Forms” on page 29).
-

Fields

Fields are an important part of VoiceXML dialogs. Field elements can contain other elements, including:

- `<prompt>` elements that give instructions to the user.
- `<grammar>`, `<dtmf>`, and `<option>` elements that specify what user input is acceptable for the field.
- `<filled>` elements that specify actions to take when the field is *filled in*; that is, when the user has provided a valid value for it.

Fields can be used in expressions, much like variables declared with `<var>`. Every field has a `name` attribute that specifies the variable name to use in expressions.

Field Types

Fields may have a `type` attribute that can help the interpreter process the user's speech input. We saw `type="number"` already; the other available types are:

- `boolean` defines a field whose value is either 0 (false) or 1 (true). Users can say "Yes" or press 1 on the DTMF keypad to generate a true result; they can say "No" or press 2 to generate a false result.
- `digits` defines a field to accept a string of digits, such as a credit card number. The user must speak each digit individually; for example, "three nine four" is valid, but "three hundred ninety four" is not. (A `number` field would accept "three hundred ninety four.")
- `date` defines a field to accept a calendar date. Users may say the date in various common forms or enter it on the DTMF keypad as a 4-digit year, 2-digit month, and 2-digit day, in that order. The field's value is a string of 8 digits or question mark (?) characters; the ? characters indicate digits for which the user did not provide a value.
- `time` defines a field to accept a time of day, in hours and minutes. The field's value is a string of four digits, followed by a fifth character which is one of:
 - `a` for AM (morning).
 - `p` for PM (evening).
 - `h` to indicate a time in 24-hour format.
 - `?` if the user's input was ambiguous.(For DTMF input, there is no way to specify AM/PM.)
- `currency` defines a field to accept a price or other monetary amount. The field's value is a string of the form `CCCddd.d`, where `CCC` is a 3-character currency indicator and `ddd.d` is a monetary value. For DTMF input, the star (*) key can be used to indicate a decimal point.
- `phone` defines a field to accept a telephone number. The field's value is a string of digits, but may contain an `x` to represent an extension. For DTMF input, the star (*) key can be used to indicate an `x`.

If a field doesn't have a `type` attribute, it must have a grammar to determine what user input is acceptable. The grammar also specifies a string value to be placed as the value of the input variable.

Interpretation Order

Normally, the VoiceXML interpreter processes a form by starting with the first item and proceeding forward. However, there are several ways to control the interpretation order, giving your dialogs more flexibility.

All form items have two attributes that can be used to control their execution, `cond` and `expr`.

`expr` allows you to specify an initial value for the form item. The value may be any JavaScript expression. The interpreter only executes form items whose value is undefined. Consequently, if you use `expr` to give an initial value to a form item, the interpreter will ignore that item. To “reactivate” the item at a later time, you can remove the value with `<clear>`.

Note: Remember that if your JavaScript expression contains any of the characters “<”, “>”, or “&”, that character must be replaced with the corresponding escape sequence “<”, “>”, or “&”; see “Special Characters in JavaScript” on page 20.

`cond` allows you to specify a *guard condition* for an item. The value may be any JavaScript expression. The condition is considered false if its value is 0 or the empty string (“”) and true for all other values. The interpreter only executes form items whose guard condition has a true value. Consequently, if you assign a guard condition to a form item, the interpreter can decide “on the fly” whether or not to execute the item, based on whatever values the condition checks.

Here’s a simple example of how to use `cond`. This form first asks the user to choose a service. If the choice is Weather, the form asks for a zip code, in order to determine the user’s location:

```
<form>
  <field name="selection">
    <prompt> Please say News, Weather, or Sports. </prompt>
    <grammar type="application/x-nuance-gsl">
      [ news weather sports ]
    </grammar>
  </field>

  <field name="zipcode" type="digits"
    cond="selection == 'weather'">
    <prompt> What's your zip code? </prompt>
  </field>
</form>
```

The `zipcode` field has a guard condition consisting of the JavaScript expression `selection == 'weather'` (note that single quotes are used around `'weather'` to distinguish them from the double quotes that enclose the whole condition). When the interpreter gets to the `zipcode` field, it will only process the field if the user has requested the weather service.

Mixed-Initiative Forms

As mentioned earlier, a mixed-initiative form is one that allows a single user input to provide values for several fields. This allows the form to process natural language, rather than directing the user through a rigid sequence of questions and answers. To be mixed-initiative, a form must have:

- An `<initial>` element that is defined for the entire form.
- A grammar that is also defined for the entire form; that is, the `<grammar>` tag must be inside the form, but not in any input items.

In Chapter 1, “Introduction”, we looked at a very simple form that allowed the user to choose one of three services:

```
<form>
  <field name="selection">
    <prompt>
      Please choose News, Weather, or Sports.
    </prompt>
    <grammar type="application/x-nuance-gsl">
      [ news weather sports ]
    </grammar>
  </field>
  <block>
    <submit next="select.jsp"/>
  </block>
</form>
```

Suppose we want to enhance our application by allowing a user to ask for two or three services in a single request. In this case, instead of having a single field named `selection`, we’ll use three separate fields called `news`, `weather`, and `sports`. Each will be a `boolean` field that is set to `true` if the user asks for that service. We will use a mixed-initiative form, letting the user ask for one, two, or all three services in a single request.

To see the document in a separate window, click *here*. (This example is also listed in Chapter 11, “Example Code”.)

In addition to being larger than the previous form, this one also has a significantly different structure. It starts with a `<grammar>` tag that defines a grammar that can recognize a variety of user inputs, such as:

- “News, weather.”
- “Tell me sports and news.”
- “I’d like news and sports and weather.”

This Nuance GSL grammar is described in detail in “Grammars for Mixed-Initiative Forms” on page 50. Its connection to the form is specified by these lines:

```
news      { <news true> }
weather   { <weather true> }
sports    { <sports true> }
```

The portions enclosed by `{` and `}` are commands that store the value `true` into special variables called *slots*. The slots are named `news`, `weather`, and `sports`.

Next is the `<inital>` element, which is always executed first when the interpreter processes a mixed-initiative form. It contains an opening prompt and could also perform other actions such as initializing variables.

Next are three `boolean` fields named `news`, `weather`, and `sports`. Because these names match the slot names defined in the grammar, the interpreter automatically maps the slots to the fields. That way the commands in the grammar can store values into the fields.

Finally there’s a `<filled>` element, which is executed after the interpreter finishes processing the user input. Notice that this element has its `mode` attribute set to `any`. By default, the `<filled>` element executes only if all of the fields in the form have been filled. In this case, that would mean until the user asked for all 3 services, which would defeat the point of the form. To have the element execute as soon as any of the form fields is filled, you set the `mode` attribute to `any`, as here.

In this example, the `<filled>` element does not submit the data to a server. It just plays a prompt to confirm what it recognized and then uses `<clear/>` to reset the form, so that you can try another input.

This chapter describes some additional features of using prompts:

- Prerecorded Audio Data
- Barge-in
- Error Prompts
- Reprompts
- Tapered Prompts
- Speech Timeouts

Prerecorded Audio Data

The ability to play and record audio data makes it possible to write more professional-sounding applications. A finished application should use recorded audio for all prompts and other messages played to the user. The VoiceXML interpreter can translate text to speech, which is very handy for developing and debugging an application. However, the higher quality of recorded audio is preferable for real-world users.

In addition to audio prompts that you record, BeVocal Café provides a library of audio files containing commonly used voice prompts and sound effects that you can use in your applications. The library is located on the Café web site, at:

<http://cafe.bevocal.com/libraries/audio/>

You use the `<audio>` tag inside a `<prompt>` tag to play a recorded audio file to the user. The file may contain any recorded sound you want, such as speech, music, “beep” tones, or other sound effects. You use the `src` attribute to specify the URI of the audio file, as in:

```
<prompt>
  <audio
    src="http://www.yourcompany.com/audio/welcome.wav">
    Welcome to the telephone message service.
  </audio>
</prompt>
```

Here the `<prompt>` tag contains an `<audio>` tag, which specifies a file name and also contains some text as a back-up message. To execute this prompt, the interpreter will first try to fetch the specified audio file. If the file is available, the interpreter plays it and ignores the text. If the file does not exist or is not available for some other reason, the interpreter will use the text-to-speech engine to read the text to the user.

It is a good idea to write all `<prompt>` tags in this form, even if you have not recorded the audio files yet. They will run correctly using the backup text; you can add the audio files at any time, without modifying the document.

Sometimes, it is useful to use JavaScript expressions to construct URIs. In such situations, you use `expr` instead of `src`, as in:

```
<audio
  expr="'http://www.yourcompany.com/audio/'
      + userlanguage + '/welcome.wav'"
/>
```

Here, the URI is constructed with an expression that combines two strings with a variable named `userlanguage`. This example could be used to make your application multilingual. Presumably, the URI is based on a file structure where there is a complete set of audio prompts for each language, with each set of prompts in a separate folder. The variable `userlanguage` would be set up early in the dialog, with the name of the appropriate folder.

For another example of using JavaScript in `<audio>` tags, see “Text Manipulation in Functions” on page 23.

Note: As seen above, the `<audio>` tag may be either a container or a stand-alone tag. When used stand-alone, remember to include the slash character (`/`) before the trailing angle bracket (`>`).

Barge-in

Normally, while the VoiceXML interpreter is playing a prompt, the user does not have to wait for the prompt to finish before speaking. Instead, he can “*barge in*” and speak a response at any time. When the interpreter detects incoming speech, it interrupts the prompt.

However, there are times when you want to disable barging in, to ensure that the user hears a complete prompt. This could be useful for messages such as legal notices, advertisements, or when the user appears to be having trouble.

There are two ways to control barge-in. A `<prompt>` tag may include the `bargein` attribute, with a value of `true` or `false`. If the attribute is `true`, barge-in is enabled. If it is `false`, the user must listen to the entire prompt, as in:

```
<prompt bargein="false">
  <audio src="advertisement.wav">
</prompt>
```

You can also control the barge-in behavior of all prompts in a document, by using the `<property>` tag to set the `bargein` property. For instance, the tag:

```
<property name="bargein" value="false"/>
```

will disable barge-in for all prompts in a document. The default value of this property is `true`; that is, barge-in is normally enabled for all prompts.

Error Prompts

There are two standard user error types your application should always handle:

- A *noinput* error occurs when the user is quiet for “too long” after a prompt.
- A *nomatch* error occurs when the user says something that the interpreter cannot recognize (or “match”) the utterance against any active grammars.

VoiceXML provides three tags to use when handling noinput and nomatch errors—`<noinput>`, `<nomatch>`, and `<reprompt>`. When a noinput or nomatch error occurs, the VoiceXML interpreter searches for an active `<noinput>` or `<nomatch>` element. These tags allow you to prompt the user again, while the interpreter waits for another utterance.

As we saw in “Help and Error Handling” on page 13, you can define noinput and nomatch handlers at the top level of your document. For example, you could have:

```
<noinput>Sorry, I didn't hear you. </noinput>
<nomatch>Sorry, I didn't understand you. </nomatch>
```

Note, however, that as written, these handlers contain no information specific to where the error occurred. If the user says something not in the active grammar, the interpreter simply says “Sorry, I didn’t understand you” and waits for the user to remember the question and respond appropriately. This is where the `<reprompt>` tag comes in.

Reprompts

If you include the empty tag `<reprompt/>` in a noinput or nomatch handler, the interpreter repeats the prompt in the field’s `<prompt>` tag. For example, assume you had the following snippet:

```
<noinput>
  Sorry, I didn't hear you.
  <reprompt/>
</noinput>
<nomatch>
  Sorry, I didn't understand you.
  <reprompt/>
</nomatch>

<field name="op">
  <prompt>Choose add, subtract, multiply, or divide.</prompt>
  <grammar type="application/x-nuance-gsl">
    [add subtract multiply divide]
  </grammar>
...

```

Then a user interaction might go like this:

Application: Choose add, subtract, multiply, or divide.
User: Truncate.
Application: Sorry, I didn’t understand you. Choose add, subtract, multiply, or divide.
User: <silence>
Application: Sorry, I didn’t hear you. Choose add, subtract, multiply, or divide.
User: Exponentiate.
Application: Sorry, I didn’t understand you. Choose add, subtract, multiply, or divide.

Tapered Prompts

So far, the error prompts we've seen aren't very helpful. At most, they've said some generic error message and then repeated a prompt the user has already heard. One of the most important ways you can improve the usability of your application is to give more appropriate error messages when things start to go wrong.

One helpful metric for how much trouble the user is having is how many times errors have occurred in a single state. The VoiceXML interpreter keeps track of three separate counters for this:

- How many times the `<prompt>` for an input item, `<initial>` element, or menu has played since the form or menu was entered.
- How many times a `<noinput>` handler has played for each of these.
- How many times a `<nomatch>` handler has played for each of these.

You can use these counters to *taper* your prompts, by using the `count` attribute of the `<prompt>`, `<noinput>`, and `<nomatch>` tags. As a simple example, a shopping application might use a prompt such as:

```
<noinput>Sorry, I didn't hear you. <reprompt/></noinput>
<nomatch>Sorry, I didn't understand you. <reprompt/></nomatch>
```

```
<field name="itemnumber" type="digits">
  <prompt>
    Please enter the item number.
  </prompt>
  <prompt count="3">
    Please say the item number of the product you wish to buy,
    or enter it on the telephone keypad.
  </prompt>
  <prompt count="4">
    If you need help choosing an item, please enter zero.
    Otherwise, please say the item number of the product you
    wish to buy, or enter it on the telephone keypad.
  </prompt>
</field>
```

This field has three prompts, two of which have `count` values. When the user initially enters the field, the interpreter will use the prompt without the `count` (the shortest one). On the first error, the interpreter will play the appropriate "Sorry" prompt, followed by the prompt without the count, for example:

```
Sorry, I didn't hear you. Please enter the item number.
```

Next, the prompt counter reaches 3; the interpreter will play the appropriate "Sorry" prompt, followed by the prompt with that count, for example:

```
Sorry, I didn't understand you. Please say the item number
of the product you wish to buy, or enter it on the telephone keypad.
```

Finally, once the count reaches 4, the interpreter will play the longest prompt.

Because you can also use counts on the `nomatch` and `noinput` handlers, you can vary your error prompts even more. Consider this variation:

```
<noinput>Sorry, I didn't hear you. <reprompt/></noinput>
<nomatch>
  Sorry, I didn't understand you.
  <reprompt/>
</nomatch>
```

```

<nomatch count="2">
  I'm sorry, I still didn't understand you.
  <reprompt/>
</nomatch>
<nomatch count="3">
  I seem to be having a lot of trouble understanding.
  <reprompt/>
</nomatch>

<field name="itemnumber" type="digits">
  <prompt>
    Please enter the item number.
  </prompt>
  <prompt count="3">
    Please say the item number of the product you wish to buy,
    or enter it on the telephone keypad.
  </prompt>
  <prompt count="4">
    If you need help choosing an item, please enter zero.
    Otherwise, please say the item number of the product you
    wish to buy, or enter it on the telephone keypad.
  </prompt>
</field>

```

Now, at the same time that the context-specific error messages are getting more explicit, the more global nomatch handler message is also changing. With this variation, you might have the following interaction:

Application: Please enter the item number.

User: The red dress.

Application: Sorry, I didn't understand you. Please enter the item number.

User: A dress.

Application: I'm sorry, I still didn't understand you. Please say the item number of the product you wish to buy, or enter it on the telephone keypad

User: I want a dress.

Application: I seem to be having a lot of trouble understanding. If you need help choosing an item, please enter zero. Otherwise, please say the item number of the product you wish to buy, or enter it on the telephone keypad.

Speech Timeouts

If a user does not speak after hearing a prompt, the interpreter will generate a *timeout* and execute the `<noinput>` event handler, if there is one. You can control the amount of time that the interpreter waits before timing out with the `timeout` property. For instance, this tag sets the timeout to 10 seconds:

```
<property name="timeout" value="10">
```

For more information on properties, see “Properties” on page 24.

The calculator example from Chapter 2, “A Basic Dialog” is unusual in that it does not call any servers; it performs all interaction within the VoiceXML code. In practice, applications almost always require some sort of communication with a server in order to offer a useful service.

This chapter describes two ways in which a VoiceXML document can send information to contact a server—by using `<submit>` and by adding query data to a URI.

Alternatively, one document can pass data to another with `<subdialog>` and `<return>`, as described in Chapter 8, “Subdialogs”. In some cases, this can eliminate the need for server access.

The server may use a variety of different mechanisms, such as CGI, ASP, JSP, Miva Script, and so on. Basically, any server that can respond to HTTP requests from web users can perform equivalent functions for VoiceXML users.

This chapter describes:

- Data Submission
- Data in URIs

Data Submission

The normal way to pass user data to a server is with the `<submit>` tag. This tag allows you to generate GET or PUT transactions for an HTTP server. These transactions may include form fields or other data values. A `<submit>` acts much like the `<INPUT TYPE="SUBMIT">` tag in an HTML form. A typical use is:

```
<submit
  next="http://yourcompany.com/cgi-bin/respond.cgi"
  method="post"/>
```

You use this tag inside a form; it causes the interpreter to submit all the form fields to the specified server. The `next` attribute specifies the URI of the server. If you want to use a JavaScript expression to generate the URI, you can use the `expr` attribute instead of `next` (see below).

As is usual for HTTP, after a document submits a form, the server is expected to respond by sending a reply document. The VoiceXML interpreter receives and executes this document. Note that all variables, grammars, and so on, declared in the first document are destroyed when it executes a `<submit>`; they are not preserved and passed to the reply document by the server.

`<submit>` becomes more flexible with the `namelist` attribute, which allows you to specify exactly which values are submitted to the server, as in:

```
<submit
  next="http://yourcompany.com/cgi-bin/respond.cgi"
  method="post"
  namelist="name acctnumber password"
/>
```

The `namelist` attribute allows you to submit only part of a form, instead of submitting all the fields. It also allows you to submit other variables currently in scope, even if they are not the fields of any form.

Data in URIs

Some servers expect data to be passed to them in the query part of the URI, for example:

```
http://yourcompany.com/voice/login.jsp?user=Mary&account=123456
```

The query part consists of the `?` character followed by a series of values or, as in this case, a series of expressions of the form `name=value` separated by ampersand (`&`) characters. You can use JavaScript to create URIs like this for `<submit>`, `<link>`, and `<goto>` tags. For example:

```
<goto expr="'http://yourcompany.com/voice/login.jsp'  
  + '?user=' + username  
  + '&account=' + acctnum"/>
```

Here, the `expr` attribute contains a rather long JavaScript expression that concatenates the base URI with two parameters. The `user` parameter is given the value of the `username` variable, which could be a form field or any other type of variable. Similarly, the `account` parameter is given the value of the `acctnum` variable.

This chapter describes a number of VoiceXML elements that are helpful in the creation of advanced documents:

- Links
- Menus
- Next Document, Form, or Field
- Exit
- User Input Recorded on the Fly

Links

As mentioned in Chapter 2, “A Basic Dialog”, a VoiceXML `<link>` is similar to a hyperlink in an HTML document. A hyperlink is available whenever it is visible on the user’s screen; the user activates the hyperlink by clicking it. A `<link>` is available whenever the user is executing a part of a document that is in the link’s scope. The user activates the link by speaking or keying some input that is recognized by the link’s grammar. A link allows a user to “escape” from one document to another, without filling out additional form fields or taking other actions to reach the end of a document.

In Chapter 2, “A Basic Dialog”, we presented this link, which can be used to reload and restart a document for debugging:

```
<link maxage="0" next="http://yourcompany.com/this.vxml">
  <grammar type="application/x-nuance-gsl">
    [
      (bevocal reload)
      (dtmf-star dtmf-star dtmf-star)
    ]
  </grammar>
</link>
```

The `next` attribute specifies the document that will be loaded when the user activates the link; in this case, the URI specifies the current document, so that it will be reloaded. You can use `expr` instead of `next` if you want to use a JavaScript expression to create the URI.

Inside the link is a `<grammar>` tag that specifies what user input will activate the link. In this case, both a spoken phrase (“BeVocal reload”) and a DTMF sequence (pressing the * key three times) is accepted. See Chapter 9, “More on Grammars”.)

A link’s scope is the element that contains it. For instance, if you put a link inside a `<form>`, it will be available only while the interpreter is executing the VoiceXML code for that form. On the other hand, if a link is placed at the top level, such as just after the `<vxml>` tag, it will be available throughout the entire document.

Menus

The `<menu>` tag is a kind of shortcut for a form with only one field. It is a convenient way to write a dialog that asks the user to pick one option from a list. `<menu>` is often used for the top level or main menu of a complex service, as in:

```
<menu>
  <prompt>
    This is the main menu.
    Please choose a service: news, weather, or sports.
  </prompt>
  <choice next="news.vxml">
    news
  </choice>
  <choice next="weather.vxml">
    weather
  </choice>
  <choice next="sports.vxml">
    sports
  </choice>
</menu>
```

This menu contains a `<prompt>` and three `<choice>` elements. Each `<choice>` specifies a document to load in the `next` attribute, as well as a word or phrase that will select that document. In a real application, the `<choice>` tags would probably each contain elements to more precisely specify prompts and grammars for that choice.

DTMF in Menus

Menus can also use DTMF key tones instead of voice input. The above example can be changed to use the telephone's 6 key (which carries the letter N) for news, the 9 key (W) for weather, and the 7 key (S) for sports:

```
<menu>
  <prompt>
    This is the main menu.
    For News say 6; for Weather say 9; for Sports say 7.
  </prompt>
  <choice next="news.vxml">
    dtmf-6
  </choice>
  <choice next="weather.vxml">
    dtmf-9
  </choice>
  <choice next="sports.vxml">
    dtmf-7
  </choice>
</menu>
```

Note that the `<choice>` tags now contain the `dtmf-n` keyword to specify the required keypress.

Automatic Enumeration of Choices

You can make the original menu even more concise by adding the `<enumerate>` tag, which is used inside a prompt. For instance, you could change the `<prompt>` in that example to:

```
<prompt>
  This is the main menu.
  Please choose a service: <enumerate/>
</prompt>
```

When the VoiceXML interpreter executes this prompt, it will list each of the choices in the `<choice>` elements that follow. Thus, it will say, "This is the main menu. Please choose a service: news, weather, sports."

`<enumerate>` is a useful shortcut for longer menus. It has the advantage that, if you change or add menu choices, the prompt message changes along with them, automatically.

In a real application, `<enumerate>` might modify exactly what was said for each choice. For example, another way to get the DTMF version of the menu is with this code:

```
<menu>
  <prompt>
    This is the main menu.
    <enumerate>
      For <value expr="_prompt"/> say <value expr="_dtmf"/>
    </enumerate>
  </prompt>
  <choice next="news.vxml" dtmf="6">
    news
  </choice>
  <choice next="weather.vxml" dtmf="9">
    weather
  </choice>
  <choice next="sports.vxml" dtmf="7">
    sports
  </choice>
</menu>
```

Here, the `<enumerate>` element contains a template for how to say each choice. The template uses the special variables `_prompt` and `_dtmf`, which are set to the prompt and associated DTMF sequence for each choice.

Next Document, Form, or Field

As its name implies, the `<goto>` tag is used to make the interpreter "go" or "jump" from one place to another while executing your document. You specify the URI with the `next` attribute. If the destination of the jump is in another document, use a complete URI, as in:

```
<goto next="http://yourcompany.com/shopping/checkout.vxml"/>
```

If the destination is another dialog in the same document, write a tag such as:

```
<goto next="#shippingaddress"/>
```

In this case, the value of `next` is a # character followed by the name of the destination dialog. You give a name to a dialog by specifying it in the `id` attribute of the `<form>` or `<menu>` tag.

A third use for `<goto>` is to jump from one item to another inside a form. In this case, you write the tag with the `nextitem` attribute, instead of `next`, as in:

```
<goto nextitem="phonenumber"/>
```

The value of `nextitem` should be the form item's name, as specified by the value of its `name` attribute.

You can use a JavaScript expression in a `<goto>`. In that case, use the `expr` attribute instead of `next`, or the `expritem` attribute instead of `nextitem`.

Caution: Indiscriminate use of `<goto>` can turn a document into a complex “maze” that is difficult to debug and maintain. You should keep your document structures “clean” and straightforward when possible. (See “Maintainability and Debugging Aids” on page 54 for details.)

Exit

Normally, the VoiceXML interpreter processes a form sequentially, from start to end. Sometimes, though, you might want to terminate execution in mid-document. You can do this with the `<exit>` tag, as in:

```
<form>
  ...
  <field name="cont" type="boolean">
    <prompt> Do you want to continue? </prompt>
    <filled>
      <if cond="cont==1">
        ...
        <goto>, <submit>, or other processing
        ...
      <else/>
        <exit/>
      </if>
    </filled>
  </field>
</form>
```

A field like this could be used in any form, anywhere in a document. It asks whether the user wants to continue. If the user's response is “No”, the `<exit>` tag stops execution immediately and terminates the user's call.

Note: If `<exit>` is used inside a subdialog, it terminates the entire application, not just the subdialog. (Don't confuse `<exit>` with `<return>`.)

User Input Recorded on the Fly

Occasionally, you may want to record and store audio input from the user, for example, for voice mail and similar applications or to store what the user said to pass on to a human operator.

The VoiceXML `<record>` tag is similar to `<field>` in that it accepts input from the user. However, the value of a `<record>` element is the spoken audio itself; this is unlike `<field>`, where the interpreter processes the audio and extracts a value such as a number or string.

The `<record>` tag must be used inside a form. A typical use of `<record>` might look like this:

```
<record name="message" maxtime="10s" finalsilence="2500ms">
  <prompt> Please say your message. </prompt>
  <noinput>
    I didn't hear anything, please try again.
  </noinput>
</record>
```

Like most form items, the tag has a `name` attribute. (You can also use the `cond` attribute to assign a guard condition if necessary.) It has two other additional attributes:

- `maxtime` specifies a maximum length for the recorded message. The suffix `s` means the time is in seconds.
- `finalsilence` specifies the amount of silence that will signal the end of the recording. Here, the suffix `ms` indicates milliseconds (in this case, 2.5 seconds).

If you want to play back a recorded message, you use the `expr` attribute of the `<audio>` element. To play back the message recorded in the previous example, you could use this code:

```
<field name="confirm" type="boolean">
  <prompt> Your message is <audio expr="message"/>
  Do you want to keep it?
</prompt>
...
```

`<record>` is a container. Inside the element, you place tags such as `<prompt>`, as well as event handlers such as `<noinput>`.

Let's look at a complete form that uses `<record>` to collect a message. To see the document in a separate window, click [here](#). (This example is also listed in Chapter 11, "Example Code".)

The form contains the `<record>` element, as well as a `boolean` field for a yes/no confirmation. The form collects a message from the user and then asks if the user wants to keep it. If the answer is Yes, the document submits the audio data to a server. If the answer is No, the document clears the form; this causes it to re-execute the dialog, allowing the user to record a new message.

VoiceXML *subdialogs* allow you to incorporate “active” elements into forms. A subdialog is another VoiceXML document. You place the `<subdialog>` tag in a form; it performs its function and returns some results to the input variable.

Subdialogs are useful for dividing a large document or dialog into several smaller ones, which can make it easier to maintain and faster to load and execute. But the main use for subdialogs is to allow documents to call each other and exchange data, without using CGI or other server-side mechanisms.

This chapter describes:

- Transfer of Control
- Subdialog Invocation
- Value Return from a Subdialog

Transfer of Control

Subdialogs provide a way for one document to transfer control to another, similar to `<goto>`. But with `<goto>`, before the second document starts up, the first one is shut down; all its variables, grammars, and so on are lost. When you transfer control with `<subdialog>`, the first document is not shut down; it is paused.

When a document calls a subdialog, all its variables and other information about the caller’s state (sometimes called the *execution context*) are preserved while the subdialog is executing. The subdialog can even transfer control to additional documents by using `<submit>`, `<goto>`, and so on. The subdialog has its own execution context; the calling document remains suspended.

Eventually, the subdialog executes a `<return>` tag. This shuts down the subdialog and returns control to the document that called it, which then resumes execution. `<return>` can also pass variables back to the calling document. The returned values are passed to the caller as properties of the input variable.

Subdialogs can call other subdialogs in a nested fashion; the interpreter creates as many execution contexts as needed to keep track of them. Note that if a subdialog at any level executes an `<exit>`, all subdialogs are terminated as well as the original calling document.

Subdialog Invocation

The following is an example of a form that uses a subdialog. To see the document in a separate window, click *here*. (This example is also listed in Chapter 11, “Example Code”.)

This example would be appropriate for a service that allowed users to execute financial transactions. It starts by asking for the user’s account number. Then it calls a subdialog to identify the user, by asking for a password or other verification. The subdialog returns a variable named `succeed` which is true if the user’s identity has been verified. If `succeed` is true, the subdialog also returns a variable called `username`, allowing the document to include the user’s name in greetings.

This form has been simplified for the tutorial. It starts with an opening prompt, after which the `accountnum` field asks the user for an account number. Next is the `<subdialog>` tag that calls another document to check the user's identity.

The `<subdialog>` element has the name `checkresults` and a `src` attribute that specifies the URI of the document to call. Inside the `<subdialog>` tag is a `<param>` tag, which is used to pass data to the subdialog. In this case, the calling dialog will pass to the subdialog a variable named `acct`, whose initial value is set to the `accountnum` field that was spoken by the user.

After the `<subdialog>` element is a `<filled>` element that uses the result values returned by the subdialog. These values are returned as properties of the `checkresults` input variable and can be referenced with JavaScript expressions.

The subdialog is expected to return two values—a Boolean variable named `succeed`, which will be true if the user was properly identified, and a string variable named `username`, which contains the user's name if the user was identified.

The document uses `<if>` to check the returned value, `checkresults.succeed`. If this value is true, the document greets the user with his or her name. If `checkresults.succeed` is false, the document issues a warning and terminates the session with `<exit>`.

The final `<block>` is set up for tutorial purposes. In a real-world application, this is the point where the dialog would proceed to ask the user to select transactions to perform. Here, it merely issues a final prompt and then uses `<clear>` to reset the form, so you can try it again.

Value Return from a Subdialog

Now that you've seen the calling document, let's take a look at the subdialog itself. To see the document in a separate window, click [here](#). (This example is also listed in Chapter 11, "Example Code".)

The subdialog's job is to take the account number passed by the calling document and conduct an additional dialog to verify the user's identity. In a real-world application, this might start by asking for a personal ID number (PIN). If the user didn't remember the PIN, the dialog could try alternates, such as asking for a Social Security number or transferring the user's call to a human operator.

Eventually, the subdialog would either identify the user or determine that the user is invalid. It would then return this information to the calling document. If the user is identified, the subdialog could return additional information, such as the user's name or a transaction code.

Because this example is for tutorial purposes, it takes some shortcuts. Instead of contacting a server to verify the user's identity, it just asks the caller whether he wants to be "valid" or "invalid"; that is, which option to test.

The document starts with a `<var>` tag to declare the variable `acct`. This will receive the value passed by the `<param name="acct">` tag in the calling document. This allows the calling document to pass the user's account number to the subdialog.

The form has an opening prompt, after which it asks for the user's PIN. After that, it would be appropriate to contact a server, passing it the account number and PIN, to verify the user's identity. However, we simply ask the user to confirm or deny his own identity. This Boolean value is stored in the `succeed` input variable.

The next `<var>` tag is another tutorial shortcut; it creates a variable to hold the user's name. In a real application, this value would be supplied by a server after the user's identity was confirmed.

Finally there is a `<filled>` element that contains a `<return>` tag. The `namelist` attribute specifies that the subdialog will return two variables to the calling document. Look back at the calling document, described above, to see how the returned values are used.

Grammars are a very powerful feature of VoiceXML. They allow you to create flexible dialogs that can respond to natural-language input by your users. VoiceXML allows you to write grammars using several formats; all examples in this tutorial use the Nuance GSL format. This chapter describes how grammars work and the basics of the Nuance GSL format you can use to create grammars.

Grammars are defined with the `<grammar>` tag. You can write a grammar as part of your document, as in:

```
<grammar>
  ;GSL2.0
  ... grammar definition text ...
</grammar>
```

Alternatively, you can place the grammar in a separate file and reference it with the `src` attribute, as in:

```
<grammar
  src="http://www.yourcompany.com/yourfile.grammar#yourRule"/>
```

This section gives a basic look at grammars. For more details, see the *Grammar Reference*.

This chapter describes:

- Basic Grammar Rules
- Grammar Slots
- Grammars for Mixed-Initiative Forms

Basic Grammar Rules

The following form is set up for testing the grammar. It includes a `<prompt>` that confirms the user's choice; then, instead of submitting the field data, it uses `<clear>` to reset itself, so you can try another phrase on it.

```
<form>
  <field name="selection">
    <prompt>
      What service would you like?
    </prompt>
    <grammar type="application/x-nuance-gsl">
      [ news weather sports ]
    </grammar>
  </field>
  <block>
    <prompt>
      You chose <value expr="selection"/>
    </prompt>
    <clear/>
  </block>
</form>
```

The `<grammar>` element looks like this:

```
<grammar type="application/x-nuance-gsl">
  [ news weather sports ]
</grammar>
```

The grammar's `type` attribute tells the interpreter that the grammar is written in the GSL grammar format. You are not required to include this attribute. However, several of the grammar formats look similar; the comment makes it easier for someone to look at your grammar and determine the format.

The content is a very simple definition, called a *rule*, that accepts one of three words as a valid input for the field. This rule illustrates several basic facts about writing GSL grammars. The square bracket characters, `[` and `]`, indicate a *choice*; the interpreter will accept any one of the three words as a valid input. Note that the three words are all written with lower-case letters only; do *not* use capital letters in words to be recognized.

Saying a single word seems rather mechanical and unfriendly; you may want to give the user some more freedom. It's common to add formalities to the grammar. You could write:

```
( [ news weather sports ] ?please )
```

This rule will recognize any of the three keywords, optionally followed by the word "please." To achieve this, we have added a few more features to the rule:

- The parentheses indicate a *sequence*. A grammar expression `(A B)` means that the interpreter should recognize `A` followed by `B` as a valid input. `A` and `B` may be single words or they may be more complex expressions, as they are in this example.
- The `?` character indicates an optional item. `?please` indicates that the grammar will recognize spoken input with or without the word "please" at that point.

Now let's extend the grammar with some more common words:

```
(
  ?[ (i'd like) (tell me) ]
  ?the
  [ news weather sports ]
  ?please
)
```

We have spread the grammar out over several lines, to make it easier to read; the extra white space does not affect the result. The rule:

```
?[ (i'd like) (tell me) ]
```

will accept either of the phrases "I'd like" or "tell me." (Remember, no capital letters.) The `?` before the `[` indicates that the whole choice expression is optional; the interpreter will accept either of the phrases or neither of them at the beginning of the spoken input.

The expression `?the` indicates that the word "the" is optional. The various options provided by these rules mean that this grammar will recognize 36 different phrases including:

- News.
- Tell me news.
- Tell me the news.
- I'd like weather.
- Sports, please.
- Tell me the weather, please.

This grammar gives the user quite a bit of flexibility. But we need to do one more thing to make the grammar complete. As it is written above, the grammar will place the entire recognized phrase in the input variable. If you say, "I'd like weather," the confirmation prompt will say, "You chose I'd like weather." We

want to filter out all the formalities, so that the grammar will set the input variables to just one of the three keywords: `news`, `weather`, or `sports`.

Grammar Slots

To control what the grammar returns to the document, we use *slots*. A grammar slot is a sort of local variable. You put expressions called *commands* in a grammar; commands assign values to the slots when certain words or phrases are recognized. Let's put some commands in the GSL grammar:

```
<![CDATA [
  (
    ?[ (i'd like) (tell me) ]
    ?the
    [
      news      { <selection news> }
      weather   { <selection weather> }
      sports    { <selection sports> }
    ]
    ?please
  )
]]>
```

The commands are enclosed by the curly braces, { and }. We have one command located after each keyword, so it will be executed if the word is recognized.

Each command consists of a name and a value enclosed by < and > characters. Because < and > have special meaning in VoiceXML, we have "escaped" the entire grammar definition with <![CDATA[and]]> to prevent the commands from being misinterpreted as tags.

Each command assigns one value to a slot named `selection`. The values happen to be strings that are equal to the keywords, `news`, `weather`, and `sports`. But if we wanted to, we could just use `n`, `w`, and `s` or any other values that might be useful for our application.

If the grammar is defined in a field, the slot name, `selection`, is typically the name of the field.

Slot names are frequently used in a grammar for an entire form or for external grammars that will be reused by multiple forms. As shown in the next section, in a *mixed-initiative* form, the slots can be used to fill in multiple fields from a single user input. In the reuse case, the field that calls the external grammar would have a `slot` attribute set to the slot name filled by the grammar; upon return, the input variable is still set with the value.

For example, assume you have an external grammar named `dates.gsl` that contains rules for recognizing dates in various formats. One part of this large grammar might be the following rule:

```
Month [
  january {<month jan>}
  february {<month feb>}
  march {<month mar>}
  ...
]
```

(Ignore the "Month" part for now.) Basically, the rule matches 12 months of the year and returns the month in a slot named `month`. You could access this grammar from a form as follows:

```
<field name="date" slot="month">
  <prompt>
    In what month do you want to do this?
```

```
</prompt>
<grammar src="foo.gsl#Month">
```

In this case, the interpreter uses the rule to match what the user says. Because the grammar returns the information in a slot called `month`, the field needs to either have its name be `month` or, as here, have a slot named `month`. Here, the grammar returns the information and the interpreter sets the field's input variable `date` to what the user says; it *does not* set a variable named `month` in this form.

Grammars for Mixed-Initiative Forms

Suppose we want to enhance our application to allow a user to ask for two or three services in a single request, by saying something like “I'd like news and weather.” In this case, instead of having a single slot called `selection`, we'll use three separate slots called `news`, `weather` and `sports`. We'll design the grammar so that each slot will be set to `true` if the user asks for that service.

We want this grammar to be very flexible. The user should be able to ask for one, two, or all three services, in any order. If we had to write these combinations explicitly, this could make the grammar three times as long, with a lot of redundant rules. Fortunately, there is a shortcut using extra “worker” rules. We can divide a complex grammar into several short rules.

In an inline grammar with multiple rules, the first one is the *top-level* rule—the main rule that defines what the grammar is looking for. Let's create a rule called `Request` that represents the entire user input:

```
Request
(
  ?[ (i'd like) (tell me) ]
  Service
  ?and ?Service
  ?and ?Service
  ?please
)
```

The rule starts with a name. Note that the first letter of the name is capitalized, to identify it as a name, rather than part of the spoken text. (And that's why capital letters are not allowed in text.)

The formalities at the beginning and end of the rule are the same as in the previous grammar. In the middle, instead of the three choices, we have another capitalized word, `Service`. This is the name for another rule that can recognize the choices.

The `Request` rule has three occurrences of `Service`, two of which are optional (preceded by `?`). This allows it to recognize when the user asks for one, two, or three services. The presence of the two `?and` expressions allows the user to use the word “and” in any reasonable place, such as:

- News and weather.
- Sports, news, and weather.
- News and sports and weather.

To handle the individual choices, we can write the `Service` rule like this:

```
Service
(
  ?the
  [
    news      { <news true> }
    weather   { <weather true> }
    sports    { <sports true> }
  ]
)
```

)

This rule contains the three service names with their slot commands. Note that there are now three separate slots, whose values can be set to `true`, as discussed above. Also, the `?the` expression allows the optional word “the” before the service name.

The grammar formed by these two rules can be used in a mixed-initiative form with three boolean fields named `news`, `sports`, and `weather`. A form that uses this grammar is described in “Mixed-Initiative Forms” on page 29.

This chapter describes a number of techniques and tips that will make your VoiceXML applications more efficient and user-friendly:

- File Retrieval and Caching
- User-Friendliness
- Maintainability and Debugging Aids

File Retrieval and Caching

Since VoiceXML documents may be stored on remote servers, rather than on the Café server, there is a chance that Internet traffic could cause delays in processing. There are several ways to prevent the user from hearing long silences in such cases.

Every tag that can cause fetching of a document (`<submit>`, `<goto>`, and so on) accepts the `fetchaudio` attribute. This attribute specifies the URI of an audio file to play while the document is being fetched. The file may contain music, some sort of announcement, advertising, and so forth. Playing of the file stops as soon as the document is ready to continue executing. There is also a `fetchaudio` property that you can modify with the `<property>` tag. This property provides a default audio file to use for tags that do not specify the `fetchaudio` attribute.

To minimize delays, the interpreter maintains a cache for VoiceXML documents, audio files, and other files used by applications. Normally, once the interpreter has fetched a file over the Internet, it keeps a copy in the cache. If the application requests the file again, the interpreter uses the cached copy.

Sometimes, even when a file is in the cache, you always want the interpreter to check whether there is a newer version of the file on the server from which it was originally fetched.

To control caching, you can specify the `maxage` attribute on tags such as `<goto>` and `<audio>` that cause fetching of a file. The value of the `maxage` attribute is the maximum age of cached files that the application will use. We saw an example of this in Chapter 2, “A Basic Dialog”, in the `<link>` that reloads a document:

```
<link maxage="0" next="http://yourcompany.com/someDoc.vxml">
  <grammar type="application/x-nuance-gsl">
    [
      (bevocal reload)
      (dtmf-star dtmf-star dtmf-star)
    ]
  </grammar>
</link>
```

This link is intended for use in debugging. There may be times when you are on the phone, testing your document, as you are editing it. After you change the file, when you say “BeVocal reload,” you want to be sure that the interpreter will fetch the new version of the document. That is why we write `maxage="0"` in the `<link>`.

There is also a `maxage` property that you can modify with the `<property>` tag. This property provides a default value to use for tags that do not specify the `maxage` attribute.

User-Friendliness

Here are some tips that will help make your applications intuitive and easy to use:

- Use recorded audio for all prompt messages. Include text for TTS as a backup in case the audio file is not available, for example:

```
<audio src="http://www.yourcompany.com/audio/welcome.wav">
  Welcome to the Company voice mail system.
</audio>
```

- Use `<help>`, `<noinput>`, and `<nomatch>` event handlers to make sure your users are guided through your dialogs. Put handlers on individual form fields, to make the messages as specific as possible.
- Use `<reprompt>` to repeat context-specific prompts.
- Use prompt counters on `<prompt>`, `<noinput>` and `<nomatch>` to make messages become more detailed if the user gets stuck on one field.
- Test all dialogs thoroughly—with a variety of real-world users, not trained engineers.
- Provide adequate customer support. Make sure users have an easy way to contact a real person when they have problems with the computers.

Maintainability and Debugging Aids

As applications grow and change, they can become large, complex, and difficult to debug and maintain. Here are some techniques that will help you to manage your application:

- Keep your documents' structure "clean." Don't use lots of `<goto>` tags that may result in a tangled maze of interlocking documents.
- If a document becomes large, divide it into several smaller ones. Try to divide it into logical sections that minimize the need to jump from one document to another. Use subdialogs to eliminate redundant code.
- Choose variable names that are long enough to be descriptive.
- Use indentation and blank lines to make your documents easy to read.
- Use plenty of comments to explain unusual or tricky parts of your documents.
- Remember to use the *BeVocal tools*, such as the VoiceXML Checker, Log Browser, and Tracing Tool to troubleshoot your documents.

This chapter contains the long examples used throughout the tutorial. This chapter contains code for these examples:

- Calculator
- Factorial Function
- Mixed-Initiative Form
- Recording Audio
- Subdialog—Main Dialog
- Subdialog—The Subdialog

Calculator

This is the calculator example used in Chapter 2, “A Basic Dialog”.

```
<?xml version="1.0" ?>

<!DOCTYPE vxml
  PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
  "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0"
  xmlns="http://www.w3.org/2001/vxml"
  xml:lang="en-US">

<!--
  <meta name="maintainer" content="you@yourcompany.com"/>
-->

<link next="http://yourcompany.com/this.vxml">
  <grammar type="application/x-nuance-gsl">
    [
      (bevocal reload)
      (dtmf-star dtmf-star dtmf-star)
    ]
  </grammar>
</link>

<noinput>
  I'm sorry. I didn't hear anything.
  <reprompt/>
</noinput>

<nomatch>
  I didn't get that.
```

```
<reprompt/>
</nomatch>

<property name="universals" value="all" />
<help>
  I'm sorry. There's no help available here.
</help>

<form>
  <block>
    This is the BeVocal calculator.
  </block>

  <field name="op">
    <prompt>
      Choose add, subtract, multiply, or divide.
    </prompt>
    <grammar type="application/x-nuance-gsl">
      [add subtract multiply divide]
    </grammar>
    <help>
      Please say what you want to do. <reprompt/>
    </help>
    <filled>
      <prompt>
        Okay, let's <value expr="op"/> two numbers.
      </prompt>
    </filled>
  </field>

  <field name="a" type="number">
    <prompt>
      Whats the first number?
    </prompt>
    <help>
      Please say a number.
      This number will be used as the first operand.
    </help>
    <filled>
      <prompt> <value expr="a"/> </prompt>
    </filled>
  </field>

  <var name="result"/>

  <field name="b" type="number">
    <prompt> And the second number? </prompt>
    <help>
      Please say a number.
      This number will be used as the second operand.
    </help>
    <filled>
      <prompt> <value expr="b"/> Okay. </prompt>

      <if cond="op=='add'">
```

```
<assign name="result" expr="Number(a) + Number(b)"/>
  <prompt>
    <value expr="a"/> plus <value expr="b"/>
    equals <value expr="result"/>
  </prompt>
<elseif cond="op=='subtract'"/>
  <assign name="result" expr="a - b"/>
  <prompt>
    <value expr="a"/> minus <value expr="b"/>
    equals <value expr="result"/>
  </prompt>
<elseif cond="op=='multiply'"/>
  <assign name="result" expr="a * b"/>
  <prompt>
    <value expr="a"/> times <value expr="b"/>
    equals <value expr="result"/>
  </prompt>
<else/>
  <assign name="result" expr="a / b"/>
  <prompt>
    <value expr="a"/> divided by <value expr="b"/>
    equals <value expr="result"/>
  </prompt>
</if>

  <clear/>
</filled>
</field>
</form>
</vxml>
```

Factorial Function

This is the factorial example used in Chapter 3, “Variables, Expressions, and JavaScript”.

```
<?xml version="1.0" ?>
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
    "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US" >
<!-- <meta name="maintainer" content="you@yourcompany.com"/> -->

    <script>
        <![CDATA[
            function factorial(n) {
                return (n <= 1) ? 1 : n * factorial(n-1);
            }
        ]]>
    </script>

    <form>
        <var name="lastresult"
            expr="'This is the first factorial you have computed'" />
        <field name="n" type="number">
            <prompt> Say a number. </prompt>
            <filled>
                <prompt>
                    The factorial of <value expr="n"/> is
                    <value expr="factorial(n)"/>
                    <value expr="lastresult" />
                </prompt>
                <assign name="lastresult"
                    expr="'The last factorial you computed was for ' + n" />
                <clear namelist="n"/>
            </filled>
        </field>
    </form>

</vxml>
```

Mixed-Initiative Form

This is the example of a mixed-initiative form used in Chapter 4, "Forms".

```
<?xml version="1.0" ?>
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
    "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US" >
<!-- <meta name="maintainer" content="you@yourcompany.com"/> -->

<form>
  <grammar type="application/x-nuance-gsl">
    <![CDATA[
      Request
      (
        ?[ (i'd like) (tell me) ]
        Service
        ?and ?Service
        ?and ?Service
        ?please
      )
      Service
      (
        ?the
        [
          news      { <news true> }
          weather   { <weather true> }
          sports    { <sports true> }
        ]
      )
    ]]>
  </grammar>

  <initial>
    <prompt>
      What service would you like?
    </prompt>
  </initial>

  <field name="news"      type="boolean"> </field>
  <field name="weather"  type="boolean"> </field>
  <field name="sports"   type="boolean"> </field>

  <filled mode="any">
    <if cond="news"><prompt>You chose the news service.</prompt> </if>
    <if cond="weather"><prompt>You chose the weather service.</prompt> </if>
    <if cond="sports"><prompt>You chose the sports service.</prompt> </if>
    <clear/>
  </filled>

</form>
</vxml>
```

Recording Audio

This is the example of recording a user's responses used in Chapter 7, "Other VoiceXML Elements".

```
<?xml version="1.0" ?>
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
  "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">
<!-- <meta name="maintainer" content="you@yourcompany.com"/> -->

<form>

  <record name="message" maxtime="10s" finalsilence="2000ms">
    <prompt> Please say your message. </prompt>
    <noinput> I didn't hear anything, please try again. </noinput>
  </record>

  <field name="confirm" type="boolean">
    <prompt> Your message is <audio expr="message"/>
      Do you want to keep it?
    </prompt>
    <filled>
      <if cond="confirm==0">
        <clear/>
      <else/>
        <prompt> Thank you </prompt>
        <submit next="http://yourcompany.com/cgi-bin/respond.cgi"
          method="post"/>
      </if>
    </filled>
  </field>

</form>
</vxml>
```

Subdialog—Main Dialog

This is the main dialog used in the subdialog example of Chapter 3, “Variables, Expressions, and JavaScript”.

```
<?xml version="1.0" ?>
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
  "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">
<!-- <meta name="maintainer" content="you@yourcompany.com"/> -->

  <form>
    <block>
      <prompt> Welcome to the financial service. </prompt>
    </block>

    <field name="accountnum" type="digits">
      <prompt>
        What's your account number?
      </prompt>
    </field>

    <subdialog name="checkresults" src="subdialog.vxml">
      <param name="acct" expr="accountnum"/>
    </subdialog>

    <filled mode="all" namelist="checkresults">
      <if cond="checkresults.succeed">
        <prompt>
          Good day, <value expr="checkresults.username"/>
        </prompt>
      <else/>
        <prompt>
          I'm sorry, you have not been identified.
        </prompt>
        <exit/>
      </if>
    </filled>

    <block>
      <prompt> Thank you for calling! </prompt>
      <clear/>
    </block>
  </form>

</vxml>
```

Subdialog—The Subdialog

This is the subdialog dialog used in the subdialog example of Chapter 3, “Variables, Expressions, and JavaScript”.

```
<?xml version="1.0"?>
<!DOCTYPE vxml PUBLIC "-//BeVocal Inc//VoiceXML 2.0//EN"
  "http://cafe.bevocal.com/libraries/dtd/vxml2-0-bevocal.dtd">

<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">
<!-- <meta name="maintainer" content="you@yourcompany.com"/> -->

  <form>
    <var name="acct"/>
    <block>
      <prompt>
        This is the I D check for account <value expr="acct"/>
      </prompt>
    </block>

    <field name="id" type="digits">
      <prompt>
        What's your personal I D number?
      </prompt>
    </field>

    <!-- the next field and var are for demonstration purposes -->
    <field name="succeed" type="boolean">
      <prompt>
        Really?
      </prompt>
    </field>

    <var name="username" expr="'Mr. Jones'"/>
    <filled>
      <return namelist="succeed username"/>
    </filled>
  </form>

</vxml>
```