BeVocal VoiceXML Tutorial

Version 1.6
June 2001

BeVocal, Inc.
1380 Bordeaux Drive
Sunnyvale, CA 94089

©2001. BeVocal, Inc. All rights reserved.

Table of Contents

Preface 7

Prerequisites 7

Conventions 7

How to Use This Tutorial 8

Other Sources of Information 9

If You Need Help 9

1. Introduction 11

Voice Access to the Web 12

Getting Started 12

Developing an Application 13

Basics of VoiceXML 13

Conclusion 15

2. A Basic Dialog 17

Initialization and Debugging Features 18

Forms and Fields 19

Working with Field Data 21

Conclusion 23

3. Variables, Expressions, and JavaScript 25

JavaScript Expressions in Attributes 25

Script Elements 26

Defining and Using a Function 26

Manipulating Text with Functions 27

More on Variables 28

Session Variables 28

Variable Scopes 28

Properties 29

Applications and Root Documents 30

4. More on Forms 31

Fields 31

Field Types 32

Controlling Forms 32

Advanced Uses of Prompts 33

Recorded Audio in Prompts 33

Reprompting and Counting Prompts 34

Timeouts 35

Barge-in 35

Mixed-Initiative Forms 35

5. Recording and Playing Audio 37

Playing Stored Audio Data 37

Barge-in 38

Recording User Input 38

6. Interaction with Servers 41

Submitting Data 41

Specifying Variables to Submit 41

Passing Data in URLs 42

7. Other VXML Elements 43

Links 43

Menus 44

Enumerating Choices 44

Using DTMF in Menus 45

Selecting the Next Document, Form, or Field 45

Exiting 46

8. Objects and Subdialogs 47

Objects 47

Subdialogs 48

Calling a Subdialog 48

Returning Values from a Subdialog 49

9. More on Grammars 51

Basic Grammar Rules 51

Grammar Slots 53

Grammars for Mixed-Initiative Forms 53

10. Design and Performance Considerations 55

File Retrieval and Caching 55

User-Friendliness 56

Maintainability and Debugging Aids 56

Preface

Welcome to the world of BeVocal application development!

BeVocal 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. Your documents can also include Nuance SpeechObjects: pre-written and tested software packages that conduct dialogs for a variety of common functions.

Prerequisites

Besides interacting with the user, a complete BeVocal application usually accesses Web pages or other Internet server functions. So 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

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

Italic font is used for:

- Introducing terms that will be used throughout the document

- Emphasis

How to Use This Tutorial

This on-line tutorial is structured somewhat differently from a normal, sequential book. The first two chapters introduce VoiceXML:

- Chapter 1, "Introduction" gives an overview of the BeVocal system, and 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 remainder of the tutorial consists of a "cookbook" of VoiceXML techniques:

- Chapter 3, "Variables, Expressions, and JavaScript" gives details on types of data manipulation that are available in VoiceXML.
- Chapter 4, "More on 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, "Recording and Playing Audio"describes how to use stored audio in your dialogs, and how to record audio messages from the telephone user.
- Chapter 6, "Interaction with Servers" describes ways to pass data from VoiceXML documents to Web servers.
- Chapter 7, "Other VXML Elements" discusses VoiceXML elements, such as links and menus, that provide additional features for your documents.
- Chapter 8, "Objects and Subdialogs" describes how to use Nuance SpeechObjects in your documents, and 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.

We recommend that you start by reading Chapters 1 and 2 to familiarize yourself with VoiceXML. After that, you may read any of the "cookbook" chapters in any order that seems most useful for your intended application. Once you have finished with this tutorial, you can find more advanced information in the World Wide Web Consortium's on-line manual, and other sources listed below.

Other Sources of Information

Besides this tutorial, BeVocal provides the following resources:

- Getting Started provides a variety of introductory information.
- VoiceXML Reference provides basic information about all VoiceXML tags.
- 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 on the BeVocal system.
- SpeechObjects Quick Reference provides a summary of the Nuance SpeechObjects that are available for use in your documents.
- Geo API Quick Reference provides descriptions of some JavaScript functions that you can use to obtain and manipulate geographical information.
- The Audio Library contains stored audio files with commonly used spoken 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 reference manual at:

`http://www.w3.org/TR/2000/NOTE-voicexml-20000505/`

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

`http://www.xml.org/xmlorg_resources/index.html`
`http://www.w3.org/XML/`

For information on Nuance SpeechObjects, see Nuance's Web site at:

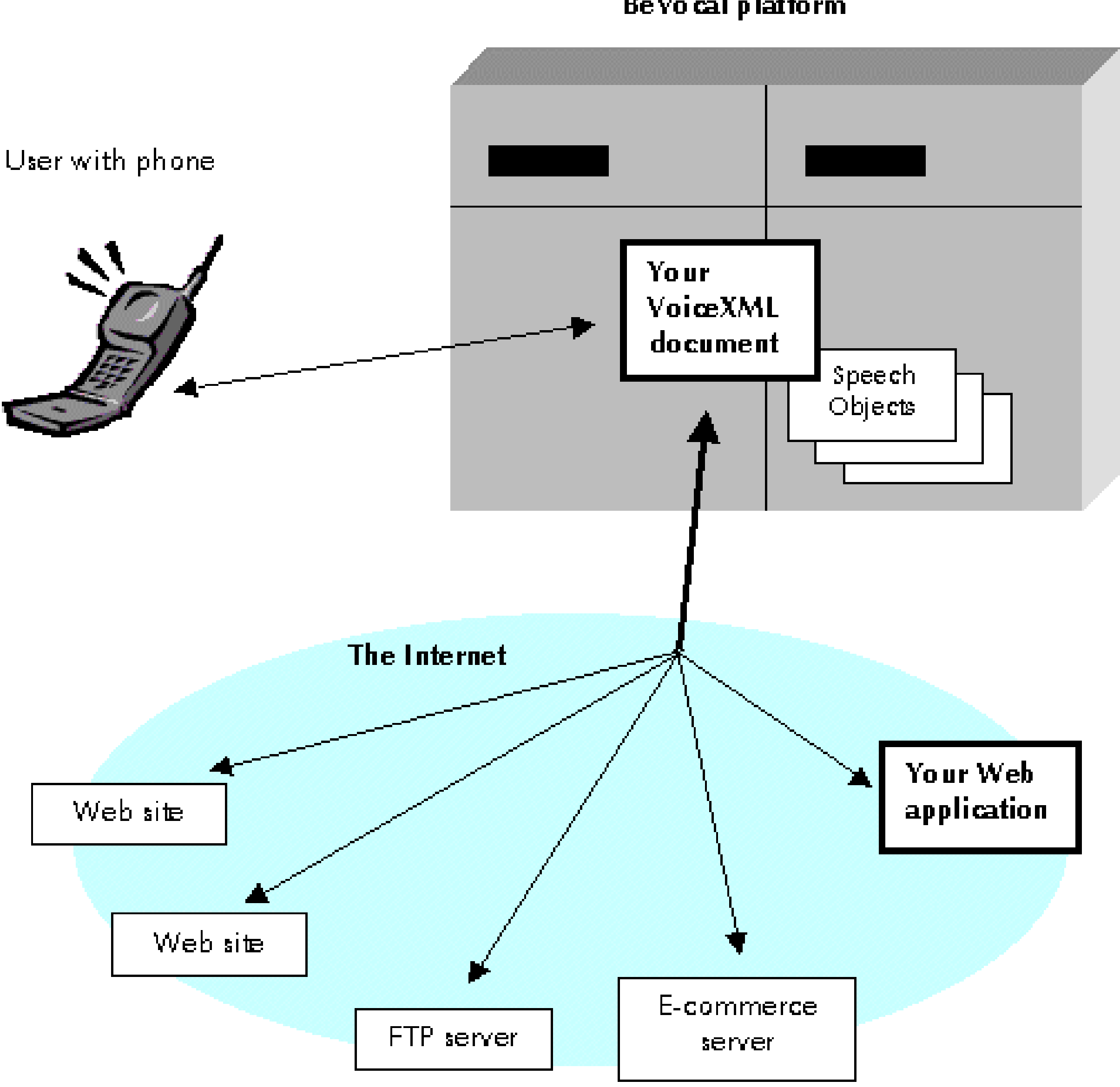
`http://www.nuance.com/products/speechobjects.html`

If You Need Help

Support for this tutorial is available by email at the BeVocal Cafe Newsgroups.

1 Introduction

The following diagram shows a typical BeVocal 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 (branching) and other complex structures.

Writing powerful documents is easier when you use Nuance SpeechObjects. These are pieces of software that are pre-written, tested, and packaged in a form that is easy for a VoiceXML document to use. SpeechObjects conduct dialogs for common functions such as accepting credit card numbers, times and dates, and dollar amounts.

Voice Access to the Web

A complete BeVocal application usually requires some resources outside the BeVocal server. 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 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 BeVocal as a "front end" to a Web application, you minimize amount of VXML coding that is required. The rest of your application can be based familiar protocols such as HTTP, CGI, etc., for which powerful programming tools are available.

Getting Started

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

- A username and password for your BeVocal account.
- The phone number that users will dial to access your application
- The location where your VoiceXML document file(s) will be stored.

To get started, you can set up an account yourself over the Web, by using the BeVocal Developer's Cafe Web site at:

`http://cafe.bevocal.com`

You may store your document and related files on BeVocal's server. In that case, you can use the File Management Web pages 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 URL of the main document by entering it in a form on the File Management page.

Developing an Application

Since VoiceXML documents consist of plain text, you can create and edit them with any simple text editor such as Notepad on Windows, or emacs of “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 provides several tools to help you develop your applications:

- The File Management page lets you upload, view, and delete files that you have stored on BeVocal’s 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, missing quote characters, etc.
- 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, etc.
- The Trace Tool lets you monitor a call in real-time, while a user is on the phone.
- The Port Estimator can help you figure out how many telephone lines (ports) you will need to support a specified volume of calls.

These tools are all available at the Developer’s Cafe’ Web site:

http://cafe.bevocal.com

Basics of VoiceXML

At this point, let’s look at a simple VoiceXML document.

Line	VoiceXML statement
1	`<?xml version="1.0"?>`
2	`<vxml version="1.0">`
3	`  <form>`
4	`    <field name="selection">`
5	`      <prompt>`
6	`        Please choose News, Weather, or Sports.`
7	`      </prompt>`
8	`      <grammar>`
9	`        [ news weather sports ]`

Line	VoiceXML statement
10	</grammar>
11	</field>
12	<block>
13	<submit next="select.jsp"/>
14	</block>
15	</form>
16	</vxml>

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

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 line 6.

A tag may have *attributes* inside the angle brackets. Each attribute consists of a *name* and a *value*, separated by an equal (=) sign; and the value must be enclosed in quotes. Examples are the version attribute in line 2, and the name attribute in line 4.

Most of the tags in this example are used in pairs. The <vxml> tag is matched by </vxml>, and tags such as <form> and <block> have similar counterparts. Tags that are used in pairs are called *containers*.

Some tags are used singly, rather than in pairs; these are called *stand-alone* or *empty* tags. The only one in this example is the <submit ... /> tag. Note 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 in all cases. 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.

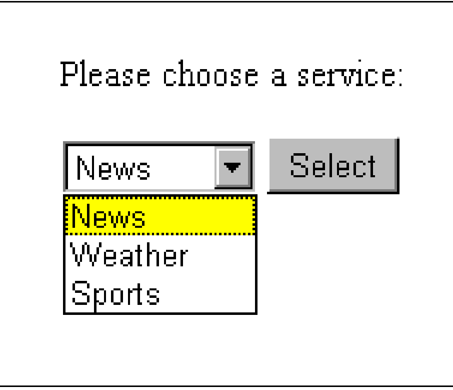
The term *element* is sometimes used in talking about languages such as HTML and VoiceXML. An element is either:

- 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.

Lines 1 and 2 are required at the beginning of all VoiceXML documents, and should be written just as shown above. The last line (16) is required at the end of all documents; it matches the <vxml> tag in line 2.

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, we 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 (line 4) that stores the user's choice.

Inside the field is a <prompt> tag (line 5), which contains a line of plain text. The VoiceXML interpreter will use its text-to-speech (TTS) translation system to read this text and send the resulting audio to the user.

After the <prompt> is a <grammar> tag (line 8) that defines the three responses that the platform 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. For more on grammars, see Chapter 9, "More on Grammars", or the on-line grammar reference.

Conclusion

In this Introduction, we have given you an overview of how BeVocal applications work, and how to write VoiceXML documents. In the next chapter, we will take a closer look at interacting with the user: how to make a voice-response application that is powerful and easy to use.

2 A Basic Dialog

In the first chapter, we looked at the fundamentals of VoiceXML as they applied to a very simple script with only one field. Next, 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; and 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:

Simple calculator

What do you want to do? Add

What's the first number? 345

What's the second number? 67

The answer is 412

Reset

Help

A VoiceXML document 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 shown above.

This document is longer than our first example, so we'll examine it one part at a time. To see the entire document in a separate window, click here.

Initialization and Debugging Features

Like all VoiceXML documents, our calculator starts with these two lines:

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

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

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

This meta-tag has the name `maintainer`, and its content is an email address. The BeVocal server will send a message to this address each time the document is executed. This allows you to keep a log of your application's activity. Of course, if your document is being used by hundreds of callers every day, you will get a lot of email…which brings up the subject of 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 VoiceXML interpreter. Comments can be used to put descriptive or historical information into a script, such as:

```
<!-- BeVocal 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:

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

Now you can leave the tag in place; and 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.

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
    caching="safe"
    next="http://myCompany.com/someDoc.vxml">
  <grammar>
    [
      (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 active at any time while the server is executing the element that contains the link. In this case, the link is a child of the <vxml> element, so it is active throughout the whole document. The `next` attribute specifies the document to run when the link is activated. We have specified the URL of this document itself; so when the link is activated, it has the effect of restarting the document from scratch, as if you had just dialed in.

The <grammar> 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.

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:

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

<noinput> is used to specify 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 server 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 possible grammar.

```
<help>
  I'm sorry. There's no help available here.
</help>
```

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

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

Note: Most examples in this tutorial will use vocal prompts that are written as plain text, to be played by the server's text-to-speech (TTS) processor. 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, it is recommended that you record all the necessary prompts, and convert your script to

play them by using the <audio> tag. (See Chapter 5, “Recording and Playing Audio” for details on this.)

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>
    [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 has the name op, which can be used in VoiceXML expressions to reference the field value, i.e., what the user selects. The field contains a <prompt> and a <grammar> tag that allow 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; afterwards, the <reprompt> tag will repeat 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 that action to take once the field is “filled in,” i.e., 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 field 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 fields 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 BeVocal system has a built-in grammar for numbers. The field contains <prompt>, <help>, and <filled> tags that function like the ones described above for the op field.

Working with Field Data

Next, we need a field to accept the second number for the calculation. The <filled> tag in this field can be the one that does the calculation and reports the answer to the user. In order to do this, 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. We have declared fields for the numbers to use for the calculation; but since the script will compute the result, there is no field for it. This variable will be used to hold the result.

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. So 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 server 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 server 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. The expression could be written as `a + b`, but the use of the `Number()` functions is necessary because the `+` symbol can indicate concatenation as well as addition. 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, so only division is left. After the prompt is the `</if>` tag that closes the entire `<if>` ... `<elseif>` ... `<elseif>` ... `<else/>` sequence.

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. This causes execution to resume at the top of the form, so the user can make another calculation.

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

These tags close the `<filled>` element, the field that contains it, the entire form, and finally, the VoiceXML document.

Conclusion

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>` in 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/>`.

3 Variables, Expressions, and JavaScript

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 provides the JavaScript language (also called ECMAScript) for manipulating data. JavaScript is very powerful: it provides many types of 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. The ECMA has a standard document that you can download.
- For a list of the specific features supported by the BeVocal implementation of JavaScript, see the on-line 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 one first.

JavaScript Expressions in Attributes

VoiceXML has very few programming features of its own; so JavaScript is, in a sense, the "native" programming language. For that reason, there are many places in VoiceXML attribute values where you can use JavaScript expressions without having to enclose them with `<script>` and `</script>`.

Specifically, you can use JavaScript expressions as values of attributes, for example, the `expr` attribute of tags such as `<assign>`, `<field>`, and `<param>`, or the `cond` attribute of tags including `<if>` and `<field`.

So there are a lot of useful places where you can use JavaScript expressions. 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 built-in 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. However, since VoiceXML is based on XML, it is helpful to also enclose the code with `<![CDATA[` and `]]>` inside the `<script>`; that is:

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

Without the `<![CDATA[` and `]]>`, you would need to replace some special characters with their entity equivalents, e.g. `<` and `>` instead of the `<` and `>` characters.

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, i.e., contained by the `<vxml>` element. In that case, it is executed only once, when the document is first loaded into the BeVocal server. 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.

Defining and Using a Function

Let's look at another type of calculator: this document will use JavaScript to define a function that computes factorials. A VoiceXML form is used to accept a number from the user; then the JavaScript function is used to compute the answer.

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

As in Chapter 2, we'll present the document in sections. To see the entire document in a separate window, click here.

First comes the initialization code:

```
<?xml version="1.0" ?>
<vxml version="1.0">
  <script>
    <![CDATA[
      function factorial(n) {
        return (n <= 1) ? 1 : n * factorial(n-1);
      }
    ]]>
  </script>
```

After the usual `<?xml>` and `<vxml>` tags is the `<script>` element that defines the function. As mentioned before, the JavaScript code is enclosed by the delimiters:

```
<![CDATA[
```

and:

```
]]>
```

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

The next section of code defines the form. The first part of this is straightforward. The field is declared as a `number`, and the `<prompt>` requests input from the user:

```
  <form id="foo">
    <field name="n" type="number">
      <prompt>
        Say a number.
      </prompt>
```

Next comes a `<filled>` item that reports the result to the user. It uses the `expr` attribute of a `<value>` tag to call the `factorial()` function:

```
      <filled>
        <prompt> The factorial of <value expr="n"/> is
           <value expr="factorial(n)"/>
        </prompt>
        <clear namelist="n"/>
      </filled>
    </field>
  </form>
</vxml>
```

Manipulating Text with 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.myCompany.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.myCompany.com/appName/audio/"
    + filename + ".wav")
}
```

In JavaScript, the + operator is used to combine or *concatentate* 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 55).

More on Variables

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

- You can declare them explicitly with the `<var>` tag.
- 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 field 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.

If you 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, e.g.:

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

The following sections describe some additional features of VoiceXML variables.

Session Variables

VoiceXML has some built-in variables called *session variables*. The VoiceXML interpreter creates these variables as soon as a user dials into your service, and their values remain constant throughout the call, even if several documents are executed. For instance, the variable `session.telephone.ani` contains the phone number from which the user dialed into your service (if the telephone service is set up to provide it). A complete list of session variables is available in the VoiceXML Programmer’s Guide.

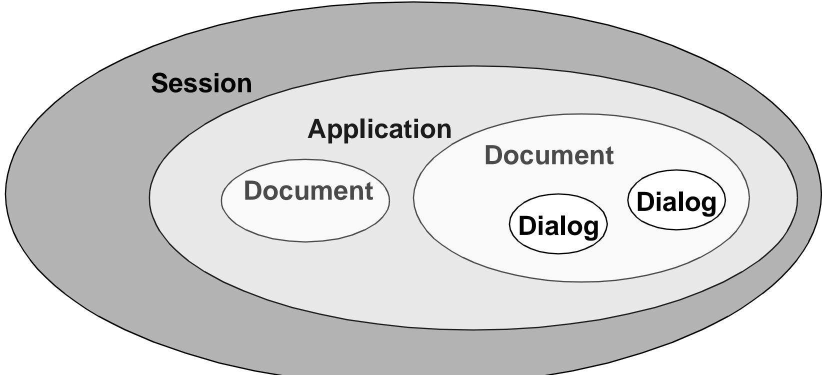
Variable Scopes

A variable’s *scope* is the range or “territory” where 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 an error in your document, if you try and use a variable in a location outside its scope. So it’s important to understand how they work.

- If you declare a variable inside a `<block>`, `<filled>`, or event handler (e.g. `<noinput>`) element, it is *local* to that element: it cannot be used in expressions outside that element. This is sometimes called *anonymous scope*, as will be explained shortly.
- 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 below) 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 (see above) are created as soon as a user dials into your service. They are preserved throughout the call, even if the application (root document) changes. This is called *session scope*.

Notice how the various scopes form a nested structure:



Under these rules, it's possible for you to 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, i.e., 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. 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: since it is the most local of all scopes, there is no need for a way to explicitly refer to it.

Properties

VoiceXML properties are built-in variables that control various features of the system. 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 system 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. But it's often helpful to have some variables that are preserved from one document to the next.

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

- 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 URL in 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 telephone 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.

4 More on Forms

In the calculator example in Chapter 2, “A Basic Dialog”, you saw a basic VoiceXML form that collected three field values from the user. The form is the basic mechanism for controlling interaction with the user. So in this chapter, we are going to 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.

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 system 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. The system can fill in several fields from a single user input. See “Mixed-Initiative Forms” on page 35.

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: *field items* and *control items*. Field items include `<field>` itself, 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 field items are listed below:

- `<field>` is used to define fields for user input. This is an important function, and fields may contain other elements in a nested structure.
- `<object>` references a predefined software package; this is how you access Nuance SpeechObjects in your documents.
- `<record>` makes an audio recording of the user’s input.
- `<transfer>` is for transferring the telephone caller to another destination.
- `<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 are listed below:

- `<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 35).

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*, i.e., when the user has spoken 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 system 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 may say "Yes" or press 1 on the DTMF keypad to generate a true result; saying "No" or pressing 2 yields 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: e.g., "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 four digit year, 2-digit month, and 2-digit day, in that order. The field's value is a string of 8 digits, with question mark (`?`) characters indicating digits for which the user did not provide a value.
- `time` defines a field that accepts 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 *`CCCdddd.dd`*, where *`CCC`* is a three-character currency indicator and *`dddd.dd`* is a monetary value. For DTMF input, the star (`*`) key can be used to indicate a decimal point.
- `phone` defines a field that accepts a telephone number. The field's value is a string of digits, and 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 some type of grammar to determine what user input is acceptable. The grammar also specifies a string value to be placed in the field variable.

Controlling Forms

Normally, the BeVocal system processes a form by starting with the first item, and proceeding forward. However, there are several ways to control the interpretation order, that can give 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 system only executes form items whose value is undefined. So 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, the value can be removed by `<clear>`.

`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 system only executes form items whose guard condition has a true value. So 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 value(s) 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 where the user is.

```
<form>
  <field name="selection">
    <prompt> Please say News, Weather, or Sports. </prompt>
    <grammar>
      [ 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). So, when the interpreter gets to the `zipcode` field, it will only process it if the user has requested the weather service.

Advanced Uses of Prompts

The following sections describe some additional features of prompts.

Recorded Audio in Prompts

A finished application should use recorded audio for all prompts and other messages that are played to the user. The BeVocal system can translate plain text to speech, and this is very handy for developing and debugging an application; but the higher quality of recorded audio is preferable for real-world users.

You can easily write prompts with a format such as:

```
<prompt>
  <audio src="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 plain text as a back-up message. To execute this prompt, the BeVocal system will first try to fetch the specified audio file. If the file is available, the system plays it, and ignores the plain text. If the file does not exist or is not available for some other reason, the system will read the plain text to the user.

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

Note: Most examples in this tutorial do not use audio prompts. This is done in order to simplify the examples and make their structure more clear.

Reprompting and Counting Prompts

The `<reprompt>` tag causes the system to repeat the prompt for the current form item or menu. This is intended mainly for use in error handlers such as `<noinput>`, as shown in Chapter 2, “A Basic Dialog”.

For every prompt in your document, the system maintains a count of how many times it has been played to the user. You can use the prompt count to make your dialogs more helpful, by writing `<prompt>` tags with a `count` attribute.

Prompt counts can be used to write a field that gives more detailed instructions if a user seems to be having trouble with it. For instance, a shopping application might use a prompt such as:

```
<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="5">
    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. The system will use the prompt without the `count` (the shortest one) the first two times the user enters the field. Once the prompt count reaches 3, the system will play the next longer prompt. When the count reaches 5, the system will play the longest prompt.

Timeouts

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

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

For more information on properties, see "Properties" on page 29.

Barge-in

Normally, while the BeVocal system is playing a prompt, the user can "*barge in*" and speak a response at any time. When the system detects incoming speech, it interrupts the prompt. However, there are times when you want disable barging in, to ensure that the user hears a complete prompt. This could be useful for important messages such as legal notices or disclaimers, and also for advertisements.

To create a prompt that cannot be interrupted by a user barging in, write the `<prompt>` tag with the `bargein` attribute set to `false`, as in:

```
<prompt bargein="false">
  <audio src="advertisement.wav">
    This service was brought to you by the
    Instant Data Company.
  </audio>
</prompt>
```

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

In Chapter 1, 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>
        [ 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, so that the user can ask for one, two, or all three services in a single request.

To see the document in a separate window, click here.

Asides from 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 grammar is described in detail in "Grammars for Mixed-Initiative Forms" on page 53. 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 come the `<inital>` element, which is always executed first when the system processes a mixed-initiative form. It contains an opening prompt, and could also do other actions such as initializing variables.

Next are three `boolean` fields named `news`, `weather`, and `sports`. Since these names match the slot names defined in the grammar, the system automatically maps the slots to the fields, so that the commands in the grammar can store values into the fields.

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

5 Recording and Playing Audio

The ability to play and record audio data makes it possible to write professional, easy-to-use applications. This chaper describes techniques for manipulating audio files in VoiceXML.

BeVocal 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 at the Developers Cafe' Web site, at:

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

Playing Stored Audio Data

You can use the `<audio>` tag to play an audio file to the telephone user. The file may contain speech, music, "beep" tones, or any other type of audio information.

You can use the `src` attribute to specify the URL of the audio file, as in:

```
<audio
  src="http://www.myCompany.com/audio/english/welcome.wav"/>
```

Sometimes, it is useful to use JavaScript expressions to construct URLs. In that case, you can use `expr` instead of `src`, as in:

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

In this case, the URL is constructed by an expression that combines two strings with a variable named `userlanguage`. This could be used to make your application multilingual. As shown here, the URL 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` is presumably set up early in the dialog, with the name of the appropriate folder.

For another example of using JavaScript in `<audio>` tags, see "Manipulating Text with Functions" on page 27.

You can set up `<audio>` tags with a back-up text message. If the beVocal system is unable to retrieve the audio file, it will read the text to the user, for example:

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

In this example, the system will try to retrieve and play the `welcome.wav` file. If the file is unavailable, the system will read the text message to the user.

Note: The `<audio>` tag may be written as either a container or a stand-alone tag. When writing it stand-alone, remember to include the slash character (`/`) before the trailing angel bracket (`>`), as shown in examples at the beginning of this section.

Barge-in

Normally, while the BeVocal system is playing a prompt to a user, it is also listening for the user's response. That way, an experienced user can save a lot of time by *barging in* on the prompts (interrupting them). Sometimes, though, you may want to disable barge-in and require the user to listen to an entire prompt before continuing. This can be helpful for error or warning messages, legal notices, and advertising.

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`; so barge-in is normally enabled for all prompts.

Recording User Input

It is sometimes useful to be able to record and store audio input from the user, for voice mail and similar applications. 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 system processes the audio and extracts the numeric or string value.

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

```
<record name="greeting" 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` is used to indicate 10 seconds in this example.
- `finalsilence` specifies the amount of silence that will signal the end of the recording. The suffix `ms` is used to indicate 2500 milliseconds (2.5 seconds).

Note that, unlike with `<field>`, the data collected by an `<audio>` element cannot be manipulated with JavaScript. You must submit it to a server that is set up to receive and process it.

`<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.

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, so the user can record a new message.

6 Interaction with Servers

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 URL.

One document can pass data to another with `<subdialog>` and `<return>`, as described in "Subdialogs" on page 48. 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.

Submitting Data

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

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

This tag would be used inside a form, and would cause the system to submit all the form fields to the specified server. The `next` attribute specifies the URL of the server. If you want to use a JavaScript expression to generate the URL, 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 BeVocal system 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.

Specifying Variables to Submit

`<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://mycompany.com/cgi-bin/respond.cgi"
    method="post"
    namelist="name acctnumber password"/>
```

namelist provides several advanced features:

- It allows you to submit only part of a form, instead of submitting all the fields.
- It allows you to submit fields from several forms at once.
- It allows you to submit other variables, which are not part of any form.

Passing Data in URLs

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

```
http://mycompany.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 "name=value" expressions separated by ampersand (&) characters. You can use JavaScript to create URLs like this for <submit>, <link>, and <goto> tags. For example:

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

Here, the expr attribute contains a rather long JavaScript expression that concatenates the base URL 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.

7 Other VXML Elements

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

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; and 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, we presented this link, which can be used to reload and restart a document for debugging:

```
<link
    caching="safe"
    next="http://yourcompany.com/this.vxml">
  <grammar>
    [
      (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 URL should specify the document itself, in order to reload it. You can use `expr` instead of `next` if you want to use a JavaScript expression to create the URL.

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 system is executing the VoiceXML code for that form. If a link is placed at the top level, for example, just after the `<vxml>` tag, it will be available throughout the entire document.

The `caching="safe"` attribute ensures that the system will always get the latest version of a document, rather than re-use a saved copy; this is probably necessary for debugging a new document. (For more on caching, see "File Retrieval and Caching" on page 55.)

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.

Enumerating Choices

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

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

When the BeVocal system executes this prompt, 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.

Using 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.
    Press N for news, W for weather, or S for sports.
  </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.

Selecting the Next Document, Form, or Field

As its name implies, the `<goto>` tag is used to make the system "go" or "jump" from one place to another while executing your document. The destination of the jump may be another document; in that case you specify the URL with the `next` attribute, as in:

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

Also, the destination may be another dialog in the same document. In that case, you 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. It is recommended that you keep your document structures "clean" and straightforward when possible. (See "Maintainability and Debugging Aids" on page 56 for details.)

Exiting

Normally, the BeVocal system processes a form sequentially, from start to end. Sometimes, though, it is useful 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 immediatly 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>.)

8 Objects and Subdialogs

VoiceXML *objects* and *subdialogs* are powerful features that allow you to incorporate "active" elements into forms. These elements conduct entire dialogs and return some result values to the form.

Objects are pre-written programs in Java or other languages, that have been "packaged" in a way that makes them easy to use in VoiceXML. Subdialogs are VoiceXML documents in which, as the name implies, one dialog calls another, passes it some parameter data, and receives results back from it.

Objects

Objects in VoiceXML are similar to applets in HTML. They are software modules that perform some useful function, and return some data to the document that uses them. You use an object by writing an `<object>` tag in a form.

BeVocal provides a library of Nuance SpeechObjects. These objects handle a variety of common dialogs such as Yes/No, times and dates, currency amounts, and so on.

Here is a simple example that demonstrates the use of the `<object>` tag to call a SpeechObject for a date. The user says a date, and the document responds by saying the day of the week. To see the document in a separate window, click here.

This example starts with a short JavaScript definition for an array of twelve month names, and another for the seven weekday names. The SpeechObject will return the month and weekday as numbers, and these arrays will allow the document to say the words.

Next is a form that has an opening prompt, followed by the `<object>` element that calls the SpeechObject. `<object>` is a field item (see Chapter 4, "More on Forms"), so it has a `name` attribute for its field variable. It also has the `classid` attribute that specifies the URL of the object. Since SpeechObjects are built into the BeVocal system, you reference them with URLs that start with `speechobject://`.

When the `<object>` element is executed, the system runs the program in the object. Any data that the object returns will be placed in the field variable, `date`. The returned variable may be a JavaScript object with multiple properties.

This SpeechObject asks the user for a date. It analyzes the user's input, gives additional prompts if necessary, and returns a variable with three properties containing numeric values representing the month, day of the month, and day of the week.

Inside the `<object>` is a `<filled>` element that is executed when the SpeechObject finishes and control returns to the document. There is a prompt that reports the 3 values returned by the SpeechObject. This is followed by a `<clear/>` tag that resets the form, so that the user can enter another date.

Inside the prompt are several `<value>` tags that insert the values returned by the SpeechObject into the spoken output. The first `<value>` uses an array expression to choose the proper name, based on the returned variable `date.Month`, that is, the `Month` property of the `date` field variable.

The second `<value>` inserts the numeric value of the `date.DayOfMonth` variable. The third `<value>` inserts the name for the weekday, based on the returned variable `date.DayOfWeek`; however, it needs to make a slight adjustment.

In JavaScript, the first item in an array is considered the "zeroth," not the first. The returned `Month` value starts with 0 for January, so this is compatible with the JavaScript array. But the returned `DayOfWeek` value starts with 1 for Sunday; it is never zero. So the `<value>` expression subtracts one from `DayOfWeek` in order to choose the correct name.

For more information on using SpeechObjects, see SpeechObjects Quick Reference.

Subdialogs

The `<subdialog>` tag is used much like `<object>`. You place it in a form, and it performs its function and returns some results to the field variable. The difference is that a subdialog is not a pre-packaged applet or software module: it's another VoiceXML document.

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.

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, and 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>`, etc. The subdialog has its own execution context, so 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 some variables back to the calling document. The returned values are passed to the caller as properties of the field variable, as they are for `<object>` (described above).

Subdialogs can call other subdialogs in a nested fashion; the system 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.

Calling a Subdialog

Here's an example of a form that uses a subdialog. This 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.

To see the document in a separate window, click here.

This form has been simplified for use in 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 URL of the document to call. Inside the tag is a `<param>` tag, which is used to pass data to the subdialog. In this case, the caller 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>` is a `<filled>` element that uses the result values returned by the subdialog. These values are returned as properties of the `checkresults` field 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 it is true, the document greets the user with his or her name. If `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 a real-world application, this is the point where the dialog would proceed to ask the user to select transactions to perform. For this demonstration, it merely issues a final prompt, and then uses `<clear>` to reset the form, so you can try it again.

Returning Values from a Subdialog

Now that you've seen the calling document, let's take a look at the subdialog itself. Its 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 assistant.

Eventually the subdialog would either identify the user, or determine that the user is invalid, and would 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.

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

To see the document in a separate window, click here.

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 starts with 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, since this is a tutorial, we take a shortcut,

and simply ask the user to confirm or deny his own identity. This Boolean value is stored in the `succeed` field variable.

The `<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 comes 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.

9 More on Grammars

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. This section describes how grammars work, and the notation that is used to create them.

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

```
<grammar>
  ...
  ... grammar definition text ...
  ...
</grammar>
```

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

```
<grammar
    src="http://www.myCompany.com/myfile.grammar#myRule"/>
```

This section gives a basic look at grammars. For more details, see the <u>on-line grammar reference</u>.

Basic Grammar Rules

Here is a form with a very simple grammar, similar to the one we looked at in Chapter 1:

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

This form is set up to be used 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.

The `<grammar>` tag contains this text:

```
[ news weather sports ]
```

which 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 grammars. The square bracket characters, `[` and `]`, indicate a *choice*: the system 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, so you may want to give the user some more freedom. It's common to add some 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 system 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 system will accept either phrase, or neither one, at the beginning of the spoken input.

The expression `?the` represents the word "the," and makes it 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.

So 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 field 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 field 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, which assign values to the slots when certain words or phrases are recognized. Let’s put some commands in the grammar:

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

The commands are enclosed by the 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 save a few bytes, we could just use `n`, `w`, and `s`; or `1`, `2`, and `3`; or any other values that might be useful for our application.

The slot name, `selection`, is also the name of the field in which the grammar is defined. This is not strictly necessary. When a grammar is assigned to a single field, *all* slot commands modify the field variable, regardless of the name specified.

Slot names are really intended for use in a grammar for an entire form, rather than for a single field. Then the slots can be used to fill in multiple fields from a single user input. A form that is set up this way is called a *mixed-initiative* form.

Grammars for Mixed-Initiative Forms

Suppose we want to enhance our application by allowing 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 any two services, or all three, in any order. This could make the grammar three times as long, with a lot of redundant rules; but fortunately, there is a shortcut called *subgrammars*. We can divide a complex grammar into several short rules.

In an in-line 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 is 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 the previous grammar. In the middle, instead of the three choices, we have another capitalized word, `Service`. This is the name for another rule (a subgrammar) 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 35.

10 Design and Performance Considerations

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

File Retrieval and Caching

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

Every tag that can cause fetching of a document (`<submit>`, `<goto>`, etc.) accepts the `fetchaudio` attribute. This attribute specifies the URL 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 system maintains a cache for VoiceXML documents, audio files, and other files used by applications. Normally, once the system has fetched a file over the Internet, it keeps a copy in the cache. If the application requests the file again, the system uses the cached copy; this is known as *fast caching*.

Sometimes, even when a file is in the cache, you may always want the system to check whether there is a newer version of the file on the server from which it was originally fetched. This is known as *safe caching*.

To control caching, you can specify the `caching` attribute on tags such as `<goto>` and `<audio>` that cause fetching of a file. The value of the `caching` attribute may be either `fast` or `safe`. We saw an example of this in Chapter 2, in the `<link>` that reloads a document:

```
<link
    caching="safe"
    next="http://myCompany.com/someDoc.vxml">
  <grammar>
    [
      (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 system will fetch the new version of the document. That is why we write `caching="safe"` in the `<link>`.

There is also a `caching` property that you can modify with the `<property>` tag. This property provides a default value to use for tags that do not specify the `caching` 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.mycompany.com/audio/welcome.wav">
  Welcome to the My 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>` with prompt counts 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 VXML Checker, Log Browser, and Tracing Tool to troubleshoot your documents.