



IBM WebSphere Voice Server Software Developers Kit (SDK) Programmer's Guide

Version 1.0

Printed in the USA

Note:

Before using this information and the product it supports, read the general information in “Notices” on page 187.

First Edition (October, 2000)

This edition applies to version 1, release 0, modification 0 of the IBM WebSphere Voice Server SDK and to all subsequent releases and modifications until otherwise indicated in new editions.

©Copyright International Business Machines Corporation 2000. All Rights Reserved.
Note to U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this Book	15	
Who Should Read This Book	15	
Related Publications	15	
Specifications and Standards	16	
Speech User Interface Design	17	
Server-side Programming	17	
Deployment Information	18	
How This Book Is Organized	18	
Document Conventions	19	
Introduction	21	
What Are Voice Applications?	21	
Why Create Voice Applications?	22	
What are Typical Types of Voice Applications?	22	
Queries	23	
Transactions	23	
What is VoiceXML?	24	
What Are the Advantages of VoiceXML?	25	
What is the IBM WebSphere Voice Server SDK?	27	
Contents of the SDK	27	
Deployment Platforms	28	
How Do You Create and Deploy A VoiceXML Application?	29	
How Do Users Access the Deployed Application?	30	
Chapter 1	Product Architecture	33
	Architectural Overview	33
	Speech Recognition Engine	35
	How It Works	35

Speech Recognition Accuracy	35
Audio Input Quality	36
Interface Design	36
Grammar Design	36
Types of Recognition Errors	37
Text-to-Speech (Speech Synthesis) Engine	38
How It Works	38
Text-to-Speech Markup Tags	38
Capabilities and Limitations of TTS	38
VoiceXML Browser	39
How It Works	39
Interactions with the DTMF Simulator	39
Interactions with Text-To-Speech and Speech Recognition Engines	40
Interactions with the Web Server and Enterprise Data Server	41
Chapter 2	
VoiceXML Language	43
Compatibility with the VoiceXML 1.0 Specification	43
VoiceXML Sample Applications	44
VoiceXML Overview	44
VoiceXML Elements and Attributes	45
Examples of IBM Extensions	50
Dialog Structure	51
Forms and Form Items	51
Fields	51
Subdialogs	51
Blocks	51
Types of Forms	52
Menus	52
Flow Control	53
Subdialogs	54
Comments	54
Built-in Field Types and Grammars	54
Recorded Audio	57
Using Prerecorded Audio Files	57
Recording Spoken User Input	58
Playing and Storing Recorded User Input	58
Document Fetching and Caching	58
Configuring Caching	58
Controlling Fetch and Cache Behavior	59

Preventing Caching	60
Events	60
Predefined Events	60
Application-specific Events	65
Recurring Events.	65
Variables and Expressions	65
Using ECMAScript.	66
Declaring Variables	66
Assigning and Referencing Variables.	67
Using Shadow Variables.	67
A Simple VoiceXML Example.	69
Static Content	70
Dynamic Content.	71
Grammars	72
Grammar Syntax	72
Grammar Header.	73
Grammar Body	73
Comments in Grammars	74
External and Inline Grammars	74
DTMF Grammars	75
Static Grammars	76
Dynamic Grammars.	77
Grammar Scope	78
Hierarchy of Active Grammars	78
Disabling Active Grammars	79
Resolving Ambiguities	79
Mixed Initiative Application and Form-level Grammars	79
Chapter 3	
VoiceXML Browser	81
VoiceXML Browser Features	81
“Say What You Hear”	82
Barge-in	83
Built-in Commands	84
Support for HTTP	85
Dynamically Generating VoiceXML Documents	85
Session Tracking and Cookie Support	86
The vsaudio and vstext Batch Files	87
Configurable Properties.	87
Specifying Properties During Application Development	90

Chapter 4

Specifying Properties for Your Deployed Application	91
Required Properties.	91
Starting the VoiceXML Browser	91
Command Line Interface	92
IBM WebSphere Studio	93
Using the Trace Mechanism	93
Sample vxml.log File	94
Format of Trace Entries	95
Stopping the VoiceXML Browser	97
Designing a Speech User Interface	99
Introduction	99
Design Methodology	100
Design Phase	100
Analyzing Your Users	101
Analyzing User Tasks	102
Making High-Level Decisions	103
Making Low-Level Decisions	103
Defining Information Flow	103
Creating the Initial Script	103
Identifying Application Interactions	104
Planning for Expert Users	104
Prototype Phase (Wizard of Oz Testing).	104
Test Phase	105
Identifying Recognition Problems	105
Identifying Any New Application Interactions	106
Identifying Any User Interface Breakdowns	106
Refinement Phase	106
Getting Started – High-Level Design Decisions	107
Selecting an Appropriate User Interface	108
Deciding on the Type and Level of Information.	109
Choosing Full-Duplex (Barge-In) or Half-Duplex Implementation.	109
Comparing Barge-in Detection Methods	111
Controlling Lombard Speech and the Stuttering Effect.	112
Weighing User and Environmental Characteristics	112
Minimizing the Need to Barge-in	113
Using Audio Formatting	113
Wording Prompts	113
Selecting Recorded Prompts or Synthesized Speech	114

Creating Recorded Prompts	114
Using TTS Prompts	114
Handling Unbounded Data.	115
Improving TTS Output.	115
Deciding Whether to Use Audio Formatting.	116
Designing Audio Tones	116
Applying Audio Formatting	116
Using Simple or Natural Command Grammars	118
Designing Simple Grammars	119
Evaluating the Need for Natural Command Grammars	119
Using Menu Flattening (Multiple Tokens In a Single User Utterance)	119
Promoting Consistency Through Built-in Grammars.	120
Adopting a Terse or Personal Prompt Style.	121
Weighing Demographic Factors.	122
Using Terse Prompts.	122
Using Personal Prompts	122
Allowing Only Speech Input or Speech plus DTMF	122
Choosing the Architecture of Mixed-Mode Application	122
Deciding When to Use DTMF	124
Structuring Mixed Mode Applications	124
Wording DTMF Prompts	124
Adopting a Consistent Set of Always-Active Navigation Commands	125
Selecting the Command List.	125
Constructing Always-Active Commands	126
Built-in Commands	126
Application-specific Commands	126
Backup	127
Exit	127
Help.	128
“List Commands”	128
Quiet/Cancel	128
Repeat	129
Start Over	129
“What-Can-I-Say-Now”	130
Using the Always-Active Commands.	130
Choosing Help Mode or Self-Revealing Help.	131
Maximizing the Benefits of Self Revealing Help.	132
Implementing Self-Revealing Help	133
“Bailing Out”	134
Mentioning Always-Active Commands.	134

Tracking the Reason for Bail Out	135
Getting Specific – Low-Level Design Decisions	136
Adopting a Consistent ‘Sound and Feel’	136
Designing Prompts	136
Standardizing Valid User Responses	137
Using Consistent Timing	137
Setting the Default Timeout Value	138
Managing Processing Time	138
Designing Consistent Dialogs	139
Writing Directive Prompts	139
Maintaining a Consistent Sound and Feel	139
Reusing Dialog Components	140
Creating Introductions	140
Welcome	140
Purpose	141
Always-Active List	141
Speak After Tone	141
Initial Prompt	141
Constructing Appropriate Menus and Prompts	142
Minimizing Transaction Time	142
Limiting Menu Length	142
Grouping Menu Items, Prompts, and Other Information	143
Separating Introductory/Instructive Text from Prompt Text	143
Separating Text for each Menu Item	144
Ordering Menu Items	144
Cycling Through Menu Items	145
Minimizing Prompt Length	146
Avoiding DTMF-style Prompts	146
Choosing the Right Words	147
Adopting User Vocabulary	147
Differentiating Between Data Prompts and Verbatim Prompts	148
Writing DTMF Prompts	149
Being Concise	149
Mixing Menu Choices and Form Data In a Single List	150
Avoiding Synonyms in Prompts	150
Promoting Valid User Input	150
Avoiding Spelling Input	151
Tapering Prompts	152
Confirming User Input	152
Providing Instructional Information	153

Managing Variable-length DTMF Input	153
“Feeding-forward” Information as Confirmation	153
Recovering from Errors	154
Designing and Using Grammars	155
Managing Tradeoffs	155
Word and Phrase Length	155
Vocabulary Robustness and Grammar Complexity	156
Number of Active Grammars	157
Improving Recognition Accuracy	157
Matching Partial Phrases	157
Improving Grammar Performance	158
Using Boolean and Yes/No Grammars	160
General Strategy	160
Recovering From a noinput Event	160
Recovering From a nomatch Event	160
When Initial Accuracy Is Paramount	161
Using the Built-in Phone Grammar	161
Testing Grammars	162
Providing Consistent Error Recovery	162
Managing Errors	163
Recovering from Out-of-Grammar Utterances	163
Confirming User Input	164
Understanding Spoke-Too-Soon and Spoke-Way-Too-Soon Errors	164
Minimizing STS and SWTS Errors	165
Recovering from STS and SWTS Errors	166
Implications for Self-Revealing Help in Half-Duplex Systems	166
Advanced User Interface Topics	167
Customizing Expertise Levels	167
Selecting an Appropriate User Interface Metaphor	168
Audio Desktop Metaphor	168
Personified Interface Metaphor	168
Conversational Participants Metaphor	169
Controlling the “Lost in Space” Problem	170
Minimizing the Number of Nested Menus	170
Using Audio Formatting	170
Providing State Information	170
Providing Bookmarks	170
Managing Audio Lists	171
Using a “Speak to Select” Strategy	171
Using List-Scanning Commands	171

	Additional Opportunities for Exploiting Audio Formatting	173
Chapter 5	Hints, Tips, and Best Practices	175
	VoiceXML Application Structure	175
	Deciding How to Group Dialogs	175
	Deciding Where to Define Grammars	176
	Fetching and Caching Resources for Improved Performance	177
	VoiceXML Coding Tips	177
	Accessing User Utterances	177
	Using <value> within <choice>	179
	XML Issues	180
	Using Relational Operators	180
	Specifying Character Encoding	180
	Security Issues	180
	Authenticating Users	181
	Using a Proxy Server	181
	Desktop Testing	181
	Simulating DTMF (Telephone Key) Input	182
	Running Text Mode or Automated Tests	182
	Using a Text File for Input	182
	Testing Built-in Field Types	183
	Timing Issues	185
Appendix A	Notices	187
	Copyright License	189
	Trademarks	189
Glossary		191

Figures

- Figure 1. Flow Chart of a Typical Call 30
- Figure 2. IBM WebSphere Voice Server SDK Interaction with Web and Enterprise Servers 33
- Figure 3. DTMF Simulator GUI 40
- Figure 4. Spoke-Too-Soon (STS) Error 165
- Figure 5. Spoke-Way-Too-Soon (SWTS) Error 165

Tables

Table 1.	VoiceXML Sample Files	28
Table 2.	Deployment Platforms	28
Table 3.	Speech Recognition Errors	37
Table 4.	Summary of VoiceXML Elements and Attributes	45
Table 5.	<ibmvoice> Example	50
Table 6.	<submit> Example	50
Table 7.	Built-in Types	55
Table 8.	Attributes for Document Fetching and Caching	59
Table 9.	Predefined Events and Event Handlers	61
Table 10.	Variable Scope	66
Table 11.	Shadow Variables	67
Table 12.	VoiceXML Example Using Menu and Form	69
Table 13.	Form Grammar Scope	78
Table 14.	“Say What You Hear” Example	83
Table 15.	Built-in VoiceXML Browser Commands	85
Table 16.	Configurable Properties for Starting the VoiceXML Browser	88
Table 17.	Required Properties for Starting the VoiceXML Browser	91
Table 18.	vxml.log Trace Codes	96
Table 19.	Influence of User Profile on Application Design	102
Table 20.	When to Use a Speech Interface	108
Table 21.	Full-duplex versus Half-duplex Implementation	110
Table 22.	Barge-in Detection Methods	111
Table 23.	Audio Formatting	117
Table 24.	Simple versus Natural Command Grammars	118
Table 25.	Prompt Styles	121
Table 26.	Mixed Input Modes	123
Table 27.	Recommended List of Always-Active Commands	126
Table 28.	Comparison of Help Styles	131

Table 29.	Recommended Maximum Number of Menu Items	143
Table 30.	Grammar Word/Phrase Length Tradeoffs	156
Table 31.	Vocabulary Robustness and Grammar Complexity Tradeoffs	156
Table 32.	Number of Active Grammars Tradeoffs	157
Table 33.	Grammar Design and Response Time	159
Table 34.	Error Recovery Techniques	163
Table 35.	Advanced Audio Formatting	173
Table 36.	Sample Input for Built-in Field Types	183

About This Book

This book provides information on using the IBM WebSphere Voice Server Software Developers Kit (SDK) to create and test voice applications written in VoiceXML. The resulting applications can then be deployed in a telephony environment using either the IBM WebSphere Voice Server with ViaVoice™ Technology or the IBM WebSphere Voice Server for DirectTalk, to provide voice access to Web-based data using standard telephony interfaces.

Who Should Read This Book

Read this book if you are:

- someone who wants to find out about the advantages of using VoiceXML to deliver Web-based voice services.
- an application developer interested in creating and testing VoiceXML applications.
- a content creator responsible for the creative aspects of VoiceXML applications.

Related Publications

Reference, design, and programming information for creating your voice applications is available from a variety of sources, as represented by the documents listed in this section.

Note: The guidelines and referenced publications presented in this book are for your information only, and do not in any manner serve as an endorsement of those materials. You alone are responsible for determining the suitability and applicability of this information to your needs.

Specifications and Standards

You may want to refer to the following sources for information on relevant specifications and standards:

- *ECMAScript Action Tags for JSGF*
installed as part of the IBM WebSphere Voice Server SDK and located in the file `%IBMVS%\docs\specs\ECMAScriptActionTagsforJSGF.html` (where `%IBMVS%` is an environment variable that contains the pathname of the IBM WebSphere Voice Server SDK installation directory)
- *ECMA Standard 262: ECMAScript Language Specification, 2nd Edition*, published by ECMA
<http://www.ecma.ch/ecma1/stand/ECMA-262.htm>
- *HTTP 1.1 Specification*
<http://www.ietf.org/rfc/rfc2616.txt>
- *HTTP State Management Mechanism (Cookie Specification)*
<http://www.w3.org/Protocols/rfc2109/rfc2109>
- *Java Speech API (JSAPI)*
<http://java.sun.com/products/java-media/speech>
- *Java Speech Grammar Format (JSGF) Specification*
<http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/index.html>
- *The International Phonetic Alphabet*, published by the International Phonetic Association
<http://www2.arts.gla.ac.uk/IPA/ipachart.html>
- *The Unicode Standard, Version 2.0*, The Unicode Consortium, Addison-Wesley Developers Press, 1996.
- *VoiceXML 1.0 Specification*
accessible from the Windows **Start** menu by choosing **Programs -> IBM WebSphere Voice Server SDK -> VoiceXML 1.0 Specification**; also available from the VoiceXML Forum Web site at **<http://www.voicexml.org>**

Speech User Interface Design

The speech user interface guidelines presented in this book are an evolving set of recommendations based on industry research and lessons learned in the process of developing our own VoiceXML and telephony applications. For more information, refer to speech industry literature and publications such as the following sources:

- *Audio System for Technical Readings (ASTeR)*
by T. V. Raman, a Ph.D. Thesis published by Cornell University, May 1994.
- *Auditory User Interfaces—Towards The Speaking Computer*
by T. V. Raman, published by Kluwer Academic Publishers, August 1997.
- “*Directing the Dialog: The Art of IVR*”
by Myra Hambleton, published in *Speech Technology*, Feb/Mar 2000.
- *How to Build a Speech Recognition Application — A Style Guide for Telephony Dialogues*
by Bruce Balentine and David P. Morgan, published by Enterprise Integration Group, San Ramon, CA, 1999.
- *Java Speech API Programmer’s Guide*, Chapter 3
<http://java.sun.com/products/java-media/speech>

Server-side Programming

Information on server-side programming is available from a number of sources, including the following:

- *The ASP (Active Server Pages) Resource Index*
<http://www.aspin.com/index/default.asp>
- *Building Blocks for CGI Scripts in Perl*
<http://www.cc.ukans.edu/~acs/docs/other/cgi-with-perl.shtml>
- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*
IBM Redbook SG24-5754-00
- *Developing an e-business Application for the IBM WebSphere Application Server*
IBM Redpiece SG24-5423-00
- *JavaServer Pages (JSP)*
<http://java.sun.com/products/jsp>

- *Java Servlet API*
<http://java.sun.com/products/servlet/>
- *The Front of IBM WebSphere Building e-business User Interfaces*
IBM Redbook SG24-5488-00
- *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*
IBM Redbook SG24-5755-00
- *Servlet/JSP/EJB Design and Implementation Guide*
IBM Redbook SG24-5754-00

Deployment Information

For information on deploying your voice applications, refer to the documentation provided with the applicable voice server product:

- *IBM WebSphere Voice Server Administrator's Guide*, for information on deploying VoiceXML applications in a Voice over IP environment
- *IBM WebSphere Voice Server for DirectTalk User's Guide*, for information on deploying VoiceXML applications in a DirectTalk environment

How This Book Is Organized

The Introduction provides an overview of creating voice applications using the IBM WebSphere Voice Server SDK.

Chapter 1 describes the system architecture and provides an overview of the components of the IBM WebSphere Voice Server SDK.

Chapter 2 introduces basic concepts and constructs of VoiceXML. For complete syntax, please refer to the VoiceXML 1.0 specification.

Chapter 3 contains information about the VoiceXML browser.

Chapter 4 presents user interface guidelines for developing voice applications.

Chapter 5 contains hints and tips for structuring, coding, and testing your VoiceXML applications.

Appendix A contains notices and trademark information.

Glossary defines key terminology used in this document.

Document Conventions

The following conventions are used to present information in this document:

<i>Italic</i>	Used for emphasis, to indicate variable text, and for references to other documents.
Bold	Used for names of VoiceXML elements, attributes, and events. Also used for properties, file names, URLs, and user interface controls such as command buttons and menus.
Courier Regular	Used for sample code.
<text>	Used for editorial comments in sample scripts.
[!!!!]	Used in sample scripts to represent a tone indicating that it is the user's turn to speak. A turn-taking tone is strongly recommended if the system is half-duplex, but is not required for systems that have barge-in enabled. For more information, refer to “Using Audio Formatting” on page 113 .

Introduction

The IBM WebSphere Voice Server SDK enables you to create Web-based, voice applications and test them on a desktop workstation before deploying them in a telephony environment.

This introduction addresses the following questions:

- [“What Are Voice Applications?”](#) — See [page 21](#).
- [“Why Create Voice Applications?”](#) — See [page 22](#).
- [“What are Typical Types of Voice Applications?”](#) — See [page 22](#).
- [“What is VoiceXML?”](#) — See [page 24](#).
- [“What Are the Advantages of VoiceXML?”](#) — See [page 25](#).
- [“What is the IBM WebSphere Voice Server SDK?”](#) — See [page 27](#).
- [“How Do You Create and Deploy A VoiceXML Application?”](#) — See [page 29](#).
- [“How Do Users Access the Deployed Application?”](#) — See [page 30](#).

What Are Voice Applications?

Voice applications are applications in which the input and/or output are through a spoken, rather than a graphical, user interface. The application files can reside on the local system, an intranet, or the Internet. Users can access the deployed applications anytime, anywhere, from any telephony-capable device, and you can design the applications to restrict access only to those who are authorized to receive it.

“Voice-enabling the World Wide Web” does *not* simply mean:

- Using spoken commands to tell a visual browser to look up a specific Web address or go to a particular bookmark.
- Having a visual browser throw away the graphics on a traditional visual Web page and read the rest of the information aloud.
- Converting the bold or italics on a visual Web page to some kind of emphasized speech.

Rather, voice applications provide an easy and novel way for users to surf or shop on the Internet — “browsing by voice.” Users can interact with Web-based data (that is, data available via Web-style architecture such as servlets, ASPs, JSPs, Java Beans, CGI scripts, etc.) using speech, rather than a keyboard and mouse.

The form that this spoken data takes is often *not* identical to the form it takes in a visual interface, due to the inherent differences between the interfaces. For this reason, transcoding (that is, using a tool to automatically convert HTML files to VoiceXML), may not be the most effective way to create voice applications.

Why Create Voice Applications?

Until recently, the World Wide Web has relied exclusively on visual interfaces to deliver information and services to users via computers equipped with a monitor, keyboard, and pointing device. In doing so, a huge potential customer base has been ignored: people who (due to time, location, and/or cost constraints) do not have access to a computer.

Many of these people do, however, have access to a telephone. Providing “conversational access” (that is, spoken input and audio output over a telephone) to Web-based data will permit companies to reach this untapped market. Users benefit from the convenience of using the mobile Internet for self-service transactions, while companies enjoy the Web’s relatively low transaction costs. And, unlike applications that rely on DTMF (telephone keypress) input, voice applications can be used in a hands-free or eyes-free environment, as well as by customers with rotary pulse telephone service or telephones in which the keypad is on the handset.

What are Typical Types of Voice Applications?

Voice applications will typically fall into one of the following categories:

- “Queries”
- “Transactions”

Queries

In this scenario, a customer calls into a system to retrieve information from a Web-based infrastructure.

The system guides the customer through a series of menus and forms by playing instructions, prompts, and menu choices using prerecorded audio files or synthesized speech.

The customer uses spoken commands and/or DTMF (telephone keypress) input to make menu selections and fill in form fields.

Based on the customer's input, the system locates the appropriate records in a back-end enterprise database. The system presents the desired information to the customer, either by playing back prerecorded audio files or by synthesizing speech based on the data retrieved from the database.

Examples of this type of self-service interaction include applications or voice portals providing weather reports, movie listings, stock quotes, health care provider listings, and customer service information (Web call centers).

Transactions

In this scenario, a customer calls into a system to execute specific transactions with a Web-based back-end.

The system may prompt the customer to log in using some type of ID and/or password, and then guides the customer to provide the data required for the transaction.

The system plays instructions, prompts, and menu choices using prerecorded audio files or synthesized speech, and the customer responds using spoken commands and/or DTMF (telephone keypress) input.

Based on the customer's input, the system conducts the transaction and updates the appropriate records in a back-end enterprise database. Typically the system also reports back to the customer, either by playing back prerecorded audio files or by synthesizing speech based on the information in the database records.

Examples of this type of self-service interaction include applications or voice portals for employee benefits, employee timecard submission, financial transactions, travel reservations, calendar appointments, customer relationship management (eRM), sales automation, and order management.

What is VoiceXML?

The Voice eXtensible Markup Language (VoiceXML) is an XML-based markup language for creating distributed voice applications, much as HTML is a markup language for creating distributed visual applications.

VoiceXML is an emerging industry standard that has been defined by the VoiceXML Forum (<http://www.voicexml.org>), of which IBM is a founding member. It has been accepted for submission by the World Wide Web Consortium (W3C) as a standard for voice markup on the Web.

The VoiceXML language enables Web developers to use a familiar markup style and Web server-side logic to deliver voice content to the Internet. The VoiceXML applications you create can interact with your existing back-end business data and logic.

Using VoiceXML, you can create Web-based voice applications that users can access by telephone. VoiceXML supports dialogs that feature:

- spoken input
- DTMF (telephone key) input
- recording of spoken input
- synthesized speech output (“text-to-speech”)
- recorded audio output
- dialog flow control
- scoping of input

What Are the Advantages of VoiceXML?

While you could certainly build voice applications without using a voice markup language and a speech browser (for example, by writing your applications directly to a speech API), using VoiceXML and a VoiceXML browser provide several important capabilities:

- VoiceXML is a markup language that makes building voice applications easier, in the same way that HTML simplifies building visual applications. VoiceXML also reduces the amount of speech expertise that developers must have.
- VoiceXML applications can use the same existing back-end business logic as their visual counterparts, enabling voice solutions to be introduced to new markets quickly. Current and long-term development and maintenance costs are minimized by leveraging the Web design skills and infrastructures already present in the enterprise. Customers can benefit from a consistency of experience between voice and visual applications.
- VoiceXML implements a client-server paradigm, where a Web server provides VoiceXML documents that contain dialogs to be interpreted and presented to the user; the user's responses are submitted to the Web server, which responds by providing additional VoiceXML documents, as appropriate. VoiceXML allows you to request documents and submit data to server scripts using *Universal Resource Indicators* (URIs). VoiceXML documents can be static, or they can be dynamically generated by CGI scripts, Java Beans, ASPs, JSPs, Java servlets, or other server-side logic.
- Unlike a proprietary Interactive Voice Response (IVR) system, VoiceXML provides an open application development environment that generates portable applications. This makes VoiceXML a cost-effective alternative for providing voice access services.
- Most installed IVR systems today accept input from the telephone keypad only; in contrast, VoiceXML is designed predominantly to accept spoken input, but can also accept DTMF (telephone keypad) input if desired. As a result, VoiceXML helps speed up customer interactions by providing a more natural interface that replaces the traditional hierarchical IVR menu tree with a streamlined dialog using a flattened command structure.
- VoiceXML directly supports networked and Web-based applications, meaning that a user at one location can access information or an application provided by a server at another geographically or organizationally distant location. This capitalizes on the connectivity and commerce potential of the World Wide Web.

- Using a single VoiceXML browser to interpret streams of markup language originating from multiple locations provides the user with a seamless conversational experience across independent applications. For example, a voice portal application might allow a user to temporarily suspend an airline purchase transaction to interact with a banking application on a different server to check an account balance.
- VoiceXML supports local processing and validation of user input.
- VoiceXML supports playback of prerecorded audio files.
- VoiceXML supports recording of user input; the resulting audio can be played back locally or uploaded to the server for storage, processing, or playback at a later time.
- VoiceXML defines a set of events corresponding to such activities as a user request for help, the failure of a user to respond within a timeout period, and an unrecognized user response. A VoiceXML application can provide catch elements that respond appropriately to a given event for a particular context.
- VoiceXML supports context-specific and tapered help, using a system of events and catch elements. Help can be tapered by specifying a count for each event handler, so that different catch blocks are executed depending on the number of times that the event has occurred in the specified context; this can be used to provide increasingly more detailed messages each time the user asks for help. For more information, see [“Choosing Help Mode or Self-Revealing Help” on page 131](#).
- VoiceXML supports subdialogs, which are roughly the equivalent of function or method calls. Subdialogs can be used to provide a disambiguation or confirmation dialog, and to create reusable dialog components. For more information, see [“Subdialogs” on page 54](#).

What is the IBM WebSphere Voice Server SDK?

The IBM WebSphere Voice Server SDK is a toolkit that brings support for VoiceXML to Web application development activities — in essence, providing a spoken equivalent to visual browsing.

With the IBM WebSphere Voice Server SDK, you can create and test Web-based voice applications. The SDK uses the workstation's speakers to play audio output; you can input data using the workstation's microphone, prerecorded audio files, or the IBM WebSphere Voice Server SDK's DTMF Simulator (to simulate any telephone key input). The SDK also supports text-mode and automated testing.

When you deploy your voice applications, users can interact with them through spoken or DTMF (telephone keypress) commands and audio output.

Contents of the SDK

The SDK includes:

- A speech browser that interprets VoiceXML markup. This VoiceXML browser includes a DTMF Simulator to generate simulated telephone keypress input during desktop testing.
- IBM ViaVoice Speech Recognition and Text-To-Speech engines for accepting voice input and generating synthesized speech output.
- Telephony acoustic models to approximate the speech recognition behavior of applications deployed in a telephony environment.
- An audio setup utility to ensure that your microphone and speakers are properly configured for use with this product.
- User documentation.
- Sample VoiceXML files, as shown in [Table 1 on page 28](#). The samples and their respective documentation are installed in subdirectories of `%IBMVS%\samples` (where `%IBMVS%` is an environment variable that contains the pathname of the installation directory).

Table 1. VoiceXML Sample Files

Sample Name	Description
<i>AudioSample</i>	VoiceXML sample that allows you to verify that your audio input and output devices are working correctly. You can run the Audio Sample from the Windows Start menu by choosing Programs -> IBM WebSphere Voice Server SDK -> Audio Sample .
<i>GrammarBuilder</i>	Servlet that you can customize to help you construct grammar files from information in enterprise databases. You can use the resulting grammar files in your VoiceXML applications.
<i>VoiceSnoopServlet</i>	Servlet that is analogous to 'snoop' programs used with a visual browser. It returns dynamic audio information to the VoiceXML browser about the requested URL, request method, URI, request protocol, path, query, server, and remote user.

Deployment Platforms

When your applications are ready to deploy in a telephony environment, system administrators can use either the IBM WebSphere Voice Server for DirectTalk or the IBM WebSphere Voice Server with ViaVoice Technology to configure, deploy, monitor, and manage the applications on a dedicated voice server. These products are described in [Table 2](#).

Table 2. Deployment Platforms

Product	Description
<i>IBM WebSphere Voice Server for DirectTalk</i>	A deployment environment in which the VoiceXML browser is a Java application running on DirectTalk; DirectTalk's telephone network connection provides the audio channel(s) for the VoiceXML browser(s).
<i>IBM WebSphere Voice Server with ViaVoice Technology</i>	A deployment environment in which the VoiceXML browser is built into a server infrastructure with a voice over Internet protocol (VoIP) telephony front-end; a VoIP telephony gateway connected to the telephone network routes the audio to/from the VoiceXML browser(s) running on the voice server.

How Do You Create and Deploy A VoiceXML Application?

1. An application developer uses the IBM WebSphere Voice Server SDK and optionally a Web site development tool (such as IBM WebSphere Studio) to create a voice application written in VoiceXML. The VoiceXML pages can be static, or they can be dynamically generated using server-side logic such as CGI scripts, Java Beans, ASPs, JSPs, Java servlets, etc.

Note: While you can certainly write VoiceXML applications using your favorite text editor, you may find it more convenient to use a graphical development environment that helps you create and manage VoiceXML files.

IBM WebSphere Studio version 3.5 has been enhanced to support VoiceXML files: it includes a VXML Editor with code assist features, wizards capable of generating Java Server Pages for use with VoiceXML files, and the ability to preview VoiceXML files using the IBM WebSphere Voice Server SDK's VoiceXML browser. For more information, refer to the IBM WebSphere Studio documentation.

2. (Optional) A system administrator uses a Web application server program (such as IBM WebSphere Application Server) to configure and manage a Web server.
3. (Optional) The developer publishes the VoiceXML application (including VoiceXML pages, grammar files, any prerecorded audio files, and any server-side logic) to the Web server.
4. The developer uses a desktop workstation and the IBM WebSphere Voice Server SDK to test the VoiceXML application running on the Web server or local disk, pointing the VoiceXML browser to the appropriate starting VoiceXML page.
5. A telephony expert configures the telephony infrastructure, as described in the product documentation for the applicable deployment platform:
 - For the IBM WebSphere Voice Server for DirectTalk, this involves installing the prerequisites.
 - For the IBM WebSphere Voice Server with ViaVoice Technology, this involves configuring the telephony hardware and connecting it to the voice server.
6. The system administrator uses the IBM WebSphere Voice Server for DirectTalk or the IBM WebSphere Voice Server with ViaVoice Technology to configure, deploy, monitor, and manage a dedicated voice server.

Note: Refer to the documentation for the applicable deployment platform for any migration tasks that might be required.

7. The developer uses a real telephone to test the VoiceXML application running on the voice server.

How Do Users Access the Deployed Application?

Once your voice applications are deployed, users simply dial the telephone number that you provide and are connected to the corresponding voice application. [Figure 1](#) shows a flow chart of a typical call.

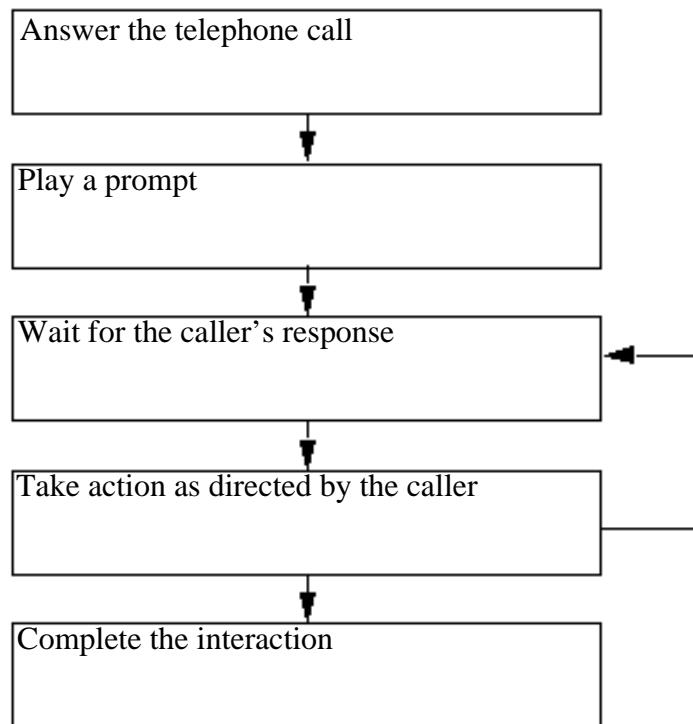


Figure 1. Flow Chart of a Typical Call

1. A user dials the telephone number you provide. The DirectTalk system (in the case of the IBM WebSphere Voice Server for DirectTalk) or VoIP gatekeeper or gateway (in the case of the IBM WebSphere Voice Server with ViaVoice Technology) answers the call and executes the application referenced by the dialed phone number.
2. The voice server plays a greeting to the caller and prompts the caller to indicate what information he or she wants.
 - The application can use prerecorded greetings and prompts, or synthesize them from text using the text-to-speech engine (see [“Text-to-Speech \(Speech Synthesis\) Engine” on page 38](#)).
 - If the application supports barge-in (see [“Barge-in” on page 83](#)), the caller can interrupt the prompt if he or she already knows what to do.
3. The application waits for the caller’s response for a set period of time.
 - The caller can respond either by speaking or by pressing one or more keys on a DTMF telephone keypad, depending on the types of responses expected by the application.
 - If the response does not match the criteria defined by the application (such as the specific word, phrase, or digits), the voice application can prompt the caller to enter the response again, using the same or different wording.
 - If the waiting period has elapsed and the caller has not responded, the application can prompt the caller again, using the same or different wording.
4. The application takes whatever action is appropriate to the caller’s response. For example, the application might:
 - Update information in a database.
 - Retrieve information from a database and speak it to the caller.
 - Store or retrieve a voice message.
 - Launch another application.
 - Play a help message.After taking action, the application prompts the caller with what to do next.
5. The caller or the application can terminate the call. For example:
 - The caller can terminate the interaction at any time, simply by hanging up; the voice server can detect if the caller hangs up and can disconnect itself.
 - If the application permits, the caller can use a command to explicitly indicate that the interaction is over (for example, by saying “Exit”).
 - If the application has finished running, it can play a closing message and then disconnect.

The IBM WebSphere Voice Server SDK consists of the following components:

- “[Speech Recognition Engine](#)” — See [page 35](#).
- “[Text-to-Speech \(Speech Synthesis\) Engine](#)” — See [page 38](#).
- “[VoiceXML Browser](#)” (includes the DTMF Simulator) — See [page 39](#).

This chapter provides an overview of the system architecture and then discusses the individual components of the IBM WebSphere Voice Server SDK.

Architectural Overview

[Figure 2](#) shows how the IBM WebSphere Voice Server SDK interfaces with information stored on Web application servers and in back-end enterprise databases.

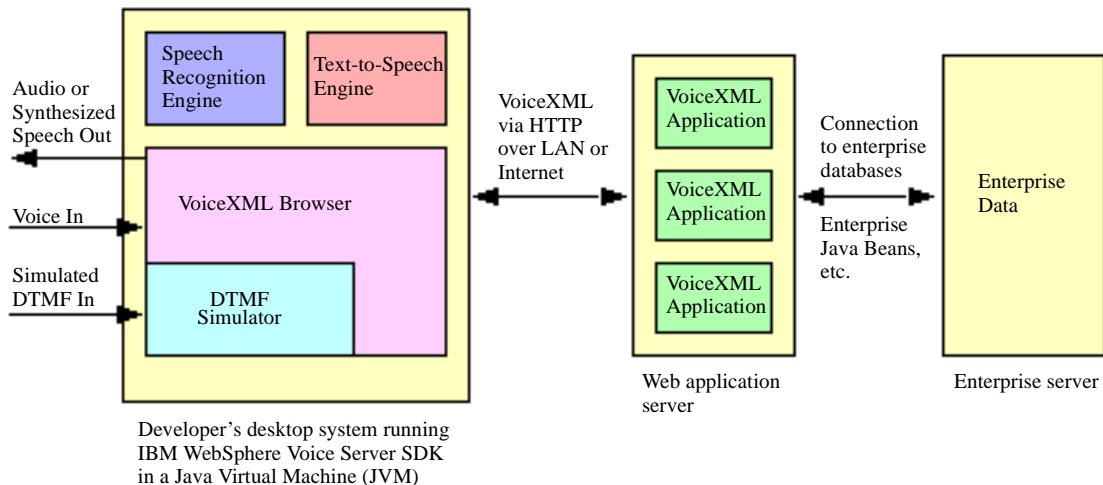


Figure 2. IBM WebSphere Voice Server SDK Interaction with Web and Enterprise Servers

To use the IBM WebSphere Voice Server SDK, you start the VoiceXML browser (as described in [“Starting the VoiceXML Browser” on page 91](#)) and pass it the URI of the first VoiceXML document in your application. The VoiceXML applications will generally reside on a Web application server, although they could reside on your desktop workstation during early testing; the VoiceXML browser uses HTTP over a LAN or the Internet to fetch the documents.

Each VoiceXML document specifies an interaction (or “dialog”) between the user and the application. The VoiceXML browser interprets and renders the VoiceXML document, using your desktop workstation’s speakers to play the prompts, instructions, and menu choices specified in the document. The information played can be:

- Prerecorded audio files
- Synthesized (by the text-to-speech engine) from text specified in your VoiceXML files
- Synthesized from information that is dynamically generated from your enterprise databases

Based on the state of the dialog, the VoiceXML browser also enables and disables speech recognition grammars, as specified by your VoiceXML application.

You respond to the prompts and make menu selections by speaking into a microphone connected to your desktop workstation. The information you speak is passed to the speech recognition engine, which compares it to the list of valid inputs specified in the currently active grammars.

If your VoiceXML application will support DTMF (telephone keypress) input when it is deployed in a telephony environment, you can use the IBM WebSphere Voice Server SDK’s DTMF Simulator to simulate this input on your desktop workstation, as described in [“Interactions with the DTMF Simulator” on page 39](#).

Based on the input you provide, the VoiceXML browser proceeds with the interaction specified by your VoiceXML application. For example, the VoiceXML browser might continue to the next dialog in the document, fetch a new VoiceXML document, or submit information to the Web server for processing. The Web server can use server-side programs to locate or update records in a back-end enterprise database and return the information to the VoiceXML browser. The VoiceXML browser presents this information to you either by playing back prerecorded audio files or by synthesizing speech based on the data retrieved from the database.

When a dialog does not specify a transition to another dialog, the VoiceXML browser exits and the session terminates.

Speech Recognition Engine

Speech recognition is the ability of a computer to decode human speech and convert it to text.

How It Works

To convert spoken input to text, the computer must first parse the input audio stream and then convert that information to text output. In the IBM WebSphere Voice Server SDK, this is done by the *IBM ViaVoice Speech Recognition engine*. The high-level process looks like this:

1. The application developer creates a series of speech recognition *grammars* defining the words and phrases that can be spoken by the user, and specifies where each grammar should be active within the application.
2. When the application runs, the speech recognition engine processes the incoming audio signal and compares the sound patterns to the patterns of basic spoken sounds, trying to determine the most probable combination that represents the audio input.
3. Finally, the speech recognition engine compares the sounds to the list of words and phrases in the active grammar(s). Only words and phrases in the *active grammars* are considered as possible speech recognition candidates.

Speech Recognition Accuracy

The key determinants of speech recognition accuracy are:

- "Audio Input Quality"
- "Interface Design"
- "Grammar Design"

Audio Input Quality

The quality of audio input is a key determinant of speech recognition accuracy. Audio quality is influenced by such factors as:

- The choice of input device. In the case of the IBM WebSphere Voice Server SDK, the input device will be a microphone connected to a desktop workstation. For applications deployed using the IBM WebSphere Voice Server for DirectTalk or the IBM WebSphere Voice Server with ViaVoice Technology, the input device could be a regular telephone, cordless telephone, speaker phone, or cellular telephone.
- Speaking environment, which could be in a car, outdoors, in a crowded room, or in a quiet office.
- Certain user characteristics such as accent, fluency in the input language, and any atypical pronunciations.

While many of these factors may be beyond your control as an application developer, you should nevertheless consider their implications when you design your applications.

Users will achieve the best possible speech recognition with a high-quality input device that gives good signal-to-noise ratio. For desktop testing, you should use one of the microphones listed at <http://www-4.ibm.com/software/speech/support/testvvmil.html>. For example, the Andrea Electronics array microphone works well in quiet environments, and the Sennheiser head-word microphone works well in noisy environments.

Speech clarity is a significant contributor to audio quality. Adult native speakers who speak clearly (without over-enunciating or hesitating) and position the microphone or telephone properly achieve the best recognition; other demographic groups may see somewhat variable performance.

Interface Design

The design of the application interface has a major influence on speech recognition accuracy. Guidelines for creating a speech user interface are in [Chapter 4 “Designing a Speech User Interface”](#).

Grammar Design

Since only words, phrases, and DTMF key sequences from active grammars are considered as possible speech recognition candidates, what you choose to put in a grammar and when you choose to make that grammar active have a major impact on speech recognition accuracy. For more information, see [“Designing and Using Grammars” on page 155](#).

Types of Recognition Errors

There are three basic types of recognition errors, as shown in [Table 3](#):

Table 3. Speech Recognition Errors

Description of Error	Possible Causes
<i>The speech recognition returns a result that does not match what the user actually said.</i>	This can have many causes, including: <ul style="list-style-type: none"> • The audio quality is poor. • Multiple choices in the active grammars sound similar, such as Newark and New York in a grammar of United States airports. • The user utterance was not in any of the active grammars, but something from an active grammar sounded similar. • The user has a strong or unusual accent. • The user paused before finishing the intended utterance.
<i>The speech recognition engine did not understand what the user said well enough to return anything at all.</i>	This type of error can occur in situations similar to those described above, plus: <ul style="list-style-type: none"> • The user spoke too soon in a half-duplex implementation, so the beginning of the audio stream was not captured and sent to the recognition engine. See “Understanding Spoke-Too-Soon and Spoke-Way-Too-Soon Errors” on page 164. • The user stuttered or mumbled.
<i>The user did not actually speak, but the speech recognition engine returned a result.</i>	This type of error can have many causes, including: <ul style="list-style-type: none"> • The environment included a lot of background noise. • The user was conducting another conversation while using the application. • The user made a non-speech utterance, such as a cough.

Text-to-Speech (Speech Synthesis) Engine

“Text-to-speech” is the ability of a computer to “read aloud” (that is, to generate spoken output from text input). Text-to-speech is often referred to as TTS or speech synthesis.

How It Works

To generate synthesized speech, the computer must first parse the input text to determine its structure and then convert that text to spoken output. In the IBM WebSphere Voice Server SDK, this is done by the *IBM ViaVoice Text-To-Speech (TTS) engine*.

Text-to-Speech Markup Tags

You can improve the quality of TTS output by using the speech markup elements provided by the VoiceXML language, as described in [“Improving TTS Output” on page 115](#).

Capabilities and Limitations of TTS

- TTS prompts are easier to maintain and modify than recorded audio prompts. For this reason, TTS is typically used during application development.
- TTS is also a powerful tool for use when the data to be spoken is “*unbounded*” (that is, not known in advance) and cannot therefore be prerecorded.
- However, TTS cannot yet mimic the complete naturalness of human speech. So, while TTS may be a necessary or convenient option when the input text is dynamic or under development, prerecorded audio is usually a better choice for deployed applications in which the input text is static.
- Some users may initially experience minor difficulties in understanding the synthesized speech; however, these difficulties tend to diminish with increased exposure to the system.
- Some users may find the synthesized speech a bit “robotic”; however, this does not usually interfere with their ability to understand the information being presented.

VoiceXML Browser

The VoiceXML browser component of the IBM WebSphere Voice Server SDK is the implementation of the interpreter context as defined in the VoiceXML 1.0 specification.

Detailed usage information for the VoiceXML browser is in [Chapter 3 “VoiceXML Browser”](#).

How It Works

One of the primary functions of the VoiceXML browser is to fetch documents to process. The request to fetch a document can be generated either by the interpretation of a VoiceXML document, or in response to an external event.

The VoiceXML browser manages the dialog between the application and the user by playing audio prompts, accepting user inputs, and acting on those inputs. The action might involve jumping to a new dialog, fetching a new document, or submitting user input to the Web server for processing.

The VoiceXML browser is a Java application. The Java console provides trace information on the prompts played, resource files fetched, and user input recognized; other than this and the DTMF Simulator GUI, there is no visual interface. For more information, see [“Using the Trace Mechanism” on page 93](#) and [“Interactions with the DTMF Simulator” on page 39](#), respectively.

Interactions with the DTMF Simulator

When your VoiceXML application is deployed in a telephony environment, you may want to allow users to provide DTMF (telephone keypress) input in addition to spoken input. The DTMF Simulator is a GUI tool that enables you to simulate DTMF tones on your desktop workstation.

The VoiceXML browser starts the DTMF Simulator automatically, unless you specify the **-Dvxml.gui=false** Java system property when you start the VoiceXML browser. If you close the DTMF Simulator GUI window, the only way to restart it is to stop and restart the VoiceXML browser.

The DTMF Simulator plus your desktop microphone and speakers take the place of a telephone during desktop testing, allowing you to debug your VoiceXML applications without having to connect to telephony hardware and the PSTN (Public Switched Telephone Network) or cellular GSM (Global System for Mobile Communication).

Using the DTMF Simulator, you can simulate a telephone keypress event by pressing the corresponding key on the computer keyboard or clicking on the corresponding button on the DTMF Simulator GUI, shown in [Figure 3](#).

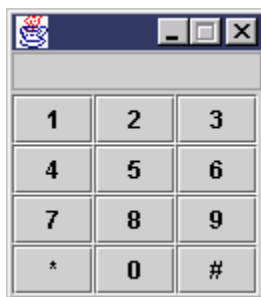


Figure 3. DTMF Simulator GUI

For example, if the application prompt is “Press 5 on your telephone keypad”, you can simulate a user response during desktop testing by clicking the 5 button on the DTMF Simulator GUI or pressing the 5 key on the computer keyboard while the cursor focus is in the DTMF Simulator GUI window. The VoiceXML browser will interpret your input as a 5 pressed on a DTMF telephone keypad. If the length of valid DTMF input strings is variable, use the # key to terminate DTMF input.

Interactions with Text-To-Speech and Speech Recognition Engines

During initialization, the VoiceXML browser starts the TTS and speech recognition engines.

Note: The VoiceXML browser uses telephony acoustic models in order to simulate the behavior of the final deployed telephony application as closely as possible in a desktop environment.

As the VoiceXML browser processes a VoiceXML document, it plays audio prompts using text-to-speech or recorded audio; for text-to-speech output, it interacts with the TTS engine to convert the text into audio.

Based on the current dialog state, the VoiceXML browser enables and disables speech recognition grammars. When the VoiceXML browser receives user audio input, the speech recognition engine decodes the input stream, checks for valid user utterances as defined by the currently active speech recognition grammar(s), and returns the results to the VoiceXML browser. The VoiceXML browser uses the recognition results to fill in form items or select menu options in the VoiceXML application. If the input is associated with a `<record>` element in the VoiceXML document, the VoiceXML browser stores the recorded audio.

As the VoiceXML browser makes transitions to new dialogs or new documents, it enables and disables different speech recognition grammars, as specified by the VoiceXML application. As a result, the list of valid user utterances changes.

Interactions with the Web Server and Enterprise Data Server

You can publish your VoiceXML applications to any Web server running on any platform; however, we recommend the IBM WebSphere Application Server version 3.5 or later, which has been enhanced to receive VoiceXML pages published from IBM WebSphere Studio version 3.5 or later.

Note: If you choose to use a different Web server, you may need to configure it to recognize that *.vxml files are of MIME type “application/x-vxml” (preferred form), “application/vxml”, “text/x-vxml”, or “text/vxml”. On some Web servers, failure to do so may generate a “file not found” exception in the VoiceXML browser.

When you start the VoiceXML browser, the VoiceXML browser sends an HTTP request over the LAN or Internet to request an initial VoiceXML document from the Web server. The requested VoiceXML document can contain static information, or it can be generated dynamically from data stored in an enterprise database using the same type of server-side logic (CGI scripts, Java Beans, ASPs, JSPs, Java servlets, etc.) that you use to generate dynamic HTML documents. See [“Dynamically Generating VoiceXML Documents” on page 85](#).

The VoiceXML browser interprets and renders the document. Based on the user's input, the VoiceXML browser may request a new VoiceXML document from the Web server, or may send data back to the Web server to update information in the back-end database. The important thing is that the mechanism for accessing your back-end enterprise data does not need to change; your VoiceXML applications can access the same information from your enterprise servers that your HTML applications do.

This chapter provides a *brief introduction* to basic VoiceXML concepts and constructs, and describes IBM's implementation of version 1.0 of VoiceXML. For a complete description of the functionality of the language, refer to the VoiceXML 1.0 specification; the information in this chapter is *NOT* a substitute for thoroughly reading the VoiceXML 1.0 specification.

You can access the VoiceXML 1.0 specification from the Windows **Start** menu by choosing **Programs** -> **IBM WebSphere Voice Server SDK** -> **VoiceXML 1.0 Specification**, or you can download a copy from the VoiceXML Forum Web site at <http://www.voicexml.org>.

This chapter discusses the following topics:

- "Compatibility with the VoiceXML 1.0 Specification" — See [page 43](#).
- "VoiceXML Sample Applications" — See [page 44](#).
- "VoiceXML Overview" — See [page 44](#).
- "A Simple VoiceXML Example" — See [page 69](#).
- "Grammars" — See [page 72](#).

Compatibility with the VoiceXML 1.0 Specification

The VoiceXML browser in the IBM WebSphere Voice Server products supports a subset of the VoiceXML Version 1.0 specification, as documented in this chapter.

Additionally, IBM has extended the VoiceXML 1.0 specification with a set of features to enhance the usability and functionality of the IBM WebSphere Voice Server products; these extensions are also documented in this chapter.

VoiceXML Sample Applications

The IBM WebSphere Voice Server SDK includes the sample VoiceXML applications shown in [Table 1 on page 28](#). The samples are located in subdirectories of `%IBMVS%\samples` (where `%IBMVS%` is an environment variable that contains the pathname of the installation directory). Each sample has its own documentation.

You can run the Audio Sample from the Windows **Start** menu by choosing **Programs -> IBM WebSphere Voice Server SDK -> Audio Sample**. To run the other samples, refer to their respective documentation.

VoiceXML Overview

VoiceXML is an XML-based application, meaning that it is defined as a set of XML tags.

This section introduces the following VoiceXML concepts and constructs:

- ["VoiceXML Elements and Attributes"](#) — See [page 45](#).
- ["Dialog Structure"](#) — See [page 51](#).
- ["Built-in Field Types and Grammars"](#) — See [page 54](#).
- ["Recorded Audio"](#) — See [page 57](#).
- ["Document Fetching and Caching"](#) — See [page 58](#).
- ["Events"](#) — See [page 60](#).
- ["Variables and Expressions"](#) — See [page 65](#).

Speech recognition grammars are a key component of VoiceXML application design; they are discussed separately in ["Grammars" on page 72](#).

VoiceXML Elements and Attributes

Table 4 on page 45 lists the VoiceXML elements (including IBM extensions) and provides implementation details specific to the VoiceXML browser in the IBM WebSphere Voice Server products.

Table 4. Summary of VoiceXML Elements and Attributes

Element	Description	Implementation Details
<assign>	Assigns a value to a variable.	Supported as documented in the VoiceXML 1.0 specification.
<audio>	Plays an audio file within a prompt.	Supported as documented in the VoiceXML 1.0 specification. The supported audio format is “ audio/basic ” (an 8KHz .au file). The IBM WebSphere Voice Server SDK uses JMF to record and play back audio; the JMF is included in the IBM WebSphere Voice Server SDK. The IBM WebSphere Voice Server for DirectTalk uses the DirectTalk system for audio playback. The IBM WebSphere Voice Server with ViaVoice Technology uses the telephony H.323 stack for audio playback.
<block>	Specifies a form item containing non-interactive executable content.	Supported as documented in the VoiceXML 1.0 specification.
<break>	Inserts a pause in TTS output.	Supported as documented in the VoiceXML 1.0 specification.
<catch>	Catches an event.	Supported as documented in the VoiceXML 1.0 specification.
<choice>	Specifies a menu item.	Supported as documented in the VoiceXML 1.0 specification.
<clear>	Clears a variable.	Supported as documented in the VoiceXML 1.0 specification.

Element	Description	Implementation Details
<disconnect>	Causes the VoiceXML browser to disconnect from a user telephone session. Applicable only to a telephony deployment environment, not the desktop development environment.	Not implemented in this release.
<div>	Identifies the type of text for TTS output.	Supported as documented in the VoiceXML 1.0 specification.
<dtmf>	Specifies a DTMF grammar.	Supported type is “ application/x-jsgf ” as defined in Appendix D of the VoiceXML 1.0 specification.
<else>	Conditional statement used with the <if> element.	Supported as documented in the VoiceXML 1.0 specification.
<elseif>	Conditional statement used with the <if> element.	Supported as documented in the VoiceXML 1.0 specification.
<emp>	Specifies the emphasis for TTS output.	Supported as documented in the VoiceXML 1.0 specification.
<enumerate>	Shorthand construct that causes the VoiceXML browser to speak the text of each <choice> element when presenting the list of menu selections to the user.	When using the <enumerate> element to play menu choices via the text-to-speech engine, you do not have to add punctuation to control the length of the pauses between <choice> elements; the VoiceXML browser will automatically add the appropriate pauses and intonations when speaking the prompts.
<error>	Catches an error event.	Supported as documented in the VoiceXML 1.0 specification.
<exit>	Exits a VoiceXML browser session.	As per the VoiceXML 1.0 specification, the <exit> element returns control to the interpreter, which determines what action to take. In the IBM WebSphere Voice Server implementation, the expr and namelist attributes are ignored.
<field>	Defines an input field in a form.	Supported as documented in the VoiceXML 1.0 specification.

Element	Description	Implementation Details
<filled>	Specifies an action to execute when a field is filled.	Supported as documented in the VoiceXML 1.0 specification.
<form>	Specifies a dialog for presenting and gathering information.	Supported as documented in the VoiceXML 1.0 specification. The VoiceXML browser also supports mixed initiative dialogs using a mechanism based on ECMAScript Action Tags. See “ Mixed Initiative Application and Form-level Grammars ” on page 79 for details.
<goto>	Specifies a transition to another dialog or document.	Supported as documented in the VoiceXML 1.0 specification.
<grammar>	Defines a speech recognition grammar.	Supported type is “ application/x-jsgf ” as defined in Appendix D of the VoiceXML 1.0 specification.
<help>	Catches a help event.	Supported as documented in the VoiceXML 1.0 specification.
<ibmvoice>	Element added by IBM to allow application developers to control the text-to-speech engine’s synthesized voice on a per-prompt basis.	This element takes two attributes, as shown in Table 5 on page 50 : gender — valid values are male (the default) or female age — valid values are child , middle_adult (the default), or older_adult Note: This element is only implemented in the IBM WebSphere Voice Server SDK and the IBM WebSphere Voice Server with ViaVoice Technology.
<if>	Defines a conditional statement.	Supported as documented in the VoiceXML 1.0 specification.
<initial>	Prompts for form-wide information in a mixed-initiative form.	Supported as documented in the VoiceXML 1.0 specification.

Element	Description	Implementation Details
<link>	Specifies a transition to a new document or throws an event, when activated.	Supported as documented in the VoiceXML 1.0 specification.
<menu>	Specifies a dialog for selecting from of a list of choices.	Supported as documented in the VoiceXML 1.0 specification.
<meta>	Specifies meta data about the document.	The http-equiv attribute is supported as documented in the VoiceXML 1.0 specification. The VoiceXML browser ignores the name attribute and any content specified with it; you can use these attributes to identify and assign values to the properties of a document, as defined by the HTML 4.0 specification (http://www.w3.org/TR/REC-html40/) and the HTTP 1.1 specification (http://www.ietf.org/rfc/rfc2616.txt).
<noinput>	Catches a noinput event.	Supported as documented in the VoiceXML 1.0 specification.
<nomatch>	Catches a nomatch event.	Supported as documented in the VoiceXML 1.0 specification.
<object>	Specifies platform-specific objects.	The IBM WebSphere Voice Server SDK platform does not provide any objects. If the VoiceXML browser encounters this element, it will throw an error.unsupported.object error event.
<option>	Specifies a field option.	Supported as documented in the VoiceXML 1.0 specification.
<param>	Specifies a parameter in an object or subdialog.	Supported as documented in the VoiceXML 1.0 specification.
<prompt>	Plays TTS or recorded audio output.	Supported as documented in the VoiceXML 1.0 specification.
<property>	Controls various aspects of the behavior of the implementation platform.	The confidencelevel , fetchaudio , inputmodes , sensitivity , and speedvsaccuracy properties are not implemented in this release.

Element	Description	Implementation Details
<pros>	Controls the prosody of TTS output.	Supported as documented in the VoiceXML 1.0 specification.
<record>	Records spoken user input.	The VoiceXML browser records the audio as an 8KHz .au file. The beep , maxtime , and finalsilence attributes are not implemented in this release.
<reprompt>	Causes the form interpretation algorithm to queue and play a prompt when entering a form after an event.	Supported as documented in the VoiceXML 1.0 specification.
<return>	Returns from a subdialog.	Supported as documented in the VoiceXML 1.0 specification.
<sayas>	Controls pronunciation of words or phrases in TTS output.	Supported values for the class attribute are digits and literal .
<script>	Specifies a block of ECMAScript code.	Supported as documented in the VoiceXML 1.0 specification.
<subdialog>	Invokes a new dialog as a subdialog of the current one, in a new execution context.	The modal attribute mentioned in the VoiceXML 1.0 specification is not supported because it is not included in the VoiceXML 1.0 DTD.
<submit>	Submits a list of variables to the document server.	IBM has extended this element to support the HTML enctype value of “multipart/form-data” for the enctype attribute. To submit the results of a <record> element, you must use this new attribute value, as shown in Table 6 on page 50 .

Element	Description	Implementation Details
<code><throw></code>	Throws an event.	Supported as documented in the VoiceXML 1.0 specification.
<code><transfer></code>	Connects the telephone caller to a third party. Applicable only to a telephony deployment environment, not the desktop development environment.	Not implemented in this release.
<code><value></code>	Embeds a variable in a prompt.	The class , mode , and recsrc attributes are not implemented in this release.
<code><var></code>	Declares a variable.	Refer to Table 10 on page 66 for information on variable scope and Table 11 on page 67 for information on shadow variables.
<code><vxml></code>	Top-level container for all other VoiceXML elements in a document.	All VoiceXML documents must specify version="1.0" ; if the version attribute is missing, the VoiceXML browser will throw an error.badfetch event. The lang attribute is not implemented in this release. Instead, the VoiceXML browser uses the value of the vxml.locale Java system property specified when the browser was started.

Examples of IBM Extensions

Table 5. <ibmvoice> Example

```
<ibmvoice gender="female" age="child">
This text will be read using a female child synthesized voice.
</ibmvoice>
```

Table 6. <submit> Example

```
<submit next="http://voiceserver.somewhere.com/servlet/submitrecording"
method="post" enctype="multipart/form-data"/>
```

Dialog Structure

VoiceXML documents are composed primarily of top-level elements called dialogs; there are two types of dialogs defined in the language: **<form>** and **<menu>**.

Forms and Form Items

Forms allow the user to provide voice or DTMF input by responding to one or more **<field>** elements.

Fields

Each field can contain one or more **<prompt>** elements that guide the user to provide the desired input. You can use the **count** attribute to vary the prompt text based on the number of times that the prompt has been played.

Fields can also specify a **type** attribute or a **<grammar>** or **<dtmf>** element to define the valid input values for the field, and any **<catch>** elements necessary to process the events that might occur. Fields may also contain **<filled>** elements, which specify code to execute when a value is assigned to a field. You can reset one or more form items using the **<clear>** element.

Subdialogs

Another type of form item is the **<subdialog>** element, which creates a separate execution context to gather information and return it to the form. For more information, see [“Subdialogs” on page 54](#).

Blocks

If your form requires prompts or computation that do not involve user input (for example, welcome information), you can use the **<block>** element. This element is also a container for the **<submit>** element, which specifies the next URI to visit after the user has completed all the fields in the form. You can also jump to another form item in the current form, another dialog in the current document, or another document using the **<goto>** element.

Types of Forms

There are two types of form dialogs:

- *Machine-directed* forms — simple forms where each field or other form item is executed once and in a sequential order, as directed by the system.
- *Mixed initiative* forms — more robust forms in which the system or the user can direct the dialog. When coding mixed initiative forms, you can use form-level grammars (`<form scope="dialog">`) to allow the user to fill in multiple fields from a single utterance, or document-level grammars (`<form scope="document">`) to allow the form's grammars to be active in any dialog in the same VoiceXML document; if the document is the application root document, then the form's grammars are active in any dialog in any document within the application. You can use the `<initial>` element to prompt for form-wide information in a mixed initiative dialog, before the user is prompted on a field-by-field basis. For more information, see [“Mixed Initiative Application and Form-level Grammars” on page 79](#).

Menus

A menu is essentially a simplified form with a single field.

Menus present the user with a list of choices, and associate with each choice a URI identifying a VoiceXML page or element to visit if the user selects that choice. The grammar for a menu is constructed dynamically from the menu entries, which you specify using the `<choice>` element or the shortcut `<enumerate/>` construction; you can use the `<menu>` element's `<scope>` attribute to control the scope of the grammar.

Note: The `<enumerate>` element instructs the VoiceXML browser to speak the text of each menu `<choice>` element when presenting the list of available selections to the user. If you want more control over the exact wording of the prompts (such as the ability to add words between menu items or to hide active entries in your menu), simply leave off the `<enumerate>` tag.

Menus can accept voice and/or DTMF input. If desired, you can implicitly assign DTMF key sequences to menu choices based on their position in the list of choices by using the construct `<menu dtmf="true">`.

Flow Control

When the VoiceXML browser starts, it uses the URI you specify to request an initial VoiceXML document. Based on the interaction between the application and the user, the VoiceXML browser may jump to another dialog in the same VoiceXML document, or fetch and process a new VoiceXML document.

VoiceXML provides a number of ways managing flow control. For example:

```
<link event="help">
```

```
<link next="http://www.yourbank.example/locations.vxml/">
```

```
<goto nextitem="field1"/>
```

```
<submit next="/servlet/login"/>
```

```
<choice next="http://www.yourbank.example/locations.vxml">
```

```
<throw event="exit">
```

When a dialog does not specify a transition to another dialog, the VoiceXML browser exits and the session terminates.

Subdialogs

VoiceXML subdialogs are roughly the equivalent of function or method calls. A subdialog is an entire form that is executed, the result of which is used to provide one input field to another form. You can use subdialogs to provide a disambiguation or confirmation dialog (as described in [“Managing Errors” on page 163](#)), as well as to create reusable dialog components for data collection and other common tasks.

Executing a **<subdialog>** element temporarily suspends the execution context of the calling dialog and starts a new execution context, passing in the parameters specified by the **<param>** element. When the subdialog exits, it uses a **<return>** element to resume the calling dialog’s execution context. The calling dialog accesses the results of the subdialog through the variable defined by the **<subdialog>** element’s **name** attribute.

Comments

Information within a comment is ignored by the VoiceXML browser. Single line comments in VoiceXML documents use the following syntax:

```
<!--comment-->
```

Comments can also span multiple lines:

```
<!--start of multi-line comment  
more comments-->
```

Built-in Field Types and Grammars

The VoiceXML browser supports the complete set of built-in field types that are defined in the VoiceXML 1.0 specification. These built-in field types specify the built-in grammar to load (which determines valid spoken and DTMF input) and how the TTS engine will pronounce the value in a prompt, as documented in [Table 7 on page 55](#). Additional examples of the types of input you might specify when using the built-in field types are shown in [Table 36 on page 183](#).

For information on writing your own, application-specific grammars, see [“Grammars” on page 72](#).

Table 7. Built-in Types

Type	Implementation Details
boolean	<p>Users can say affirmative such as yes, okay, sure, and true or negative responses such as no, false, and negative. Users can also provide DTMF input: 1 is yes, and 2 is no.</p> <p>The return value sent is a boolean true or false. If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak either yes or no.</p>
currency	<p>Users can say U.S. currency values in dollars and cents from 0 to \$999,999,999,999.99 including common constructs such as “a buck fifty” or “nine ninety nine”.</p> <p>Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key.</p> <p>The return value sent is a string in the format <i>UUUddddddddd.cc</i>, where <i>UUU</i> is a currency indicator or null; currently, the only supported currency type is USD (for US dollars). If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak the currency value. For example, the TTS engine would speak “a buck fifty” as one dollar and fifty cents and “nine ninety nine” as nine ninety nine.</p>
date	<p>Users can say a date using months, days, and years, as well as the words yesterday, today, and tomorrow. Common constructs such as “March 3rd, 2000” or “the third of March 2000” are supported.</p> <p>Users can also provide DTMF input in the form <i>yyyymmdd</i>.</p> <p>Note: The date grammar does not perform leap year calculations; February 29th is accepted as a valid date regardless of the year. If desired, your application or servlet can perform the required calculations.</p> <p>The return value sent is a string in the format <i>yyyymmdd</i>, with the VoiceXML browser returning a ? in any positions omitted in the spoken input. If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak the date.</p>

Type	Implementation Details
digits	<p>Users can say numeric integer values as individual digits (0 through 9). For example, a user could say 123456 as “1 2 3 4 5 6”.</p> <p>Users can also provide DTMF input using the numbers 0 through 9, and must terminate DTMF entry using the # key.</p> <p>The return value sent is a string of one or more digits. If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak individual digits from 0 through 9. In the above example, the TTS engine would speak 123456 as 1 2 3 4 5 6.</p> <p>Note: Use this type instead of the number type if you require very high recognition accuracy for your numeric input.</p>
number	<p>Users can say natural numbers (that is, positive and negative integers and decimals) from 0 to 999,999,999,999 as well as the words point or dot (to indicate a decimal point) and minus or negative (to indicate a negative number). Numbers can be spoken individually or in groups. For example, a user could say 123456 as “one hundred twenty-three thousand four hundred and fifty-six” or as “one, twenty-three, four, fifty-six”.</p> <p>Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to indicate a decimal point), and must terminate DTMF entry using the # key.</p> <p>Note: Only positive numbers can be entered using DTMF.</p> <p>The return value sent is a string of one or more digits, 0 through 9, with a decimal point and a + or - sign as applicable. If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak the natural number. In the above example, the TTS engine would speak 123456 as one hundred twenty-three thousand four hundred fifty-six.</p> <p>Note: Use this type to provide flexibility when collecting account numbers; users can speak long numbers as natural numbers, single digits, or groups of digits. If you want to force the TTS engine to speak the number back as a string of digits, use the <code><sayas class="literal"></code> markup tag.</p>

Type	Implementation Details
phone	<p>Users can say a telephone number, including the optional word extension.</p> <p>Users can also provide DTMF input using the numbers 0 through 9 and optionally the * key (to represent the word “extension”), and must terminate DTMF entry using the # key.</p> <p>The return value sent is a string of digits without hyphens, and includes an x if an extension was specified. If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak the telephone number, including any extension.</p> <p>Note: For tips on minimizing recognition errors that are due to user pauses during input, refer to “Using the Built-in Phone Grammar” on page 161.</p>
time	<p>Users can say a time using hours and minutes in either 12 or 24 hour format, as well as the word now.</p> <p>Users can also provide DTMF input using the numbers 0 through 9.</p> <p>The return value sent is a string in the format <i>hhmmx</i>, where <i>x</i> is a for AM, p for PM, h for 24 hour format, or ? if unspecified or ambiguous; for DTMF input, the return value will always be h or ?, since there is no mechanism for specifying AM or PM. If the field name is subsequently used in a value attribute within a prompt, the TTS engine will speak the time.</p>

Recorded Audio

VoiceXML supports the use of recorded audio files as output. The VoiceXML browser plays an audio file when the corresponding URI (`<audio src=file>`) is encountered in a VoiceXML document.

Using Prerecorded Audio Files

In the IBM WebSphere Voice Server SDK and the IBM WebSphere Voice Server with ViaVoice Technology, prerecorded audio files must be in 8KHz **.au** audio format; the IBM WebSphere Voice Server for DirectTalk supports this format plus 8KHz **.wav** files.

Recording Spoken User Input

You can use the VoiceXML `<record>` element to capture spoken input; the recording ends when the user presses any DTMF key. To allow a spoken command to terminate the recording, you can specify a `<grammar>` element within the `<record>` element; users must pause briefly before and after speaking an utterance from this grammar; all other grammars are turned off while recording is active.

Playing and Storing Recorded User Input

You can play the recorded input back to the user immediately (using a `<value>` element), or submit it to the server (using `<submit next="URL" method="post" enctype="multipart/form-data"/>`) to be saved as an audio file.

Document Fetching and Caching

The VoiceXML browser uses caching to improve performance when fetching documents and other resources (such as audio files, grammars, and scripts); see [“Fetching and Caching Resources for Improved Performance” on page 177](#).

Configuring Caching

By default, caching is enabled and the files are stored on the local file system in the directory specified by the `vxml.cache.dir` Java system property. You can turn caching off by specifying the Java system property `-Dvxml.cache=false` when you start the VoiceXML browser. These Java system properties are described in [Table 16 on page 88](#).

In a deployment (telephony) environment, the VoiceXML browser supports persistent caching for grammars, documents, and audio clips; it stores these files locally to permit them to be shared between multiple browser instances. Refer to the documentation for the applicable deployment platform for implementation details.

Controlling Fetch and Cache Behavior

Table 8 presents the attributes for controlling document fetching and caching; these are available with various elements, including `<audio>`, `<choice>`, `<goto>`, `<grammar>`, `<prompt>`, `<property>`, `<script>`, `<subdialog>`, and `<submit>`.

Table 8. Attributes for Document Fetching and Caching

Attribute	Description	Implementation Details
caching	Specifies whether to retrieve the content from cache or from the server.	Valid values are fast (the default) and safe . The fast caching policy is similar to that used by HTML browsers: if a requested document is unexpired in the cache or expired but not modified, the cached copy is used; if the requested document is expired and modified or is not in the cache, the document is fetched from the server. You can specify the safe caching policy to force a query to fetch the document from the server.
fetchaudio	Specifies the URI of an audio file to play while fetching a document.	Not implemented in this release.
fetchhint	Specifies when to retrieve and process the content.	Valid values are prefetch (fetch when the page is loaded) and safe (fetch when the content is needed). In this release, stream (process as the content arrives) is not implemented; if you specify stream , safe is used instead.
fetchtimeout	Specifies how long to wait for the content before throwing an error.badfetch event.	Supported as documented in the VoiceXML 1.0 specification.

Preventing Caching

You can use the **<meta>** element to prevent caching of a VoiceXML document by specifying **http-equiv="expires"** and **content="0"**. To specify when the VoiceXML browser should fetch a fresh copy of the document, use the **content** attribute to specify the date and time. For example:

```
http-equiv="expires" content="Tue, 20 Aug 2000 14:00:00 GMT"
```

Events

The VoiceXML browser throws an event when it encounters a **<throw>** element or when certain specified conditions occur; these may be normal error conditions (such as a recognition error) or abnormal error conditions (such as an invalid page reference). Events are caught by **<catch>** elements that can be specified within other VoiceXML elements in which the event can occur, or inherited from higher-level elements.

Predefined Events

The VoiceXML browser supports a subset of the predefined events documented in the VoiceXML 1.0 specification, as described in [Table 9 on page 61](#):

Table 9. Predefined Events and Event Handlers

Event	Description	Implementation Details
cancel	The VoiceXML browser throws this event when the user says a valid word or phrase from the Quiet/Cancel grammar. See “Built-in Commands” on page 84 .	The default event handler stops the playback of the current audio prompt.
error.badfetch	The VoiceXML browser throws this event when it detects a problem (such as an unknown audio type, an invalid URI, or the expiration of a fetchtimeout) that prevents it from jumping to the next URI.	The default event handler plays the message, “Sorry, must exit due to processing error” and then exits.
error.noauthorization	The VoiceXML browser throws this event when the user attempts to perform an action for which he/she is not authorized.	The default event handler plays the message, “Sorry, must exit due to processing error” and then exits.
error.semantic	The VoiceXML browser throws this event when it detects an error within the content of a VoiceXML document (such as a reference to a non-existent application root document or an ill-formed ECMAScript expression) or when a critical error (such as an error communicating with the speech recognition engine) occurs during operation.	The default event handler plays the message, “Sorry, must exit due to processing error” and then exits.

Event	Description	Implementation Details
error.unsupported.element	The VoiceXML browser throws this event when it encounters a reference to an unsupported element.	For example, if a document contains a <transcribe> element, the VoiceXML browser will throw an error.unsupported.transcribe event. The default event handler plays the message, “Sorry, must exit due to processing error” and then exits.
error.unsupported.format	The VoiceXML browser throws this event when it encounters a reference to an unsupported resource format.	Not implemented in this release.
exit	The VoiceXML browser throws this event when it processes a <throw event=’exit’> .	The VoiceXML browser performs some cleanup and then exits.
help	The VoiceXML browser throws this event when the user says a valid word or phrase from the Help grammar. See “Built-in Commands” on page 84 .	The default event handler plays the message, “Sorry, no help available” and then reprompts the user; for guidelines on choosing a scheme for your own help messages, see “Choosing Help Mode or Self-Revealing Help” on page 131 .
noinput	The VoiceXML browser throws this event when the timeout interval (as defined by a timeout attribute on the current prompt, the timeout property, or the vxml.timeout Java system property) is exceeded.	The default event handler reprompts the user; if you adhere to the guidelines for self-revealing help, this event can use the same messages as the help event. See “Implementing Self-Revealing Help” on page 133 for details.

Event	Description	Implementation Details
nomatch	The VoiceXML browser throws this event when the user says something that is not in any of the active grammars.	<p>The default event handler plays the message, “Sorry, I didn’t understand” and then reprompts the user; if you adhere to the guidelines for self-revealing help, this event can use the same messages as the help event. See “Implementing Self-Revealing Help” on page 133 for details.</p> <p>If the user pauses after uttering a partial response, and the silence period exceeds the duration specified by the incompletetimeout property, the VoiceXML browser’s response depends on whether the user’s partial utterance matches a valid utterance from any active grammar:</p> <ul style="list-style-type: none"> • If the partial user utterance is itself a valid utterance, the VoiceXML browser returns a result. For example, if the built-in phone grammar is active and the user pauses after uttering 7 digits of a 10-digit telephone number, the VoiceXML browser will return the 7-digit number. • If the partial user utterance is not itself a valid utterance (for example, if the user pauses in the middle of a word), the VoiceXML browser throws a nomatch event.

Event	Description	Implementation Details
telephone.disconnect.hangup	<p>The VoiceXML browser throws this event when the user hangs up the telephone or when a <disconnect> element is encountered.</p> <p>This event is applicable only to the telephony deployment environment, not the desktop development environment.</p>	Not implemented in this release.
telephone.disconnect.transfer	<p>The VoiceXML browser throws this event when the user has been unconditionally transferred to another line (via a <transfer> element) and will not return.</p> <p>This event is applicable only to the telephony deployment environment, not the desktop development environment.</p>	Not implemented in this release.

If desired, you can override the predefined event handlers by specifying your own **<catch>** element to handle the events. For example:

```
<catch event="error.badfetch">
  Caught bad fetch.
  <goto nextitem="field1"/>
</catch>
```

Application-specific Events

In addition, you can define application-specific events and event handlers. Error types should be of the form: **error.packageName.errorType**, where *packageName* follows the Java convention of starting with the company's reversed Internet domain name (for example, **com.ibm.mypackage**).

Recurring Events

If desired, you can use the **<catch>** element's **count** attribute to vary the message based on the number of times that an event has occurred.

Note: This mechanism is used for self-revealing help and tapered prompts.

For example:

```
<prompt> Name and location?</prompt>
<noinput count="1">
    <prompt> Please state the name and location of the employee you wish
to call.</prompt>
</noinput>
<noinput count="2">
    <prompt>For example, to call Joe Smith in Kansas City, say, "Joe
Smith in Kansas City".</prompt>
</noinput>
```

Variables and Expressions

You can specify an executable script within a **<script>** element or in an external file.

Using ECMAScript

The scripting language of VoiceXML is ECMAScript, an industry-standard programming language for performing computations in Web applications. For information on the ECMAScript, refer to the *ECMAScript Language Specification*, available at: <http://www.ecma.ch/ecma1/stand/ECMA-262.htm>.

ECMAScript is also used for Action Tags in mixed-initiative forms; see “[Mixed Initiative Application and Form-level Grammars](#)” on page 79 for details.

Declaring Variables

You can declare variables using the `<var>` element or the **name** attribute of various form items. [Table 10](#) lists the possible variable scopes:

Table 10. Variable Scope

Scope	Description	Implementation Details
anonymous	Used by <code><block></code> , <code><filled></code> , and <code><catch></code> elements for variables declared in those elements.	Supported as documented in the VoiceXML 1.0 specification.
application	Variables are visible to the application root document and any other loaded application documents that use that root document.	Supported as documented in the VoiceXML 1.0 specification.
dialog	Variables are visible only within the current <code><form></code> or <code><menu></code> element.	Supported as documented in the VoiceXML 1.0 specification.
document	Variables are visible from any dialog within the current document.	Supported as documented in the VoiceXML 1.0 specification.
session	Variables that are declared and set by the interpreter context.	In this release, no session variables are declared. This means that ANI (Automatic Number Identification), DNIS (Dialed Number Identification Service), II Digit (Information Indicator Digit), and User to User Information functionality are not available.

Assigning and Referencing Variables

After declaring a variable, you can assign it a value using the `<assign>` element, and you can reference variables in `cond` and `expr` attributes of various elements.

Note: Because VoiceXML is an XML-based application, you must use escape sequences to specify relational operators (such as greater than, less than, etc.). For example, instead of “<” you must use “<” to represent “less than”.

Using Shadow Variables

The result of executing a field item is stored in the field’s `name` attribute; however, there may be occasions in which you need other types of information about the execution, such as the string of words recognized by the speech recognition engine. Shadow variables provide this type of information; [Table 11](#) documents the VoiceXML browser’s support for shadow variables.

Table 11. Shadow Variables

Element	Shadow Variable	Description	Implementation Details
<code><field></code>	<code>x\$.confidence</code>	The recognition confidence level from the speech recognition engine.	The VoiceXML browser always returns a value of 1.0.
	<code>x\$.inputmode</code>	The input mode for a given utterance: voice or dtmf.	Supported as documented in the VoiceXML 1.0 specification.
	<code>x\$.utterance</code>	A string that represents the actual words that were recognized by the speech recognition engine.	The actual value of the shadow variable is dependent on how the grammar was written.
<code><record></code>	<code>x\$.duration</code>	The duration of the recording.	Not implemented in this release.
	<code>x\$.size</code>	The size of the recording, in bytes.	Not implemented in this release.
	<code>x\$.termchar</code>	The DTMF key used to terminate the recording.	Not implemented in this release.

Element	Shadow Variable	Description	Implementation Details
<transcribe>	x\$.confidence	The confidence level of the transcription.	Not implemented in this release.
	x\$.termchar	The DTMF key used to terminate the transcription.	Not implemented in this release.
	x\$.utterance	A string that represents the actual words that were recognized by the speech recognition engine.	Not implemented in this release.
<transfer>	x\$.duration	The duration of a successful call. This event is applicable only to the telephony deployment environment, not the desktop development environment.	Not implemented in this release.

A Simple VoiceXML Example

Table 12 illustrates the basic dialog capabilities of VoiceXML, using a menu and a form:

Table 12. VoiceXML Example Using Menu and Form

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

  <menu>
    <prompt>Welcome to Your Bank Online speech demo.</prompt>
    <prompt>Please choose<enumerate/></prompt>
    <choice next="http://www.yourbank.example/locations.vxml">
      Branch Locations
    </choice>
    <choice next="http://www.yourbank.example/interest.vxml">
      Interest Rates
    </choice>
    <choice next="#login">
      Account Information
    </choice>
  </menu>

  <form id="login">
    <field name="account_number" type="number">
      <prompt>
        What is your account number?
      </prompt>
    </field>
```

```
<field name="pin_code">
  <dtmf src="builtin:dtmf/digits"/>
  <prompt>
    Use your telephone keypad to enter your PIN code,
    followed by the # key.
  </prompt>
</field>
<filled>
  <submit next="/servlet/login"/>
</filled>
</form>
</vxml>
```

Each menu or form field in a VoiceXML application must define a set of acceptable user responses. The menu uses **<choice>** elements to do this, while the **<field>** elements in the above example use the **type** attribute and the **<dtmf>** element to specify a built-in grammar (here, “number” and “digits”). You can also create your own, application-specific grammars (see [“Grammars” on page 72](#)).

The resulting dialog can proceed in numerous ways, depending on the user’s responses. Two possible interactions are described next.

Static Content

One sample interaction between the system and the user might sound like this:

```
System:  Welcome to Your Bank Online speech demo.
         Please choose: Branch Locations, Interest Rates,
         Account Information.
User:    Branch Locations.
```

At this point, the system retrieves and interprets the VoiceXML document located at **http://www.yourbank.example/locations.vxml**; this new document would specify the next segment of the interaction between the system and the user, namely a dialog to locate the nearest bank branch.

This interaction visits the **<menu>** element, but not the **<form>** element, from the sample VoiceXML document. It illustrates a level of capability similar to that provided for visual applications by a set of static HTML hyperlinks. However, static linking of information is merely the basic function of the World Wide Web; the real power of the Web is in dynamic distributed services, which begins with forms.

Dynamic Content

This interaction visits both the **<menu>** and the **<form>** from the above VoiceXML example:

```
System:  Welcome to Your Bank Online speech demo.
         Please choose: Branch Locations, Interest Rates,
         Account Information.
User:    Account Information.
System:  What is your account number?
User:    123
System:  Use your telephone keypad to enter your PIN code,
         followed by the # key.
User:    (presses telephone keys corresponding to PIN code,
         and terminates data entry using the # key)
```

At this point, the system has collected the two fields needed to complete the login, so it executes the **<filled>** element, which contains a **<submit>** element that causes the collected information to be submitted to a remote server for processing. For example, the servlet might access a database to verify the user account number and PIN code, after which a new VoiceXML document might present dynamically-generated data related to this particular user's account.

Grammars

A grammar is an enumeration, in compact form, of the set of *utterances* (words and phrases) that constitute the acceptable user response to a given prompt. The VoiceXML browser requires all speech and DTMF grammars to be specified using the *Java Speech Grammar Format (JSGF)*, as described in Appendix D of the VoiceXML 1.0 specification.

Additionally, the VoiceXML browser supports grammars that combine JSGF syntactic tagging together with ECMAScript Action Tags that specify semantic actions. This mechanism was proposed jointly by IBM and Sun Microsystems and is outside of the scope of the VoiceXML language; for more information, refer to the document *ECMAScript Action Tags for JSGF*, located in the `%IBMVS%\docs\specs` directory (where `%IBMVS%` is an environment variable that contains the pathname of the installation directory).

When you write your application, you can use the built-in grammars and/or create one or more of your own; in either case, you must decide when each grammar should be active. The speech recognition engine uses only the *active grammars* to define what it listens for in the incoming speech.

This section discusses the following topics:

- "Grammar Syntax" — See [page 72](#).
- "Static Grammars" — See [page 76](#).
- "Dynamic Grammars" — See [page 77](#).
- "Grammar Scope" — See [page 78](#).
- "Hierarchy of Active Grammars" — See [page 78](#).
- "Mixed Initiative Application and Form-level Grammars" — See [page 79](#).

Grammar Syntax

A complete discussion of grammar syntax is beyond the scope of this document; please refer to the JSGF documentation at <http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/index.html>. The information provided in this section is merely to acquaint you with the basics of writing JSGF grammars.

Grammar Header

A JSGF grammar file consists of a grammar header and a grammar body. The grammar header declares the version of JSGF and the grammar name.

The form of the grammar name can be either a “*simple grammar name*” (that is, *grammarName*) or a “*full grammar name*” (that is, *packageName.simpleGrammarName*).

A simple grammar header might be:

```
#JSGF V1.0;  
grammar citystate;
```

Grammar Body

The grammar body consists of one or more *rules* that define the valid set of utterances. The syntax for grammar rules is:

```
public <rulename> = options;
```

where:

public

is an optional declaration indicating that the rule can be used as an active rule by the speech recognition engine.

rulename

is a unique name identifying the grammar rule.

options

can be any combination of: text that the user can speak, another rule, and delimiters such as:

- | to separate alternatives
- [] to enclose optional words, phrases, or rules
- () to group words, phrases, or rules
- { } to associate an arbitrary tag with a word or phrase
- * to indicate that the previous item may occur zero or more times
- + to indicate that the previous item may occur one or more times

Comments in Grammars

You can specify comments in the header or body of a grammar using the standard Java format:

```
/* ignore this text */
```

or

```
// ignore from here to the end of the line
```

External and Inline Grammars

You can specify grammars in an external file. For example:

```
#JSGF V1.0;  
grammar employees;  
public <name>= Jonathan | Jon {Jonathan} | Larry | Susan | Melissa;
```

To access an external grammar, you reference either the URI of the grammar file (to use all public rules) or the URI of the grammar file plus a terminating *#rule* (to use only the specified public rule).

You can also specify grammar rules inline, by using VoiceXML's **<grammar>** element. For example:

```
<grammar>  
    Jonathan | Jon {Jonathan} | Larry | Susan | Melissa  
</grammar>
```

Note: An inline grammar that contains XML reserved terms or non-terminals must be enclosed within a PCDATA block.

DTMF Grammars

The VoiceXML browser also uses JSGF as the format for DTMF grammars. For example, the following example defines an inline DTMF grammar that allows the user to make a selection by pressing the numbers 1 through 4, the asterisk, or the pound sign on the DTMF Simulator (during desktop testing) or on a telephone keypad (when the application is deployed):

```
<dtmf type="application/x-jsgf">  
  1 | 2 | 3 | 4 | "*" | "#"  
</dtmf>
```

Static Grammars

The following example illustrates the use of grammars in a hypothetical Web-based voice application for a restaurant:

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

  <form>
    <field name="drink">
      <prompt>What would you like to drink?</prompt>
      <grammar>
        coffee | tea | orange juice | milk | nothing
      </grammar>
    </field>
    <field name="sandwich">
      <prompt>What type of sandwich would you like?</prompt>
      <grammar src="sandwich.gram"/>
    </field>
    <filled>
      <submit next="/servlet/order"/>
    </filled>
  </form>
</vxml>
```

In this example, the first grammar (for drinks) consists of a single rule, specified inline. In contrast, the second grammar (for sandwiches) is contained in an external grammar file, shown below:

```
#JSGF V1.0;
grammar sandwich;
<ingredient>= ham | turkey | swiss [cheese];
<bread>= rye | white | [whole] wheat;
public <sandwich>=<ingredient> ([and] <ingredient>)* [on <bread>];
```

Here, the ingredient and bread rules are private, meaning that they can only be accessed by other rules in this grammar file. The sandwich rule is public, meaning that it can be accessed by specifying this grammar file or by referencing the fully-qualified name of the sandwich rule.

The public rule in this grammar allows the user to build a sandwich by specifying, in order:

- <ingredient> one item from the ingredient rule
- ([and] <ingredient>)* zero or more (denoted by the *) additional ingredients, optionally (denoted by the [...]) separated by the word “and”
- [on <bread>] optionally, the word “on” followed by a type of bread from the bread rule

So, for example, a typical dialog might sound like this:

```
System:  What would you like to drink?
User:    Coffee
System:  What type of sandwich would you like?
User:    Turkey and cheese on rye.
```

At this point, the system has collected the two fields needed to complete the order, so it executes the <filled> element, which contains a <submit> that causes the collected information to be submitted to a remote server for processing.

Dynamic Grammars

In the restaurant example, the contents of the inline and the external grammars were static. However, it is possible to create grammars dynamically, such as by using information from a back-end database located on an enterprise server. You can use the same types of server-side logic to build dynamic VoiceXML as you would use to build dynamic HTML: CGI scripts, Java Beans, servlets, ASPs, JSPs, etc.

For more information, refer to the Grammar Builder application located in the `%IBMVS%\samples` directory (where `%IBMVS%` is an environment variable that contains the pathname of the installation directory).

Grammar Scope

Form grammars can specify one of the scopes shown in [Table 13](#):

Table 13. Form Grammar Scope

Scope	Description	Implementation Details
dialog	The grammar is accessible only within the current <form> or <menu> element.	Supported as documented in the VoiceXML 1.0 specification.
document	The grammar is accessible from any dialog within the current document; if the document is the application root document, the grammar is accessible to all loaded application documents that use that root document.	Supported as documented in the VoiceXML 1.0 specification. You can improve performance by using a combination of fetching and caching to prime the cache with all of the referenced grammar files before your application starts, as described in “Fetching and Caching Resources for Improved Performance” on page 177.

Field, link, and menu choice grammars cannot specify a scope:

- Field grammars are accessible only within the specified field.
- Link grammars are accessible only within the element containing the link.
- Grammars in menu choices are accessible only within the specified menu choice.

Hierarchy of Active Grammars

When the VoiceXML browser receives a recognized word or phrase from the speech recognition engine, the browser searches the active grammars, *in the following order*, looking for a match:

1. Grammars for the current field, including grammars contained in links in that field.
2. Grammars for the current form, including grammars contained in links in that form.
3. Grammars contained in links in the current document, and grammars for menus and other forms in the current document that have **document** scope.
4. Grammars contained in links in the application root document, and grammars for menus and other forms in the application root document that have **document** scope.
5. Grammars for the VoiceXML browser itself (that is, built-in commands).

Disabling Active Grammars

You can temporarily disable active grammars, *including the VoiceXML browser's grammars for built-in commands*, by using the **modal** attribute on various form items. When **modal** is set to **true**, all grammars are temporarily disabled except the grammar for the current form item.

Resolving Ambiguities

Ambiguities can arise when a user's utterance matches more than one of the following:

- Phrases in one or more active grammars
- Items in a menu or form
- Links within a single document

You should exercise care to avoid using the same word or phrase in multiple grammars that could be active concurrently; the VoiceXML browser resolves any ambiguities by using the first matching value.

Understanding how disambiguation works is especially important when multiple phrases in concurrently active grammars contain the same key words. See [“Matching Partial Phrases” on page 157](#) for details.

Mixed Initiative Application and Form-level Grammars

In a *machine-directed application*, the computer controls all interactions by sequentially executing each form item a single time.

However, VoiceXML also supports *mixed-initiative applications* in which either the system or the user can direct the conversation. One or more grammars in a mixed initiative application may be active outside the scope of its own dialog; to achieve this, you can use the `<link>` element and code them as either form-level grammars (**scope="document"**) or document-level grammars (**scope="document"**) defined in the application root document. If the user utterance matches an active grammar outside of the current dialog, the application jumps to the dialog specified by the `<link>`.

When you code a mixed initiative application, you may also use one or more `<initial>` elements to prompt for form-wide information, before the user is prompted on a field-by-field basis.

Form-level grammars allow a greater flexibility and more natural responses than field-level grammars because the user can fill in the fields in the form in any order and can fill more than one field as a result of a single utterance. For example:

```
#JSGF V1.0;
grammar citystate;
public <cityandstate> =
    <city> {this.city=$} [<state> {this.state=$}] [please] |
    <state> {this.state=$} [<city> {this.state=$}] [please] |;
<city> = Los Angeles | San Francisco | Yorktown Heights;
<state> = California | New York;
```

The VoiceXML browser is the component of the IBM WebSphere Voice Server SDK that fetches and processes VoiceXML documents and manages the dialog between the application and the user. Its relationship to the overall system architecture is described in [Chapter 1, “VoiceXML Browser” on page 39](#).

This chapter discusses the following topics:

- [“VoiceXML Browser Features”](#) — See [page 81](#).
- [“The vsaudio and vstext Batch Files”](#) — See [page 87](#).
- [“Starting the VoiceXML Browser”](#) — See [page 91](#).
- [“Using the Trace Mechanism”](#) — See [page 93](#).
- [“Stopping the VoiceXML Browser”](#) — See [page 97](#).

VoiceXML Browser Features

Key features of the VoiceXML browser include:

- [“Say What You Hear”](#) — See [page 82](#).
- [“Barge-in”](#) — See [page 83](#).
- [“Built-in Commands”](#) — See [page 84](#).
- [“Support for HTTP”](#) — [page 85](#).

“Say What You Hear”

The VoiceXML browser supports a “Say What You Hear” algorithm for selecting items from a menu or option list. The VoiceXML browser reads the prompt text, followed by the text of the choices or options; users can say *all or part of the phrase* from the text of a **<choice>** or **<option>** element to select from:

- The items already read.
- Any other items that will be read (a feature for expert users).
- Any previously read items in other currently active grammars, depending on the scope of those grammars.

For example, if the VoiceXML application menu presents the following choices:

System:	Say one of the following:
	Activate my credit card
	Change my address
	Show my credit limit

[Table 14 on page 83](#) lists some examples of what the user might say, and which menu item would be selected as a result:

Table 14. “Say What You Hear” Example

User Utterance	Menu Item Selected
Activate my credit card	Activate my credit card
Activate credit card	Activate my credit card
Activate card	Activate my credit card
Activate	Activate my credit card
Credit card	Activate my credit card
Change my address	Change my address
Change address	Change my address
Change	Change my address
Address	Change my address
Show my credit limit	Show my credit limit
Show credit limit	Show my credit limit
Show credit	Show my credit limit
Show limit	Show my credit limit
Credit limit	Show my credit limit
Credit	Activate my credit card Note: When the user utterance is a partial phrase that matches multiple items (“Activate my credit card” and “Show my credit limit” in this example), the VoiceXML browser selects the first conforming item. See “Matching Partial Phrases” on page 157 .

Barge-in

An important feature of the VoiceXML browser is that, with the audio input in full-duplex mode, a user can interrupt the system’s TTS or prerecorded audio output; this feature is referred to as “*barge-in*”.

For example, a user might interrupt the prompt by saying, “Quiet”, which would cause the VoiceXML browser to stop playing the current output and wait for further user input; to repeat the interrupted output, the user could say “Repeat that.” The user could also interrupt the prompt with a DTMF keypress or by providing spoken input, such as selecting a menu item.

When you start the VoiceXML browser, you can use the **vxml.bargein** Java system property (see [Table 16 on page 88](#)) to specify the barge-in mechanism for full-duplex applications; half-duplex applications permit the user to barge-in with a DTMF keypress, regardless of the value of the **vxml.bargein** property. You can turn barge-in off for an individual prompt by specifying `<prompt bargein="false">`.

Note: In a full-duplex telephony environment, barge-in requires good echo cancellation. If you encounter a lot of errors in which the speech recognition engine returned a result when the user did not actually speak, you might try switching to recognition barge-in detection, turning off barge-in entirely, or switching to half-duplex.

For more information on the tradeoffs involved with using barge-in and the types of barge-in detection available, see [“Choosing Full-Duplex \(Barge-In\) or Half-Duplex Implementation” on page 109](#).

Built-in Commands

The VoiceXML browser implements a number of built-in commands, similar in spirit to the built-in functions presented on menus and toolbars in visual browsers. [Table 15 on page 85](#) shows these built-in commands, using grammar-like syntax to indicate valid user utterances:

Table 15. Built-in VoiceXML Browser Commands

Grammar name	Valid User Utterances	VoiceXML Browser Response
Quiet/Cancel	[[please] be] quiet quiet please [please] cancel [this that] cancel [this that] please	In a full-duplex system, stops the current spoken output and waits for further instructions from user.
Repeat	[please] repeat [that] repeat [that] [please]	Repeats the most recently visited element (<menu>, <form>, or <body>).
Help	[please] help help please	Plays the help event handler message. See Table 9 on page 61 .

For guidelines on using these commands, see “[Adopting a Consistent Set of Always-Active Navigation Commands](#)” on page 125.

Support for HTTP

Support for HTTP is provided by the Java platform.

Dynamically Generating VoiceXML Documents

VoiceXML documents can be static files, or dynamically generated by a program in response to user input, typically from a form. The same types of server-side logic (such as CGI programs, Java Beans, ASPs, JSPs, and Java servlets) used to dynamically generate HTML for visual applications can be used to dynamically generate VoiceXML for voice applications. The high-level process looks like this:

1. The application collects user input and uses a <submit> element to submit the data to the server.
2. When the <submit> is processed, the VoiceXML browser formats an HTTP request that is sent to the server program.

The first line of the request message contains a method indicating the action to perform (GET or POST), the URL, and a list of arguments (that is, variable names and the data from the form); for the <record> element, only the POST method is supported.

The request also contains an **accept** header of “**application/x-vxml**” (preferred form), “**application/vxml**”, “**text/x-vxml**”, or “**text/vxml**”, which tells the server-side logic that the request is coming from the VoiceXML browser, as opposed to a visual browser.

Note: You may need to configure your Web server to recognize that *.vxml files are of this type; on some servers, failure to do so may generate a “file not found” exception in the browser.

The VoiceXML browser also supports the HTTP “expires” header, which specifies the date and time after which a VoiceXML document is considered stale. You can set this header using the “**http-expires**” attribute of the VoiceXML `<meta>` element.

3. The server-side logic processes the request and returns the newly-created VoiceXML document to the VoiceXML browser. The MIME type of the new document must be **application/x-vxml**, **application/vxml**, **text/x-vxml**, or **text/vxml** for a VoiceXML file, or **application/x-jsgf** for a grammar file.

Session Tracking and Cookie Support

Cookies enable a browser to save information (such as user preferences and data that the user submits while browsing a Web application) across multiple HTTP connections. You can use cookies to track the state of your VoiceXML application by sharing information between the VoiceXML browser and server-side logic running on a Web application server. Cookies persist only for the duration of a VoiceXML browser session, and can only be sent back to the originating server.

To use cookies, the server-side logic must:

1. Create a simple cookie of the form “*name=value*”.
2. If desired, check the value of the user-agent field contained in the HTTP Servlet request. For the VoiceXML browser, the user-agent will be **IBM VoiceXML** *release-level*.
3. Set the content type (that is, the MIME type) of the response to “**application/x-vxml**” (or **application/vxml**, **text/x-vxml**, **text/vxml**, or **application/x-jsgf**).
4. Add the cookie to the response.
5. Send the output to the VoiceXML browser.

The VoiceXML browser receives the VoiceXML document from the server and extracts the cookie from the HTTP header. When the VoiceXML browser makes a subsequent call to the same server, it passes the cookie back in the HTTP header.

For more information on cookies, refer to the HTTP State Management Mechanism (Cookie specification), published by W3C Network Working Group. Document available at <http://www.w3.org/Protocols/rfc2109/rfc2109>.

The vsaudio and vstext Batch Files

The **vsaudio.bat** and **vstext.bat** batch files define Java system properties for the VoiceXML browser executable in the IBM WebSphere Voice Server SDK.

When you use **vsaudio.bat** to run your application, the VoiceXML browser generates spoken output (using text-to-speech and/or recorded audio) and accepts spoken and simulated DTMF input (see [“Interactions with the DTMF Simulator” on page 39](#)).

When you use **vstext.bat** to run your application, the VoiceXML browser writes prompts and other output as text in the Java console window, and accepts input from your keyboard or the DTMF Simulator. If desired, you can provide input from a file by specifying a *<inputfile* parameter, as described in [“Command Line Interface” on page 92](#). This mode is useful for automated testing purposes, or if your desktop workstation does not have a microphone.

Configurable Properties

You can edit the batch files to specify the configurable properties shown in [Table 16 on page 88](#); each property must be in the form **-Doption=value**.

Table 16. Configurable Properties for Starting the VoiceXML Browser

Configurable Property	Description
vxml.bargein	<p>Specifies the barge-in detection method used by the VoiceXML browser for full-duplex applications. Valid values are recognition (the default) and energy.</p> <p>When recognition barge-in detection is specified, audio output stops only when the user finishes speaking a complete word or phrase from an active grammar. In energy barge-in detection, audio output stops as soon as the speech recognition engine detects sound.</p> <p>Note: If your application will be deployed using the IBM WebSphere Voice Server for DirectTalk, you should set this property to energy. For an introduction to barge-in, see “Barge-in” on page 83; for a discussion of the pros and cons of each barge-in detection method, see “Choosing Full-Duplex (Barge-In) or Half-Duplex Implementation” on page 109.</p>
vxml.cache	Specifies whether the VoiceXML browser caches resources. Valid values are true (the default) and false .
vxml.cache.dir	Specifies the pathname where the VoiceXML browser stores cached resources; this property is ignored if vxml.cache=false . The default value is .\cache .
vxml.cache.size	Specifies the target size of the cache (in MB) that remains after the VoiceXML browser exits. The default value is 1 .

Configurable Property	Description
vxml.duplex	<p>Specifies full (the default) or half duplex implementation.</p> <p>Note: You must use a full-duplex sound card even if you set this value to half.</p> <p>When full-duplex is specified, the user and the computer can speak at the same time; when half-duplex is specified, the user should not speak while the computer is speaking because the speech recognition engine does not receive audio while the computer is speaking.</p> <p>For more information, see “Choosing Full-Duplex (Barge-In) or Half-Duplex Implementation” on page 109.</p> <p>Note: If your application will be deployed using the IBM WebSphere Voice Server for DirectTalk, you should only set this property to half if you intend to disable barge-in for your entire application in the deployed environment.</p>
vxml.gui	<p>Specifies whether the VoiceXML browser should start the DTMF Simulator. Valid values are true (the default) and false. See “Interactions with the DTMF Simulator” on page 39 for information on using the DTMF Simulator.</p> <p>Note: This property applies to the SDK only; there is no equivalent parameter in either the IBM WebSphere Voice Server for DirectTalk or the IBM WebSphere Voice Server with ViaVoice Technology.</p>

Configurable Property	Description
vxml.site	<p>Specifies the URI of a VoiceXML site customization file that is read when the VoiceXML browser starts and remains loaded for the duration of the VoiceXML browser session.</p> <p>For a “<i>voice portal</i>” (that is, a site from which users can access various VoiceXML applications) deployed using the IBM WebSphere Voice Server with ViaVoice Technology, you can use this file to establish a Main Menu that is active from within any of the applications.</p> <p>Note: There is no equivalent parameter in the IBM WebSphere Voice Server for DirectTalk.</p> <p>You can also use it to create global <code><menu></code> and <code><link></code> elements for your site, to augment the set of built-in VoiceXML browser commands.</p> <p>Note: Do not define variables in this file.</p>
vxml.timeout	<p>Specifies the default timeout value before the VoiceXML browser throws a noinput event, in the form <i>numberms</i> (milliseconds) or <i>numbers</i> (seconds). The default is 7s; you may want to increase this value if a significant percentage of your expected users are non-native speakers, older adults, or other special populations that might require a bit more time to respond. See “Adopting a Consistent ‘Sound and Feel’” on page 136 for suggestions.</p> <p>Note: There is no equivalent parameter in the IBM WebSphere Voice Server for DirectTalk; however, you can achieve a similar effect by specifying a value for the timeout property in your application root document.</p>

Specifying Properties During Application Development

For example, to use energy-based barge-in detection, you would edit the `%IBMVS%\bin\vsaudio.bat` file, search for “**:Execute**” to locate the command that starts the VoiceXML browser, and specify the property `-Dvxml.bargein=energy`. When you run the revised batch file, the VoiceXML browser will execute your application using energy-based barge-in detection.

Specifying Properties for Your Deployed Application

When you are ready to deploy your VoiceXML application, you will need to tell the system administrator the appropriate values to set. The IBM WebSphere Voice Server with ViaVoice Technology specifies equivalent properties (plus additional, telephony-specific ones) in a configuration file, **sysmgmt.properties**; the IBM WebSphere Voice Server for DirectTalk specifies its properties in the **default.cff** configuration file. For more information, refer to the documentation for the applicable deployment platform.

Required Properties

The properties shown in [Table 17](#) are *required*; do not change them.

Table 17. Required Properties for Starting the VoiceXML Browser

Required Property	Description
vxml.locale	Specifies the language for the speech recognition and TTS engines. Valid values are en , en-US , and en_US , which are equivalent.
vxml.rate	Specifies the audio sampling rate to be used by the speech recognition engine when decoding audio input. This value must be 8000 .
vxml.vv.user	Specifies the default user name for the speech recognition engine. This value must be Vxmlus .

Starting the VoiceXML Browser

To run your applications using the SDK, you can start the VoiceXML browser using either of two methods:

- “[Command Line Interface](#)” — See [page 92](#).
- “[IBM WebSphere Studio](#)” — See [page 93](#).

Note: In the desktop environment, only one instance of the VoiceXML browser can be run at a time; the IBM WebSphere Voice Server for DirectTalk and the IBM WebSphere Voice Server with ViaVoice Technology deployment platforms support multiple, concurrent VoiceXML browser sessions. Refer to the documentation for the applicable deployment platform for details.

Command Line Interface

The VoiceXML browser runs as a stand-alone application in its own JVM. You can start the VoiceXML browser in a DOS (Java console) window using the following command line interface:

```
"%IBMVS%\bin\batchfile" URL <inputfile
```

where:

IBMVS is the environment variable containing the location of the IBM WebSphere Voice Server SDK installation destination directory.

batchfile is one of the following:

vsaudio Generates spoken output and accepts spoken and simulated DTMF input.

vstext Generates text output and accepts text or simulated DTMF input.

URL is the initial URL for your application.

<inputfile is optional and applies only when using the **vstext** batch file. You can use this parameter to specify a text file containing the user input for your VoiceXML application. This parameter is useful for automated testing purposes, or if your desktop workstation does not have a microphone. For more information, see [“Running Text Mode or Automated Tests” on page 182](#).

IBM WebSphere Studio

IBM WebSphere Studio is a Web development team environment for organizing and managing Web projects. IBM WebSphere Studio version 3.5 has been enhanced to include the following support for VoiceXML files:

- A VoiceXML editor with coding assistance and syntax checking features.
- Wizards capable of generating servlets, JavaBeans, and Java Server Pages for creating dynamic VoiceXML documents.
- The ability to import VoiceXML applications, follow flow control links, and natively understand VoiceXML tags.
- The ability to preview VoiceXML files using the IBM WebSphere Voice Server SDK's VoiceXML browser.
- The ability to publish VoiceXML applications to IBM WebSphere Application Server.

If you are developing your applications using IBM WebSphere Studio version 3.5, you can start the VoiceXML browser by right-clicking on a **.vxml** file in any Studio view and then selecting **Preview with -> WebSphere Voice Server**. Studio launches the batch file **vsaudio.bat**, which invokes the VoiceXML browser and passes it a URL that refers to the selected file. You can modify the startup parameters for the VoiceXML browser by editing the batch file, as described in [“Configurable Properties” on page 87](#).

For more information on using the VoiceXML browser with IBM WebSphere Studio, refer to the IBM WebSphere Studio documentation.

Using the Trace Mechanism

When the VoiceXML browser is running, it writes all trace messages to the file **.\vxml.log**; the VoiceXML browser also writes a subset of trace messages to standard output, displaying them in the Java console window.

Sample vxml.log File

The **vxml.log** file shown below was generated by the interaction described in [“Dynamic Content” on page 71](#):

```
10:25:04.170 S: Starting V000414 at Wed Apr 19 10:25:04 EDT 2000
10:25:04.170 X: Java: IBM Corporation 1.2.2
10:25:11.030 A: listening
10:25:12.620 V: file:C:/Workspace/bin/test1.vxml (fetch get)
10:25:13.830 C: Welcome to Your Bank Online speech demo.
10:25:13.940 C: Please choose: Branch Locations. Interest Rates. Account
Information.
10:25:19.820 A: Audio level (0.6)
10:25:20.150 A: Audio level (0.7)
10:25:20.370 A: Audio level (0.4)
10:25:20.860 A: Audio level (0.4)
10:25:22.070 H: Account Information
10:25:22.070 V: #login
10:25:22.350 C: What is your account number?
10:25:25.590 A: Audio level (0.4)
10:25:26.250 A: Audio level (0.4)
10:25:26.410 A: Audio level (0.4)
10:25:26.740 A: Audio level (0.5)
10:25:28.940 H: one two three
10:25:29.050 C: Please use your telephone keypad to enter your PIN code.
10:25:34.260 T: 1
10:25:35.090 T: 2
10:25:35.750 T: 3
10:25:38.990 T: #
10:25:39.150 H: 1 2 3
10:25:39.320 V: file:/servlet/login?account_number=123&pin_code=123
(fetch get)
```

```
10:25:39.370 E: Event error.badfetch: line 30:
java.io.FileNotFoundException:
\servlet\login?account_number=123&pin_code=123 (The system cannot find
the file specified)
10:25:39.370 C: Sorry, must exit due to processing error.
10:25:43.220 K: clean cache 1
10:25:43.220 K: clean cache 2
10:25:43.270 K: clean cache 3
10:25:43.270 S: exit
```

Note: The contents of the `vxml.log` file are cumulative; you should delete this file once the information is no longer needed.

Format of Trace Entries

The format of each entry in the `vxml.log` file is `hh:mm:ss.sss code: message`, and the format of each entry in the Java console is `code: message`, where:

`hh:mm:ss.sss` is a timestamp, in 24 hour format.

`code` is a single character indicating the type of log entry, as shown in [Table 18 on page 96](#).

`message` is the trace message.

Table 18. vxml.log Trace Codes

Code	Description
A:	<p>Logged when the VoiceXML browser detects audio input, but the speech recognition engine does not return a recognized phrase; this may be due to breath or background noise. The message column contains audio level messages.</p> <p>Note: If you receive an excessive number of these messages, you may want to run the audio setup program; from the Windows Start menu, choose Programs -> IBM WebSphere Voice Server SDK -> Audio Setup and follow the instructions.</p>
C:	Logged when the computer plays a prompt. The message column displays the prompt text.
E:	Logged when the VoiceXML browser throws an event. The message column indicates the type of event.
F:	Logged when the VoiceXML browser fetches a resource such as a grammar file, an audio file, or a script. The message column contains the URI of the file, and whether it was fetched from the server or was in the cache.
H:	Logged when the user responds using voice input. The message column displays the word or phrase that was recognized by the speech recognition engine.
K:	Provides information about cache activity.
S:	Indicates system information, including the version of the VoiceXML browser and the system time. This information is provided only in the vxml.log file; it is not written to the Java console.
V:	<p>Logged when the VoiceXML browser fetches a .vxml file. The message column contains the URI of the file, and whether it was fetched from the server or was in the cache.</p> <p>Note: When transferring control within a document (using a URI that begins with #), the document is not reloaded; in this case, the message column contains the form id.</p>
X:	Specifies the version of the JVM. This information is provided only in the vxml.log file; it is not written to the Java console.
?:	Logged when the speech recognition engine determines that the user said something, but the confidence level is not high enough to justify using the results. In response, the VoiceXML browser throws a nomatch event. The message column contains the word or phrase that was recognized.

Stopping the VoiceXML Browser

The VoiceXML browser terminates normally when the VoiceXML browser processes an **<exit>** element, or if a VoiceXML document has been processed and does not specify a transition to another document. To stop the VoiceXML browser at any other time, you can terminate the browser process by typing **Ctrl-C** in the Java console window or by closing the window in which the Java console is running.

Note: If there was no abnormal activity during the VoiceXML browser session, you may want to purge the `vxml.log` file after you stop the VoiceXML browser.

This chapter covers the following topics:

- ["Introduction"](#) — See [page 99](#).
- ["Design Methodology"](#) — See [page 100](#).
- ["Getting Started – High-Level Design Decisions"](#) — See [page 107](#).
- ["Getting Specific – Low-Level Design Decisions"](#) — See [page 136](#).
- ["Advanced User Interface Topics"](#) — See [page 167](#).

Introduction

The speech user interface guidelines presented here are just that: guidelines. In some cases, the requirements and objectives of particular VoiceXML applications may present valid reasons for overriding certain guidelines. Furthermore, these guidelines address the design points we have found to be the most important in producing speech/audio user interfaces, but are not as comprehensive as those found in a book dedicated to the topic. Finally, keep in mind that while the following guidelines can help you produce a usable application, they do not guarantee usability or user satisfaction; you should plan to conduct usability tests with your application (see ["Design Methodology" on page 100](#)).

Note: **The guidelines and referenced publications presented in this book are for your information only, and do not in any manner serve as an endorsement of those materials. You alone are responsible for determining the suitability and applicability of this information to your needs.**

The goals of the guidelines presented in this chapter include:

- Helping you create standardized, well-behaved VoiceXML applications.
- Reducing development time by taking some of the guesswork out of interface design.
- Increasing the usability of the speech user interface and reducing the end user's learning curve by promoting consistent computer output and predictable user input.
- Decreasing the amount of rework required after prototype testing.

Design Methodology

Developing speech user interfaces, like most development activities, involves an iterative 4-phase process:

- ["Design Phase"](#) — See [page 100](#).
- ["Prototype Phase \(Wizard of Oz Testing\)"](#) — See [page 104](#).
- ["Test Phase"](#) — See [page 105](#).
- ["Refinement Phase"](#) — See [page 106](#).

Because this process is iterative, you should attempt to keep the interface fluid until after user testing.

Design Phase

In this phase, the goal is to define proposed functionality and create an initial design. This involves the following tasks:

- ["Analyzing Your Users"](#) — See [page 101](#).
- ["Analyzing User Tasks"](#) — See [page 102](#).
- ["Making High-Level Decisions"](#) — See [page 103](#).
- ["Making Low-Level Decisions"](#) — See [page 103](#).
- ["Defining Information Flow"](#) — See [page 103](#).
- ["Creating the Initial Script"](#) — See [page 103](#).
- ["Identifying Application Interactions"](#) — See [page 104](#).
- ["Planning for Expert Users"](#) — See [page 104](#).

Analyzing Your Users

The first step in designing your VoiceXML applications should be to conduct user analysis to identify any user characteristics and requirements that might influence application design. For example:

- How frequently will your users use the system?
- What is their motivation for using the system?
- In what type of environment will your users use the system (quiet office, outdoors, noisy shopping mall)?
- What type of phone connection will most of your users have (land-line, cordless, cellular)?
- Are many of your intended users non-native speakers of English?
- How comfortable are your users with automated (“self-service”) applications?
- What are the most common tasks your users will perform?
- What tasks are less common?
- Where does the information that your users need reside? (For example, is it in three different enterprise databases?)
- How do you plan to access that information?
- Are your users familiar with the tasks they will need to complete?
- Will your users be able to perform these tasks by other means (in person, using a visual Web interface, by calling a customer service representative, etc.)?
- What words and phrases do your users typically use to describe the tasks and items in your proposed application?

[Table 19 on page 102](#) illustrates some of the ways in which user characteristics can impact application design.

Table 19. Influence of User Profile on Application Design

User Characteristic	Application A - Stock Quotes	Application B - Employee Benefits enrollment
<i>Frequency of Use</i>	Daily - Users will become experienced and may want to be able to cut through the interface to complete common tasks more quickly. See “Customizing Expertise Levels” on page 167.	Annually - Users won’t remember how to use the application from one session to the next, and may therefore need (and be willing to hear) more verbose prompts.
<i>Access hours</i>	24 hours a day, 7 days a week - Users may rely on this “self-service” application and therefore be more motivated to use it.	Normal business hours - Users may have other ways to access the data (for example, by speaking with a Human Resources representative). The availability of alternative access methods may make users less tolerant of any dissatisfaction with the voice application.
<i>Expertise in application domain</i>	Expert - Users know what to do and use a common terminology (buy, sell, short, margin, etc.); your prompts should use this terminology.	Novice - Users may need more guidance, and you may need to define certain terms for them (for example, ESPP=employee stock purchase plan). Users may also give less predictable responses; this impacts prompt, grammar, and help design.

Analyzing User Tasks

After you have identified who your users are, the next step is to determine what tasks your application should support, and assign them to the application or user based on the strengths of each.

For example, tasks in a banking application could include transferring money, obtaining current account balance, listing some number of most recent transactions, etc. You might allocate the functions as follows:

- Have the application locate, sort, and store account information, and make any routine decisions. For example, if the user attempts to transfer more money than is available, the application plays a warning message.

- Have the user confirm transactions and make any non-routine or elective decisions. For example, ask the user to confirm the amount of money and account number before the application submits a form that initiates a monetary transfer.

Making High-Level Decisions

The next step is to make high-level application decisions, such as selecting the appropriate user interface, half- versus full-duplex implementation, prompt style, and help style. For details, see [“Getting Started – High-Level Design Decisions” on page 107](#).

Making Low-Level Decisions

After you have made the high-level decisions, you should proceed to the lower-level system decisions that address such issues as sound and feel, word choice, etc. [“Getting Specific – Low-Level Design Decisions” on page 136](#) provides information to help you make these decisions.

Defining Information Flow

Next, you will want to outline an information flow that maps the interaction between your application and the user. For example:

- What questions do you need to ask the user?
- When the user answers, how should the application respond?

Your application interaction should have a logical progression that takes into account typical responses, unusual responses, and any error conditions that might occur.

Creating the Initial Script

After you have defined the information flow, you should be ready to create an initial draft of the script for the dialog between the application and the user. The script should include all of the text that will be spoken by the application, as well as expected valid user utterances.

Identifying Application Interactions

Next, you should look through the draft script and try to identify any intra-application and inter-application interactions in the scripted dialogs. For example, a movie selection application might ask:

- Which movies are playing?
- Where is a particular movie playing?
- When is a particular movie playing at a specified theater?

These queries are, by nature, sequential; since the information for the second and third queries depends on the results of the previous queries, there is an intra-application dependency.

Sometimes, studying the sequence of tasks may point to inefficiencies in the underlying business processes. For example:

- An initial dialog prompts for the user's account number.
- The user takes certain actions, eventually transitioning to a separate underlying application (transparent to the user).
- Because the new application uses a different back-end database, you might be tempted to reprompt for the account number (especially if this is how it is done in the existing visual web page).
- Instead of prompting the user to provide the information again, you might consider using cookies, variables in application root document, or a site configuration file to store the user's account number and pass this data to the new back-end database.

Planning for Expert Users

As the final step in the Design phase, you may want to identify the potential for expert users and begin considering where you may be able to help them cut through some of the interface to quickly perform common tasks.

Prototype Phase (Wizard of Oz Testing)

The goal of this phase is to create a prototype of the application, leaving the design flexible enough to accommodate changes in prompts and dialog flow in subsequent phases of the design.

For the first iteration, you may want to use a technique known as “*Wizard of Oz*” testing. This technique can be used before you begin coding, as it requires only a prototype paper script and two people: one to play the role of the user, and a human “wizard” to play the role of the computer system. Here’s how it works:

- The script should include the proposed introduction, prompts, list of always-active commands, and all planned self-revealing help.
- The two participants should be physically separated so that they cannot use visual cues to communicate; a partition will suffice, or you could use separate rooms and allow the people to communicate by telephone.
- The wizard must be very familiar with the script, while the user should never see the script.
- The user telephones (or pretends to telephone) the wizard, who begins reading the script aloud. The user responds to the prompts, and the wizard continues the dialog based on the scripted response to the user’s utterance.

Wizard of Oz testing lets you fix problems in the script and task flow before committing any of the design to code. However, it cannot detect other types of usability problems, such as recognition or audio quality problems; these types of errors require a working prototype.

Test Phase

After you’ve incorporated the results of the Wizard of Oz testing, you will want to code and test a working prototype of the application. During this phase, be sure to analyze the behavior of both new and expert users.

Identifying Recognition Problems

As you proceed with the Test phase, note any *consistent* recognition problems.

The most common cause of recognition problems is acoustic confusability among the currently active phrases. For example, both Madison and Addison are U.S. airports. Thus, these potential user inputs to a travel application are highly confusable:

User:	Flying from Madison
User:	Flying from Addison

Sometimes there isn't anything you can do when this happens. Other times you can try to correct the problem by:

- Using a synonym for one of the terms. For example, if the system is confusing 'no' and 'new', you might be able to replace 'new' with 'recent', depending on the application's context. Similarly, if the system is confusing Bach (the musician) and Back (a command), try changing the command to Go Back.
- Adding a word to one or more of the choices. For the Madison/Addison airport confusion, you could make states optional in the grammar for most cities, but require the state for low-traffic airports that have acoustic confusability with higher-traffic airports.

Identifying Any New Application Interactions

At this point, you may also want to note any newly identified inter-application and intra-application interactions.

Identifying Any User Interface Breakdowns

The Test phase is also where you will identify potential user interface breakdowns. Some factors you may want to analyze include:

- Percentage of users who did not successfully complete your test scenarios
- Points in the application where users experienced the most difficulty
- Unexpected user behaviors
- Effectiveness of error recovery mechanisms
- Time to complete typical transactions
- Self-reported level of user satisfaction

The first round of user testing typically reveals places where the system's response needs to be rephrased to improve usability. For this reason, system responses should be left flexible until after the first round of user testing.

Refinement Phase

During this phase, you will be updating the user interface based on the results of testing the prototype.

For example, you may revise prototype scripts, add tapered prompts and customizable expertise levels, create dialogs for inter- and intra-application interactions, and prune out dialogs that were identified as potential sources of user interface breakdowns.

Finally, you will want to iterate the Design—Prototype—Test—Refine process, including in the Test phase both users from previous rounds of testing and users new to the system.

Getting Started – High-Level Design Decisions

Designing a speech user interface involves at least two levels of design decisions. First, you need to make certain high-level design decisions regarding system-level interface properties. Only then can you get down to the details of designing specific system prompts and dialogs.

The high-level decisions you need to make include:

- ["Selecting an Appropriate User Interface"](#) — See [page 108](#).
- ["Deciding on the Type and Level of Information"](#) — See [page 109](#).
- ["Choosing Full-Duplex \(Barge-In\) or Half-Duplex Implementation"](#) — See [page 109](#).
- ["Selecting Recorded Prompts or Synthesized Speech"](#) — See [page 114](#).
- ["Deciding Whether to Use Audio Formatting"](#) — See [page 116](#).
- ["Using Simple or Natural Command Grammars"](#) — See [page 118](#).
- ["Adopting a Terse or Personal Prompt Style"](#) — See [page 121](#).
- ["Allowing Only Speech Input or Speech plus DTMF"](#) — See [page 122](#).
- ["Adopting a Consistent Set of Always-Active Navigation Commands"](#) — See [page 125](#).
- ["Choosing Help Mode or Self-Revealing Help"](#) — See [page 131](#).

There is no single correct answer; the appropriate decisions depend on the application, the users, and the users' environment(s). The remainder of this section presents the tradeoffs associated with each of these decisions.

Selecting an Appropriate User Interface

The first decision that you must make is to select the appropriate user interface for your application. Not all applications are well-suited to a speech user interface; some work best with a visual interface, and others would benefit from a *multi-modal* interface (that is, both a speech and a visual interface).

The characteristics in [Table 20](#) can help you decide whether your application is suited to a speech user interface.

Table 20. When to Use a Speech Interface

Consider using speech if:	Applications may not be suited to speech if:
Users are motivated to use the speech interface, because it: <ul style="list-style-type: none">• Saves them time or money.• Is available 24 hours a day.• Provides access to features not available through other means.• Allows them to remain anonymous and avoid discussing sensitive subjects with a human.	Users are not motivated to use the speech interface.
Users will not have access to a computer keyboard when they want to use the application.	The nature of the application requires a lot of graphics or other visuals (for example, maps or commerce applications for apparel).
Users want to use the application in a “hands-free” or “eyes-free” environment.	Users will be operating the application in an extremely noisy environment (due to simultaneous conversations, background noise, etc.)
Users are visually impaired or have limited use of their hands.	Users are hearing impaired or have difficulty speaking.

Deciding on the Type and Level of Information

To keep from overloading the user’s short-term memory, information presented in a speech user interface must generally be more concise than information presented visually. Often, only the most essential information should be presented initially, with the opportunity for the user to access detailed information at a lower level.

For example, consider a banking application in which a user can request a list of recently cleared checks. In a visual interface, the application might return a table showing the check number, date cleared, payee name, and amount. A similar application with a speech interface might return only the check number and date cleared, and then permit the user to select a specific check number to hear the payee name and amount, if desired.

Choosing Full-Duplex (Barge-In) or Half-Duplex Implementation

Full-duplex implementations allow the computer and the user to speak at the same time, permitting the user’s speech to interrupt system prompts as the machine plays them, a feature known as “*barge-in*”. (See “[Barge-in](#)” on page 83.) Full-duplex implementations require good echo-cancellation, which you must configure in the telephony hardware for deployment.

On the surface, it might seem as if full-duplex would always be preferable to half-duplex, because it is easy to imagine experienced users wanting to interrupt prompts (especially lengthy ones) when they know what to say. There are situations, however, in which a half-duplex system will be as easy or easier to use than a full-duplex system.

[Table 21 on page 110](#) compares full- and half-duplex implementations:

Table 21. Full-duplex versus Half-duplex Implementation

Implementation	Advantages	Disadvantages
Full-duplex	<p>Experienced users can interrupt system prompts to speed up the interaction.</p> <p>Users can say “Quiet” to stop the prompt, and “Repeat” to replay the prompt. See “Built-in Commands” on page 84 for more information on these commands.</p>	<p>Inexperienced users may inadvertently interrupt the prompt before hearing enough to form an acceptable response. You can minimize this problem by keeping system prompts short, to lessen the user’s need to barge-in; if your prompts are long, you should try to present key information early in the prompt.</p> <p>When using recognition barge-in detection (see Table 22 on page 111), Lombard speech and the stuttering effect can be problematic. To minimize this problem, you should keep required user inputs very short; see “Controlling Lombard Speech and the Stuttering Effect” on page 112 for more information.</p>
Half-duplex	<p>Guarantees that the entire prompt text plays. This may be especially useful for applications with lots of legal notices, advertisements, or other information that you want to make sure always gets presented to the user.</p> <p>Creates a “my turn-your turn” rhythm for the dialog.</p>	<p>Experienced users cannot interrupt prompts; however, if the prompts are short enough, users shouldn’t need to interrupt.</p> <p>Inexperienced users may experience turn-taking errors. Keeping prompts short helps minimize this.</p>

Note: Applications deployed using the IBM WebSphere Voice Server for DirectTalk are always full-duplex; however, you can disable barge-in for individual prompts (<prompt bargein=“false”>) or the entire application (<property name=“bargein” value=“false”>).

Comparing Barge-in Detection Methods

To understand how to use full-duplex effectively, it is important to understand how the system determines when to stop an interrupted prompt. For the IBM WebSphere Voice Server SDK, the barge-in detection method is controlled by the value of the `vxml.bargein` Java system property specified when you start the VoiceXML browser (see [Table 16 on page 88](#)).

[Table 22](#) compares the available barge-in detection methods.

Table 22. Barge-in Detection Methods

Barge-in Detection Method	Description	Advantages	Disadvantages
<i>recognition</i>	<p>Audio output stops only after the system determines that the user has spoken a complete word or phrase that is valid in a currently active grammar.</p> <p>Note: This barge-in detection method is not available in the IBM WebSphere Voice Server for DirectTalk.</p>	<p>Resistant to accidental interruptions such as those caused by:</p> <ul style="list-style-type: none"> • coughing • muttering • using the system in an environment with loud ambient conversation 	<p>Increased incidence of <i>Lombard speech</i> and the “stuttering” effect (see next section); however, you can control this somewhat by making required user responses as short as possible.</p>
<i>energy</i>	<p>Audio output stops as soon as the speech recognition engine detects sound.</p>	<p>Minimizes Lombard speech, the stuttering effect, and the distortion to the first syllable of user speech that often occurs when users barge-in.</p>	<p>Susceptible to accidental interruption due to background noise and non-speech sounds. May require a corresponding increase in the use of confirmation dialogs, which may disrupt the dialog flow.</p>

Controlling Lombard Speech and the Stuttering Effect

When speaking in noisy environments, people tend to exaggerate their speech or raise their voices so that others can hear them over the noise. This distorted speech pattern is known as “*Lombard speech*”, and it can occur even when the only noise is the voice of another participant in the conversation (for example, when one person tries to interrupt another, or, in the case of a voice application, when the user tries to barge-in while the computer is speaking).

The “*stuttering effect*” may occur when a prompt keeps playing for more than about 300 ms after the user begins speaking. Unless users have undergone training with the system, they may interpret the continued playing of the prompt as evidence that the system didn’t hear them. In response, some users may stop what they were saying and begin speaking again – causing a stuttering effect. This stuttering makes it virtually impossible for the system to match the utterance to anything in an active grammar, so the system generally treats the input as an “*out-of-grammar*” utterance, even if what the user *intended* to say was actually in one of the active grammars.

To control Lombard speech and the stuttering effect when using recognition barge-in detection, the prompt should stop within about 300 ms after the user begins talking. Because the average time required to produce a syllable of speech is about 150-200 ms, this means that the system design should promote short user responses (ideally no more than two or three syllables) when using recognition barge-in detection. You should also try to keep prompts as short as possible, to minimize the likelihood that users will want to interrupt the prompt. If this is not possible, you should consider either switching to energy barge-in detection or designing the system to be half-duplex rather than full-duplex.

Weighing User and Environmental Characteristics

When deciding whether to use barge-in and which type of barge-in detection is most appropriate, you should consider how frequently users will use your application (expert users are more likely to barge-in), and in what environment (quality of the telephone connection, general noise level, etc.).

In general, you should enable barge-in for deployed applications. However, if echo cancellation on your telephony equipment is not good enough, it may be necessary to disable barge-in or switch to half-duplex.

Some users may be hesitant to barge-in, since doing so violates a norm of what is considered polite human conversation in many cultures: not talking while someone else is talking.

Minimizing the Need to Barge-in

To minimize the user’s need to barge-in, you might consider placing extremely short (say, 0.75 second) recognition windows at logical pausing places during and between prompts, such as at the end of a sentence or after each menu item. These brief pauses will give users the opportunity to talk without feeling that they are “being rude” by interrupting the application.

Using Audio Formatting

If you need to temporarily disable barge-in (using `<prompt bargein="false">`, such as while the system reads legal notices or advertisements), you may want to use a unique background sound, tone, or prompt as an indicator. For guidance, see [“Applying Audio Formatting” on page 116](#).

In half-duplex systems, we strongly recommend that you consider playing a beep or tone (using an audio file) to signal the user when it is his or her turn to speak; the introductory message should explicitly tell users to speak only after this “turn-taking” tone. In the examples in this document, [!!!] represents a turn-taking tone for half-duplex systems; it is not required for systems that have barge-in enabled.

Note: The use of tones to signal user input is somewhat controversial, with some designers avoiding tones based on a belief that tones are unnatural in speech and annoying to users. Others contend that effective computer speech interfaces need not perfectly mimic human conversation, and that a well-designed tone can promote clear turn-taking without annoyance. For guidance in creating an effective turn-taking tone, see [“Designing Audio Tones” on page 116](#).

Wording Prompts

For systems without barge-in, you should try to place key information near the end of the prompt, to discourage users from beginning to speak before it is their turn.

You can do the same for systems with barge-in, assuming your prompts are relatively short; if the prompts are long, you may decide to move the key information to the beginning of the prompt, so that users know what input to provide if they interrupt the prompt.

Selecting Recorded Prompts or Synthesized Speech

Synthesized speech (“text-to-speech” or “TTS”) is useful as a placeholder during application development, or when the data to be spoken is “*unbounded*” (not known in advance) and cannot therefore be prerecorded.

When deploying your applications, however, you should use professionally recorded prompts whenever possible. Users expect commercial systems to use high-quality recorded speech; only recorded speech can guarantee highly natural pronunciation and prosody.

Creating Recorded Prompts

The following guidelines will help you generate high quality recorded prompt:

- Use professional voices, quality recording equipment, and a suitable recording environment.
- Maintain consistency in microphone placement and recording area.
- If prompts contain long numbers, or if many of the application’s users are not native speakers of the language in which the application speaks, consider slowing down the speech.
- Aggressively trim (that is, remove the beginning and ending silence from) recorded prompts.
- As a general rule, use only one voice. When using multiple voices, have a clear design goal (for example, a female voice for introduction and prompts, and a male voice for menu choices). Otherwise, changing voice within the application may interfere with the user’s ability to become familiar and comfortable with the voice used.

When using recorded prompts, you can improve performance by prefetching and caching the audio files. See [“Fetching and Caching Resources for Improved Performance” on page 177](#).

Using TTS Prompts

Although recorded prompts are best for many applications, it is important to keep in mind that it is easier to maintain and modify an application that uses TTS prompts. For this reason, TTS prompts are typically used during application development.

When you are ready to deploy your application, we recommend that you use recorded prompts when possible; however, if part of a prompt or other message requires production via TTS, it is generally better to generate that entire prompt with TTS to avoid the jarring juxtaposition of recorded and artificial speech.

Handling Unbounded Data

If the information that the application needs to speak is unbounded, you will need to use TTS for prompts. Examples of unbounded information include:

- Telephone directories
- E-mail messages
- Frequently updated lists of employee or customer names, movie titles, or other proper nouns
- Up-to-the-minute news stories

Improving TTS Output

You can improve the quality of synthesized speech output by using VoiceXML's text-to-speech markup elements to provide additional information in the input text. For example:

- You can improve the TTS engine's structural analysis by using the **<div>** element to explicitly mark paragraph and sentence structure.
- You can improve the TTS engine's processing of acronyms, abbreviations, numerical constructs, uncommon names, and even email and Web addresses by using the **<sayas>** element to specify the desired pronunciation.
- For synthesized speech, a speed of 150-180 words per minute is generally appropriate for native speakers. You can use the **<pros>** element to slow down the speed of TTS output on a prompt-by-prompt basis for long numbers, or if many of the application's users are not native speakers of the language in which the application speaks.
- You can further improve the prosody of the TTS output by using the **<break>** and **<emp>** elements.

For example:

```
<emp level="strong"> Wait for the beep</emp> before you speak.
```

- You can also change also the gender and age characteristics of the TTS voice by using the **<ibmvoice>** element. As with recorded prompts, however, it is generally a good idea to use a single voice throughout your application unless there is a clear design goal that requires multiple voices.

Deciding Whether to Use Audio Formatting

Audio formatting is a useful technique that increases the bandwidth of spoken communication by using speech and non-speech cues to overlay structural and contextual information on spoken output. Audio formatting is analogous to the well-understood notion of visual formatting, where designers use font changes, indenting, and other aspects of visual layout to cue the reader to the underlying structure of the information being presented.

Designing Audio Tones

The downside to using audio formatting is that there are not yet standard sounds for specific purposes. If you plan to use audio formatting, you may want to work with an audio designer (analogous to a graphic designer for graphical user interfaces) to establish a set of pleasing and discriminable sounds for these purposes.

When designing audio formatting, the tones should be kept short: typically no longer than 0.5-1.0 seconds, and even as short as 75 ms. Shorter tones are generally less obtrusive, and users are therefore more likely to perceive them as useful, rather than distracting.

Applying Audio Formatting

You can use non-speech cues to indicate dialog state, exceptions to normal system behavior, and content formatting, as described in [Table 23 on page 117](#).

Table 23. Audio Formatting

Purpose	Recommendations
<i>Turn-taking tone (half-duplex only)</i>	<p>For half-duplex applications, you may want to use audio formatting to indicate when it is the user’s turn to speak, as described in Table 21 on page 110. An effective turn-taking tone will generally have the following characteristics:</p> <ul style="list-style-type: none"> • duration of 75-150 ms • pitch of 750-1250 kHz • not too loud • gentle on the ear (a complex wave rather than sinusoid). <p>Note: In the examples in this document, [!!!] represents a turn-taking tone.</p>
<i>Barge-in temporarily disabled (full-duplex only)</i>	<p>When barge-in is temporarily disabled on a full-duplex application (say, when legal notices are read), you may want to play a unique background sound or use a special tone or prompt as an indicator.</p> <ul style="list-style-type: none"> • For recorded audio prompts, you will need to prerecord the speech mixed with the background sound. • For synthesized (TTS) prompts, you can use an introductory tone or prompt when you disable barge-in, and another tone or prompt to let the user know when you have re-enabled barge-in. <p>For more information on barge-in, see “Barge-in” on page 83.</p>
<i>Audio cue for bulleted list</i>	Consider using a short sound snippet as an auditory icon.
<i>Audio cue for emphasis (akin to visual bold and italics)</i>	Consider using an auditory inflection technique, such as changing volume or pitch.
<i>Audio cue for secure transactions</i>	For secure transactions, you may want to play a unique background sound or use a special tone or prompt. See the recommendations for “Barge-in temporarily disabled”, above.

For additional uses of audio formatting, see [“Additional Opportunities for Exploiting Audio Formatting” on page 173](#).

Using Simple or Natural Command Grammars

The VoiceXML browser uses grammar-based speech recognition, as explained in [“Grammars” on page 72](#).

Grammars can be very simple or extremely complex, as explained in [Table 24](#). The appropriate type to use depends on your application and user characteristics.

Table 24. Simple versus Natural Command Grammars

Type of Grammar	Description	Advantages	Disadvantages
<i>Simple Grammar</i>	A grammar that includes basic words and phrases that a user might reasonably be expected to say in response to a directed prompt. Many applications will not require complex grammars to be effective, as long as the system prompts constrain likely user input to the valid responses specified by the grammar.	Easier to code and maintain. When used with properly worded prompts, can be as effective as much more complex grammars.	When taken to the extreme, may impose excessive restrictions on what users can say. For example, while you could code an application that requires users to respond to every prompt with a “YES” or a “NO”, this would result in a cumbersome interface for all but the simplest of applications.
<i>Natural Command Grammar</i>	A very complex grammar that approaches <i>natural language understanding (NLU)</i> in its lexical and syntactic flexibility.	Can enhance the ability of the system to recognize what users are saying. Can increase dialog efficiency. See “Using Menu Flattening (Multiple Tokens In a Single User Utterance)” on page 119 .	Time-consuming to build and more difficult to maintain. Uses more system resources, possibly impacting performance.

Designing Simple Grammars

Simple grammars do not attempt to cover *all* possible ways that a user could respond; rather, you word your prompts in a way that guides users to speak one of a reasonably-sized list of responses, and you code your grammars to accept those responses.

The number and types of responses you will want to support (and therefore, the size and complexity of your grammar) depend on your application and your users.

Evaluating the Need for Natural Command Grammars

New speech user interface designers often start out by assuming that unless a system has NLU or NLU-like capability, it will not be usable. *This assumption is simply not true.* Well-designed prompts can focus user input so that a fairly small grammar has an excellent chance of matching the user input. In many ways, this type of system will be more usable and satisfying (although arguably less natural) than a system with more complex recognition capabilities. Furthermore, the performance consequences of using more complex grammars in a client-server setting could adversely affect overall system usability.

For these reasons, using focused prompts and smaller grammars is often preferable to using natural command grammars.

Using Menu Flattening (Multiple Tokens In a Single User Utterance)

Natural command grammars do, however, have some potential usability advantages in certain settings and with certain users. Specifically, they have the potential for substantial *menu flattening* because the system can parse the user input and extract multiple tokens (where a *token* is the smallest unit of meaningful linguistic input), rather than requiring the user to provide these tokens one at a time. For example, if a user says to a travel reservations application, “Tell me all the flights from Miami to Atlanta for tomorrow before noon”, there are at least six tokens: all (rather than one or two), flights (rather than bus trips or trains), Miami (departure point), Atlanta (destination), tomorrow (date), before noon (time).

Note: **Mixed initiative forms that permit multiple fields to be filled in from a single utterance offer some of the same menu-flattening benefits. See [“Mixed Initiative Application and Form-level Grammars” on page 79.](#)**

To take full advantage of the increased efficiency that menu flattening provides, the system prompts must encourage users to provide input with multiple tokens. Therefore, appropriate prompts for a natural command system are quite different from appropriate prompts for simple grammar systems.

Promoting Consistency Through Built-in Grammars

Regardless of your decision on which type of grammars to use, remember that the way the system behaves at any one point in the dialog sets the user's expectations of the system behavior throughout the application. Using the built-in grammars is an easy way to ensure consistency; if you want to augment a built-in grammar, you can code a grammar containing the additional terms and make it active every time you use the built-in grammar.

The VoiceXML browser offers several built-in grammars, including simple grammars for the following built-in commands (see [“Built-in Commands” on page 84](#)):

- Help
- Quiet
- Repeat

and more complex grammars for the following built-in types (see [“Built-in Field Types and Grammars” on page 54](#)):

- Boolean (yes/no)
- Currency
- Date
- Digits
- Number
- Phone
- Time

Adopting a Terse or Personal Prompt Style

As one part of developing a consistent ‘sound and feel’ for the interface, you need to decide whether you will use terse or personal prompts. Each has advantages and disadvantages, as described in [Table 25](#):

Table 25. Prompt Styles

Prompting Style	Advantages	Disadvantages
<i>Terse</i>	<p>Uses the fewest words possible; makes efficient use of time.</p> <p>Often leads user to respond as if talking to a machine, producing well-regulated spoken responses that are easy to recognize.</p>	<p>May be perceived as machine-like and impersonal.</p>
<i>Personal</i>	<p>Perceived as less impersonal, more human-like.</p>	<p>May lead to verbose prompts, especially if you don’t adhere to the guidelines on prompt length, tapering, and wording.</p> <p>May cause the user to ascribe an excessive level of intelligence to the system; if this proves inconsistent with the actual abilities of the system, the interface can fail rapidly. You can minimize this risk by telling users up front that they are speaking to a computer (for example, “Welcome to the automated banking system.”)</p> <p>Verbose prompts and mistaken assumptions about system intelligence can also lead users to produce responses that are not in the active grammar.</p>

Weighing Demographic Factors

Accepted standards for terminology, formality, and interaction style in spoken communication vary widely based on demographic factors such as age, culture, and socioeconomic status, as well as the subject matter or purpose of the conversation. Obviously, you will want to design your application to conform to these behavioral norms.

Using Terse Prompts

In most cases, the balance will tip in favor of terse prompts; however, your decision will depend on the purpose of your specific application.

Using Personal Prompts

If you elect to use personal prompts, you will want to give careful consideration to the following:

- Designing your dialogs to accept multiple pieces of information in a single utterance, and to parse the input regardless of order. See [“Mixed Initiative Application and Form-level Grammars” on page 79](#).
- Robustness of the vocabulary and expressions that you will support.

Allowing Only Speech Input or Speech plus DTMF

Applications that mix speech and DTMF are called *mixed-mode applications*. Because speech applications and DTMF applications do not typically have the same ‘sound and feel,’ it can be tricky to mix the two. For this reason, it is generally better not to explicitly design mixed-mode applications, unless you are migrating from a legacy DTMF application to a speech-enabled version.

Note: **VoiceXML’s built-in types (grammars) automatically enable DTMF input; however, your system prompts should generally *not* attempt to mention both speech and DTMF. The application interface will be simpler if your prompts are focused primarily on speech.**

Choosing the Architecture of Mixed-Mode Application

If you must mix modes, one of your first design decisions must be to choose the fundamental architecture of the mix. There are four choices, as described in [Table 26 on page 123](#):

Table 26. Mixed Input Modes

Type of Mix	Advantages	Disadvantages
<i>Completely separate speech and DTMF interfaces. Users make the choice as their first interaction with the system.</i>	Most straightforward approach.	Requires you to maintain two separate applications. Both applications require full functionality. Users can't switch from one mode to the other.
<i>A unified system that allows users to freely switch between speech and DTMF modes, perhaps with the use of a DTMF key or key sequence.</i>	Almost as straightforward as separate interfaces. Users can switch between modes.	Users must deal with two different interfaces. Users might experience mode errors – thinking they are in one mode when they are in the other.
<i>A single application with DTMF-style prompts</i>	Users only experience one interface.	Speech is not used to its full advantage. Note: Try to put the DTMF action before the speech command, because users (even in half-duplex systems) can interrupt the prompt with a DTMF keypress. For example, “Press 1 for Recent Account Transactions.”
<i>Speech as the primary interface, with DTMF support as required.</i>	Seamless multi-modal user interface. Makes best use of both technologies.	Most expensive and time-consuming to develop. If based on a legacy system, makes previously experienced users novices again.

Deciding When to Use DTMF

You may want to consider using DTMF for password entry and/or confirmation of sensitive transactions that users would not want overheard. However, you should keep in mind the difficulty this can pose for users of mobile and rotary pulse phones, as well as phones where the keypad is on the handset.

If your application supports both speech and DTMF modes, you may want to switch to DTMF mode automatically if the user is experiencing *consistent* speech recognition errors. With good error recovery techniques (see [“Managing Errors” on page 163](#)), it is not generally necessary or desirable to switch to DTMF mode after a single recognition error.

You may also want to consider using a DTMF key sequence to allow the user to switch to DTMF mode; if severe recognition problems are what is causing the user to want to switch modes, providing only a spoken command for mode switching may prove ineffective.

Structuring Mixed Mode Applications

When feasible, you may want to move any interactions requiring DTMF input to the beginning of the application. This is especially important for password entry or other interactions involving secure information, because once users start talking to the application, they might continue to do so even when explicitly instructed to press keys; in some cases, this could compromise passwords.

Wording DTMF Prompts

You should consider using one word to introduce prompts for a single DTMF digit or command sequence and a different word to prompt for a string of digits. For instance, you might use “Press” to indicate that the user should input a single DTMF digit or command, and “Enter” to indicate that the user should input a string of DTMF digits. For example:

```
System: Press *s to stop a prompt.
```

For applications that will not be translated, you may want to make the DTMF prompts mnemonic, like using “*s” for “stop” in the previous example.

Whenever possible, you should avoid “Press or say” prompts. For example, provide menus like this:

```
System: Please say one of the following:  
        Password services  
        Account services  
        E-mail options [!!!!]
```

rather than:

```
System: For password services, say 'password services' or  
        press '1.' [!!!!]
```

Adopting a Consistent Set of Always-Active Navigation Commands

Selecting the Command List

The basic commands recommended for any speech application are listed in [Table 27 on page 126](#); commands shown in **bold** in the table are the minimum recommended set.

Table 27. Recommended List of Always-Active Commands

Always-Active Commands	Description
Backup	Backs up to the previous prompt. See page 127 .
Exit	Exits the application. See page 127 .
Help	Provides help. See page 128 .
List Commands	Lists always-active commands. See page 128 .
Quiet/Cancel	Stops playback of the current prompt. See page 128 .
Repeat	Repeats the last prompt. See page 129 .
Start Over	Returns to the beginning of the interaction. See page 129 .
What-Can-I-Say-Now	Lists all available commands. See page 130 .

Constructing Always-Active Commands

While the specific implementation of each command depends on the requirements of your application, the general function is described here.

Built-in Commands

The VoiceXML browser includes simple grammars for Help, Quiet/Cancel, and Repeat (see “[Built-in Commands](#)” on [page 84](#)); depending on the grammar style (simple or natural command grammar) of the rest of your application, you might want to add alternate ways of saying these commands by creating additional, auxiliary grammars and making them always active.

Application-specific Commands

When you implement these commands, you should construct them in grammars that are always active (by specifying them in the application root document with `<form scope="document">`), so that users can access them throughout your application. However, there is a tradeoff to having a large and robust set of always-active commands: the potential for misrecognition increases. You should therefore exercise extra care when constructing these grammars.

Backup

This command lets users back up to the previous prompt.

In any reasonably complex application, users (especially first-time users) might need to explore the interface; while exploring, they might go down an unintended path, and will need a command to backup through the menu (dialog) structure. For example:

```
System:    Say one of the following choices:
           Leave message
           Camp
           Forward call [!!!!]
User:      Forward call
System:    Forward to which 4-digit extension? [!!!!]
User:      Backup
System:    Say one of the following choices:
           Leave message
           Camp
           Forward call [!!!!]
```

Exit

This command lets users exit the system.

Most users will probably just hang up when they have finished using the system; some users, however, might feel more comfortable if they can say “Exit” and hear the system confirm task completion with a closing message.

When this command is used, the application can also alert the user if a task has been left incomplete, and can prompt the user to complete the task. For example:

```
User:      Exit.
System:    You haven't specified an appointment date. Do you
           want to cancel this appointment? [!!!!]
User:      Yes.
System:    Goodbye. Thank you for using our automated
           appointment system.
```

Note: If you find that users are frequently triggering the Exit command accidentally due to acoustic confusability with other active commands, you may need to reword the other commands.

The Exit command also provides an opportunity to engage users in post-usage satisfaction surveys.

Help

This command starts a help mode or causes the application to play the next prompt in a sequence of self-revealing help prompts. See [“Choosing Help Mode or Self-Revealing Help” on page 131](#) for more information.

“List Commands”

If your application has a large number of commands that you would like to be always active, you might consider offering users a “List Commands” command.

Quiet/Cancel

This command stops playback of the current prompt in full-duplex applications with barge-in.

Repeat

This command repeats the last prompt. (Refer to the description of the `<reprompt>` element in the VoiceXML 1.0 specification for implementation details.) For example:

```
System: The conversion rate from French francs to Australian
        dollars is 1.5678 dollars per franc. Would you like to
        do another conversion? [!!!!]

User:   Repeat.

System: The conversion rate from French francs to Australian
        dollars is 1.5678 dollars per franc. Would you like to
        do another conversion? [!!!!]
```

Start Over

This command lets users start over when they want to abandon the path they are on and return to the beginning of the interaction. For example:

```
System: Which option?
        Phone
        Fax [!!!!]

User:   Phone.

System: Which phone action?
        Leave message
        Camp
        Forward call [!!!!]

User:   Forward call.

System: Forward to which 4-digit extension? [!!!!]

User:   Start over.

System: Which option?
        Phone
        Fax [!!!!]
```

If your application contains multiple modules, it might not be clear at which point the system will restart. In this case, you might want to code a set of commands in always-active grammars to allow users to jump to the different modules or back to the main menu. For example, the commands LIBRARY, BANKING, CALENDAR, and MAIN MENU might be in always-active grammars for a voice portal application with library, banking, and calendar modules.

“What-Can-I-Say-Now”

For most interactions, the Repeat and Backup commands will suffice; however, if you think that your users might need additional information about what they can say at a given point in their interaction with an application, you might consider including an always active “What-Can-I-Say-Now” command. This command lists all the commands that are currently active, including those in the always-active grammars; if you have defined a List Commands command, you may choose to reference it instead of listing all of the always-active commands.

If your application includes dialogs where the list of things that the user can say is *very large* (say, a dialog in which the user can specify any of the more than 6000 airports in the world), you will need to take a different approach when implementing the “What-Can-I-Say-Now” command: *describing* (rather than listing) the valid user input. For example, “You can say the name of any airport in the world.”

Using the Always-Active Commands

Simply having these commands in the system isn’t always sufficient to make them usable; you may need to let users know that they exist. You can accomplish this in two ways:

1. Make sure that the application’s introductory message tells users about the two or three most important always-active commands.

It is not always necessary or even desirable to list *all* of the always-active commands; remember that if the user accidentally or naturally speaks one that wasn’t mentioned, it will still have the desired effect.

Users will generally remember the two or three commands, so choose them well; the appropriate commands to mention will depend on the application, but usually include Repeat and Start Over. For example:

```
System: Welcome to the automated monetary conversion system.
        The commands 'Repeat' and 'Start Over' are always
        available.
```

- At some point in the sequence of help prompts, inform the user about the other always-active commands. See [“Mentioning Always-Active Commands” on page 134](#) for more details.

Choosing Help Mode or Self-Revealing Help

Certainly, you will want to have the command Help (and possibly a set of alternate phrases that act as synonyms for help) in an always-active grammar. But what should you do when a user asks for help? You can either switch to a separate help mode, or engage in a technique known as *self-revealing help*. [Table 28](#) compares and contrasts these help styles:

Table 28. Comparison of Help Styles

Help Mode	Self-Revealing Help
Help is provided through a separate dialog.	Help is integrated into each application dialog.
You will need to save the state of your application dialog so that you can return to it when the user exits help mode, and you must tell the user how to exit help mode. Alternately, you could use a “one-shot” help mode message that automatically returns to the application after presenting the help text.	Does not require explicit mode management.
You may want to use a <i>help mode indicator</i> (such as a background sound, a different voice, or introductory and terminating audio cues or messages) to signal when the user is in help mode, rather than in the regular application.	Audio formatting is not required, because help is integrated into each application dialog.
Good for providing general help information at an application, module, form, or menu level.	Uses a sequence of prompts to provide progressively greater levels of context-sensitive assistance at each turn in a dialog.

Help Mode	Self-Revealing Help
Requires less help text, but generally provides less-specific information.	Requires you to compose multiple prompts for each turn in a dialog.
Must be triggered explicitly, by the user speaking something from the Help grammar.	Can be triggered: <ul style="list-style-type: none">• Explicitly, by the user speaking something from the Help grammar.• Implicitly, by the user speaking an out-of-grammar utterance (a nomatch event).• Implicitly, by the user failing to respond to a prompt before a timeout occurs (a noinput event).

Because the introduction of modes into a system complicates the interface, we generally recommend the self-revealing help technique. Alternately, the requirements of your application or users may lead you to adopt a strategy that incorporates both styles of help: a **<catch>** element in the application root document, plus context-sensitive self-revealing help where appropriate. Regardless of the help style you choose, you should use it consistently throughout your application.

Maximizing the Benefits of Self Revealing Help

It is possible that a user might experience momentary confusion or distraction, which would lead the user to explicitly request help. If the system has been designed to be self-revealing, these explicit requests for help could receive the same treatment as a silence timeout or any out-of-grammar utterance, allowing you to reuse the same code and prompts. This is style of help is referred to as “implicit” because the system never enters an explicit help mode. The theory is that implicit help provides more of a sense of moving forward (resulting in a better, more satisfying user interface) and is simpler to code than explicit help.

Implementing Self-Revealing Help

The following example assumes a full-duplex system with barge-in:

```
System:           Welcome to the IBM automated directory dialer. You
                  can call any IBM employee by speaking the employee's
                  name and location. You can hang up or say "Help" at
                  any time.

User (interrupting): Help. <or any out-of-grammar utterance>

System: (continues, Please state the desired name and location.
playing explicit
prompt)

User:             I need more help. <or silence timeout or any
                  out-of-grammar utterance>

System:           For example, to call Joe Smith in Kansas City, say,
                  "Joe Smith in Kansas City".
```

Notice the self-revealing nature of the dialog, and the way it works whether triggered by a silence timeout, an utterance from the Help grammar, or any out-of-grammar utterance. In addition, if this application is able to detect when the user requests an unsupported name or city, it can respond appropriately (for example, "Sorry, no office in that location").

Also note the pattern for the introduction, which has the general sequence Welcome-Purpose-MoreInfo, followed by an ExplicitPrompt. Usually, the explicit prompt at the end of the introduction is enough to guide the user to provide an appropriate response; however, if the user input after that explicit prompt is an out-of-grammar utterance, something from the Help grammar, or a silence timeout, then the system falls through a sequence of prompts designed to provide progressively greater assistance. The beauty of this system is that the user never hears the same prompt twice, so the system appears to be responsive to the user and always moving the dialog forward; this help style promotes efficient use by most users. If the user still fails to provide an appropriate response, the sequence of prompts ultimately "*bails out*" (typically, exits).

“Bailing Out”

Deciding how many attempts to allow the user before bailing out is a judgement call. You do not want to bail out after the first unsuccessful attempt, or you will lose users too quickly; conversely, you do not want to require too many attempts, or you run the risk of frustrating users who are experiencing serious problems with the system. For most applications, if users are experiencing serious problems with the system, it is better to get them out of the system quickly and, if possible, give them another means to access the desired information (say, by pointing them to a visual Web site).

In general, two or three progressively more directed prompts should suffice to help users recover from **noinput** events, **nomatch** events, or explicit requests for help. For example:

System:	Transfer \$500 from checking to savings?
User:	<noinput>
System:	Please say Yes or No.
User:	<nomatch or noinput>
System:	If you want to transfer \$500 from checking to savings, say Yes. Otherwise, say No.
User:	<nomatch or noinput>
System:	To authorize the transaction, please say Yes or No. To start over, say Start Over. To hear a prompt again, say Repeat.

For Yes/No questions only, if there is a **nomatch** event in response to the initial prompt, you can attempt a quick recovery by implementing a confirmation subdialog before cycling through the standard help sequence of prompts. See [“Recovering From a nomatch Event” on page 160](#).

Mentioning Always-Active Commands

One potential drawback to this scheme is that it does not necessarily reveal to the user the existence of any always-active commands (see [“Adopting a Consistent Set of Always-Active Navigation](#)

Commands” on page 125). If deemed necessary for a particular application, the system’s dialog design could reveal these features just before bailing out. For example:

System:	For example, to call Joe Smith in Kansas City, say, “Joe Smith in Kansas City”.
User:	I still don’t get it. <or silence timeout or any out-of-grammar utterance>
System:	Would you like to hear a list of the things you can say at any time?
User:	Yes.
System:	The following commands are always available: Help Repeat Backup Start over Quiet Exit Please say one of these options, or state the desired name and location.

Tracking the Reason for Bail Out

If you want the final prompt to indicate *why* you are bailing out, you may want to keep track of the return codes that come back as the user progresses through the help-to-bailout sequence. For example:

Return Code	Bailout Prompt
In Help grammar	“No additional help is available. Try looking up the employee in the Employee Locator database.”
Out of grammar	“Sorry for the confusion. Try looking up the employee in the Employee Locator database.”
Silence timeout	“Experiencing bad connection. Please call back later.”

Note: The only time that the bailout prompt for silence timeout is appropriate is if *all* of the opportunities for user speech have resulted in silence timeout (as might happen with a bad connection on a cellular telephone).

Getting Specific – Low-Level Design Decisions

Once you've decided on the high-level properties of your system, it's time to consider some of the low-level issues, especially regarding specific interaction styles and prompts. This section addresses the following issues:

- ["Adopting a Consistent 'Sound and Feel'"](#) — See [page 136](#).
- ["Using Consistent Timing"](#) — See [page 137](#).
- ["Designing Consistent Dialogs"](#) — See [page 139](#).
- ["Creating Introductions"](#) — See [page 140](#).
- ["Constructing Appropriate Menus and Prompts"](#) — See [page 142](#).
- ["Designing and Using Grammars"](#) — See [page 155](#).
- ["Providing Consistent Error Recovery"](#) — See [page 162](#).

Adopting a Consistent 'Sound and Feel'

To promote a consistent sound and feel, you may want to adopt the following guidelines.

Designing Prompts

When designing prompts, you should choose one prompt style and use it consistently throughout your application. (See ["Adopting a Terse or Personal Prompt Style"](#) on [page 121](#).)

In general, you should try to use the present tense and active voice for all prompts. For example, use:

```
System:      Transfer how much? [!!!!]
```

rather than:

```
System:      Amount to be transferred? [!!!!]
```

Whenever possible, try to use a parallel structure (all nouns, all gerunds, etc.) in lists. For example:

```
System:      Say one of the following:
              Activate my credit card
              Change my address
              Show my credit limit[!!!!]
```

rather than:

```
System:      Say one of the following:
              Activate my credit card
              Change of address
              Credit limit[!!!!]
```

Unless there is a clear design goal to the contrary, you should adopt a consistent voice for your application, whether your prompts are recorded or synthesized. (See [“Creating Recorded Prompts” on page 114.](#))

Standardizing Valid User Responses

Users master applications more quickly when they can predict what responses are valid. For example, if you allow affirmative responses of ‘Yes’, ‘Okay’, and ‘True’ to one yes/no question, you should allow the same responses for all yes/no questions in your application. You can accomplish this by using the built-in types or reusing your own grammars. (See [“Reusing Dialog Components” on page 140.](#))

Using Consistent Timing

Consistent timing is important in developing a conversational rhythm for the dialog. Activities you may want to consider include:

- The amount of user “think time”
- The length of pauses between menu items
- The amount of time the system takes to respond to a user utterance

Setting the Default Timeout Value

The default timeout value for the VoiceXML browser is 7 seconds. For applications designed for special populations (such as non-native speakers or older adults), you may want to increase this by specifying a larger value (around 10 seconds) using the **-Dvxml.timeout** Java system property when you start the VoiceXML browser, or by specifying a value for the **timeout** property in your application root document.

Whatever timeout interval you choose, use it consistently . . . with the possible exception of a very short silence timeout after the initial prompt to identify expert users. (See [“Welcome” on page 140](#) for details.)

Managing Processing Time

Some users may interpret a long processing delay as an indication that the system did not hear or did not accept the most recent input. This can lead to the stuttering effect (see [“Controlling Lombard Speech and the Stuttering Effect” on page 112](#)), or can prevent the user from hearing a system response that is spoken while the user repeats the input.

If you anticipate that processing a user’s request may take an extended period of time, you should consider playing a prompt to inform the user of the delay, or at least to confirm that the system accepted the input. For example:

```
System:      This may take a few minutes. Please wait.
```

or:

```
System:      Processing your request.
```

or simply:

```
System:      Okay.
```

Designing Consistent Dialogs

When users don't know what they can say at a given point in a dialog, the interaction between the user and the application can quickly break down. To help users avoid this “What can I say now?” dilemma, try adopting consistent sound and feel standards to create dialogs that behave consistently, both within your application and across your platform (if you support multiple applications).

Writing Directive Prompts

You should try to write prompts that clearly indicate to users what they can say. For example:

```
System:      Say either Yes or No. [!!!!]
```

or:

```
System:      Travel to which US state? [!!!!]
```

or:

```
System:      Please say one of the following:  
              E-mail  
              Voice mail  
              Faxes [!!!!]
```

Maintaining a Consistent Sound and Feel

When an application behaves in a way that the user didn't anticipate, the user can become confused and/or frustrated and is more likely to respond inappropriately to subsequent prompts. This, in turn, causes the interaction between the user and the application to break down. To help users avoid this “But why did the application say that?” dilemma, try adopting consistent sound and feel standards (see [“Adopting a Consistent ‘Sound and Feel’” on page 136](#)) to create dialogs that maintain synchronization between the application's actual state and the user's mental model of the application's behavior.

Reusing Dialog Components

You can improve application consistency and decrease user learning curves by reusing dialogs or dialog components where possible. Components you might reuse include:

- Subdialogs for data collection, error recovery, and other common tasks
- Built-in field types
- Application-specific grammars

Creating Introductions

The introductory message(s) that the user hears when starting the application can be very important. In general, introductions should contain the following information:

1. "Welcome"
2. "Purpose" (optional)
3. "Always-Active List" (only two or three key commands)
4. "Speak After Tone" (if using turn-taking tones)
5. "Initial Prompt"

Welcome

The welcome prompt should be short and simple, but also make clear that the user is talking to a machine, not a person. For example:

```
System:      Welcome to the automated currency conversion
              system.
```

To speed up the interaction for expert users, you might want to provide a brief (for example, 1.5 second) recognition window without a beep after this welcome prompt. Expert users will know how to provide valid input; if a silence timeout occurs, you can presume that the user is a novice and proceed to play the remaining introductory prompts.

Note: A similar strategy with much shorter (say, 0.75 second) recognition windows can be used at other logical pausing places during and between prompts, such as at the end of a sentence or after each menu item. These brief pauses will give users the opportunity to talk without feeling that they are “being rude” by interrupting the application.

Purpose

If necessary, state the purpose of the system. For example:

```
System:    This system provides exchange rates between
           American dollars and the major currencies of the
           world.
```

Note: You may not need to include a statement of purpose, if the welcome prompt adequately conveys the system’s purpose. Don’t include a statement of purpose unless it’s necessary, because it makes the introduction longer.

Always-Active List

Next, you’ll probably want to tell the user what commands are always available. If you have a large always-active set of commands, you probably don’t want to list all of them here – just the key ones. For example:

```
System:    The commands ‘Repeat’ and ‘Start over’ are always
           available.
```

Speak After Tone

In general, half-duplex systems should use a beep or other tone to signal the user to speak. In such cases, you should include a prompt like:

```
System:    Speak to the system after the tone.
```

Initial Prompt

Finally, present the user with the first prompt requiring user input. If you are using tones to signal turn-taking, this would be the first application message, followed by a tone. For example:

```
System:    To begin, say either ‘buy’ or ‘sell’. [!!!!]
```

Constructing Appropriate Menus and Prompts

A characteristic of many voice applications is that they have little or no external documentation. Often, these applications must support both novice and expert users. Part of the challenge of designing a good voice application is providing just enough information at just the right time. In general, you don't want to force users to hear more than they need to hear, and you don't want to require them to say more than they need to say. Adhering to the following guidelines can help you achieve this:

- "Minimizing Transaction Time" — See [page 142](#).
- "Limiting Menu Length" — See [page 142](#).
- "Grouping Menu Items, Prompts, and Other Information" — See [page 143](#).
- "Minimizing Prompt Length" — See [page 146](#).
- "Avoiding DTMF-style Prompts" — See [page 146](#).
- "Choosing the Right Words" — See [page 147](#).
- "Tapering Prompts" — See [page 152](#).
- "Confirming User Input" — See [page 152](#).
- "Providing Instructional Information" — See [page 153](#).

Minimizing Transaction Time

If possible, try to limit the time it takes a user to complete a typical transaction to under 1 minute.

Limiting Menu Length

When designing menus, you will want to limit the number of menu items based on how much information users must remember, as shown in [Table 29 on page 143](#).

Table 29. Recommended Maximum Number of Menu Items

Application	Maximum Number of Menu Items
<i>Full-duplex implementations with barge-in</i>	9
<i>Full-duplex implementations with barge-in disabled</i>	5
<i>Half-duplex implementations</i>	5
<i>Any implementation in which the menu items are long phrases</i>	3

For cases in which you cannot stay within these limits, see [“Managing Audio Lists” on page 171](#).

Grouping Menu Items, Prompts, and Other Information

There are a number of strategies that you should consider when deciding how to group information.

Separating Introductory/Instructive Text from Prompt Text

When appropriate, you should consider separating any introductory or instructive text from the prompt text; this allows you to reprompt without repeating the introductory text.

For example, you might create one audio file that says, “Welcome to the WebVoice demo” and a separate audio file that says, “Say one of the following options: Library, Banking, Calendar.” The first time through the sequence, the application could play the files in succession. If the user returns to this main menu later in the application session, the application could play only the second audio file. For example, the first time the user would hear:

```
System:  Welcome to the WebVoice demo. Say one of the
         following options:
         Library
         Banking
         Calendar [!!!]
```

On the second and any subsequent times, the user would only hear:

```
System: Say one of the following options:  
        Library  
        Banking  
        Calendar [!!!!]
```

Conversely, if the introduction and prompt are likely to be replayed together, you may elect to combine them into a single audio file.

Separating Text for each Menu Item

If appropriate for your application design, you might even create separate audio files for each menu choice.

Ordering Menu Items

When presenting menu items, consider putting the most common choices first so that most users don't have to listen to the remaining options. For example:

```
System: Please choose:  
        List specials  
        Place an order  
        Check order status  
        Get mailing address [!!!!]
```

A possible exception to this guideline is when the most common choice is a more specific case of another choice. In this example, "Other loans" is presented last, regardless of its relative frequency of use:

```
System: Loan type?  
        Car  
        Personal  
        Other loans [!!!!]
```

Cycling Through Menu Items

If many users will access all items in your menu during a single session, you may want to consider adopting a strategy that allows the user to cycle through the options without first having to return to the main menu. While you could accomplish this by placing the menu options in an always-active grammar, a better design would be to do something like this:

```
System:  Welcome to the Web Trip Planner. Say one of the following
         options:
         Airline
         Hotel
         Rental Car
         Travel Advisory
         Tourist Information
         Exit
User:    Airline
         <dialog for making airline reservations>
System:  Your flights are confirmed. Choose one:
         Hotel <this is the next option from the Main Menu>
         Main Menu
         Exit
User:    Hotel
         <dialog for making hotel reservations>
System:  Your hotel is confirmed. Choose one:
         Rental Car <this is the next option from the Main Menu>
         Main Menu
         Exit
User:    Rental Car
         <dialog for making rental car reservations>
         <etc.>
```

Minimizing Prompt Length

Both half- and full-duplex systems work best when system prompts are as short as possible (minimizing users' need to interrupt prompts) and user responses are relatively short (minimizing the likelihood of Lombard speech and the “stuttering effect” in full-duplex systems — see [“Controlling Lombard Speech and the Stuttering Effect” on page 112](#)).

Note: Especially for full-duplex interfaces using recognition barge-in detection, try to keep required user responses to no more than two or three syllables. If this is not possible, you may want to consider using energy-based barge-in or making the system half-duplex.

Effectively worded shorter prompts are generally better than longer prompts, due to the time-bound nature of speech interfaces and the limitations of users' short-term memory. A reasonable goal might be to strive for prompt lengths of no more than 3 seconds, and to try to keep the greeting and opening menu to less than 20 seconds. (For planning purposes, assume that each syllable in a prompt or message lasts 150-200 ms.) If prompt lengths must consistently exceed 3 seconds, you should consider making the system full-duplex.

Avoiding DTMF-style Prompts

Prompts in an application with a DTMF interface typically take the form “For option, do action.” With a speech user interface, the option *is* the action. In general, you should avoid prompts that mimic DTMF-style prompts; these types of prompts are longer than they need to be for most types of menu selections. For example, use:

System:	Please say one of the following departments:
	Marketing
	Finance
	Human Resources
	Accounting
	Research

rather than:

```
System:      For the Marketing department, say 1
              For Finance, say 2
              For Human Resources, say 3
              For Accounting, say 4
              For Research, say 5
```

If the menu items are difficult for a user to remember (for example, if they are long or contain unusual terms or acronyms), you might choose to mimic DTMF prompts. Alternately, you could encourage users to take advantage of the VoiceXML browser’s “Say What You Hear” feature (see [page 82](#)) by speaking only part of a long menu item; however, you would probably need to convey this information before presenting the menu, or as part of self-revealing help for this prompt.

Choosing the Right Words

There are many aspects to consider when deciding how to word your application’s prompts and menus. The choices you make will have a significant impact on the types of responses your users provide, and therefore on what you will need to code in your grammars. Some of the issues you may need to address include the following:

- “Adopting User Vocabulary” — See [page 147](#).
- “Differentiating Between Data Prompts and Verbatim Prompts” — See [page 148](#).
- “Writing DTMF Prompts” — See [page 149](#).
- “Being Concise” — See [page 149](#).
- “Mixing Menu Choices and Form Data In a Single List” — See [page 150](#).
- “Avoiding Synonyms in Prompts” — See [page 150](#).
- “Promoting Valid User Input” — See [page 150](#).
- “Avoiding Spelling Input” — See [page 151](#).

Adopting User Vocabulary

Applications that require users to learn new commands are inherently more difficult to use. During the Design phase of your application development process, you will want to make note of the words and phrases that your users typically use to describe common tasks and items. See “[Design Phase](#)” on [page 100](#). These are the words and phrases that you will want to incorporate into your prompts and grammars.

Differentiating Between Data Prompts and Verbatim Prompts

Consistency helps users learn your interface easier and faster. You may want to use one word to introduce *data prompts* (where the user must supply information to fill in the field of a form) and a different word for *verbatim prompts* (menu choices that the user can select by repeating what the system say). For instance, you might use “State” for data prompts and “Say” for initial verbatim prompts, shortening the verbatim prompts to “Choose” on subsequent presentation:

```
System:      State your name. [!!!]
User:        John Smith
System:      Say one of these choices:
              E-mail
              Voice mail
              Faxes [!!!]
User:        E-mail
              :
              :
              :
              <later in the dialog>
System:      Choose one:
              E-mail
              Voice mail
              Faxes [!!!]
User:        E-mail
```

Note: An exception to this occurs when you ask the user to provide the name of a state, such as Florida or Wyoming. Following this guideline would result in the clumsy prompt, “State the state.” One alternative to use in this situation is “Speak the state”, but a better approach might be to phrase the prompt in the form of a question, such as “Which state?”

Writing DTMF Prompts

Similarly, for DTMF input you may want to use one word to introduce prompts for a single DTMF digit or command sequence and a different word to prompt for a string of digits. For instance, you might use “Press” to indicate that the user should input a single DTMF digit or command, and “Enter” to indicate that the user should input a string of DTMF digits. For example:

```
System:   Press *h for Help. [!!!!]
```

but:

```
System:   Use your telephone keypad to enter your Social  
          Security number. [!!!!]
```

Being Concise

Try to phrase prompts in a way that conveys the maximum amount of information in the minimum amount of time. For example, use:

```
System:   State the author's last name, followed optionally  
          by the author's first name. [!!!!]
```

rather than:

```
System:   State the author's last name. If you also know the  
          author's first name, state the author's last name  
          and first name. [!!!!]
```

Sometimes, the most effective way to prompt the user is to word the prompt as a question. For example, use:

```
System:   Savings or checking? [!!!!]
```

rather than:

```
System:   Please choose from the inquiry menu:  
          Savings  
          Checking [!!!!]
```

Question prompts are especially useful when the user can choose by repeating one of the options verbatim. You can also use question prompts to collect information, as long as the question restricts likely user input to something the application can understand. For example:

```
System:   Transfer how much? [!!!!]
```

Mixing Menu Choices and Form Data In a Single List

When mixing menu choices and form filling in the same list, you will want to word the prompt to clearly indicate what the user can say. For example, use a prompt like:

```
System:    Please state the author's last name, or say
           List Best Sellers. [!!!!]
```

rather than:

```
System:    Please say the author's last name or List Best
           Sellers. [!!!!]
```

which might elicit a literal response of:

```
User:      Author's last name
```

rather than the actual name of an author.

Avoiding Synonyms in Prompts

You should avoid using synonyms in prompts; these may mislead the user regarding what is valid input, especially given the VoiceXML browser's support for "Say What You Hear" functionality. For example, use:

```
System:    To search the database, say Search by Author.
           [!!!!]
```

rather than:

```
System:    To query the database, say Search by Author.
           [!!!!]
```

because the latter might cause the user to think that "query" is a valid response.

Promoting Valid User Input

The prompt text should guide the user to the proper word choice. For example:

```
System:    If this is correct, say Yes. [!!!!]
```

Since users have a tendency to respond using key words from the prompt, an even better choice might be:

```
System:   Are you sure? [!!!!]
User:     Sure. <active grammar could include Sure, Yes,
          etc.>
```

You should try to phrase prompts in a way that minimizes the likelihood of the user inserting extraneous words in the response. For example, if the system cannot interpret dates embedded in sentences, use:

```
System:   Please state the year you were born. [!!!!]
```

or:

```
System:   Birth year? [!!!!]
```

rather than:

```
System:   When were you born? [!!!!]
```

because the latter is likely to elicit a response such as:

```
User:     I was born on November 3rd, 1954.
```

Avoiding Spelling Input

Whenever possible, you will want to avoid requiring users to spell words. Many letters have high acoustic confusability (for example, B, D, P, and T in the English alphabet), and therefore are prone to be recognized poorly.

Generally, you can design your dialogs to avoid this problem. For example, rather than requiring the user to spell an obscure order number, you could write your prompts to ask for the name of the person placing the order and the date on which the order was placed.

Depending on your application, you may also be able to verify what was spoken against a database of expected responses.

Alternately, if your application is for a narrow set of well-trained users, it may be possible to adopt a word-based “phonetic alphabet” for spelling, as is common in the aviation and military industries (for example, “alpha” for the letter “a”, “bravo” for “b”, “charlie” for “c”, etc.). Users could then be trained to spell using these words, and could possibly be given a wallet-sized card documenting the “phonetic alphabet” to use.

Tapering Prompts

For dialogs that users might revisit in a single session, you may want to create multiple, progressively shorter prompts. After the initial pass through the dialog, you can then play the tapered prompts on subsequent passes. (Users perceive this technique as more helpful than repeatedly playing the same prompt.) For example, the initial prompt might be:

System:	Please state the employee's name and serial number. [!!!!]
---------	---

shortening on the second pass to:

System:	Employee name and serial number? [!!!!]
---------	---

and perhaps eventually to:

System:	Name and number? [!!!!]
---------	-------------------------

Note that this is similar, though not identical to the technique of self-revealing help. One difference is that the self-revealing help prompts tend to get longer (more instructive) as the user moves through the prompting sequence.

Because designing good tapered prompts requires intimate knowledge of how the users interact with the application, you should typically leave this step to the second iteration of the user interface design (see [“Refinement Phase” on page 106](#)).

Confirming User Input

Confirming user input can be time-consuming and may even interrupt the natural flow of the interaction. In general, you should use confirmations judiciously, such as when updating user information in a back-end database, or when performing actions cannot be easily undone.

Before providing lengthy confirmation feedback, you may want to ask users if they want to hear this confirmation. For example:

```
System:    Review entries before submitting? [!!!!]
```

This is especially important for half-duplex systems (since users won't be able to interrupt the lengthy output by barging-in), but is also recommended for full-duplex systems.

Providing Instructional Information

In certain situations, you may need to provide instructional information to the users.

Managing Variable-length DTMF Input

For DTMF input of variable length, you should tell users to use the # key to signal the end of DTMF input. For example:

```
System:    Use your telephone keypad to enter the desired
           amount, followed by the # key.
```

Note: For fixed length DTMF input, the system should not require the # key, and should be able to ignore it if the user does happen to press it.

“Feeding-forward” Information as Confirmation

Where applicable, you may want to word prompts in a way that “feeds the result forward” (that is, incorporates the user response) into the next prompt. For example:

```
System: Please say one of the following options:
        Phone
        Fax [!!!!]
User:   Phone.
System: Which phone action?
        Leave message
        Camp
        Forward call [!!!!]
```

This technique provides feedback that the system correctly understood the response and also reinforces the user's mental model of the dialog state. Using this technique eliminates the need for cumbersome confirmation of every user input; however, you should still confirm user requests for actions that cannot be undone.

Recovering from Errors

This section deals with error recovery only as it relates to prompt wording. See [“Providing Consistent Error Recovery” on page 162](#) for additional information on error recovery.

To promote faster error recovery, prompts should focus on keeping the dialog moving rather than on any mistakes. For example, if you are using self-revealing help, you should avoid having the system say:

```
System:      Sorry, I don't understand what you said.
```

because this does not help the user understand what to do next. See [“Implementing Self-Revealing Help” on page 133](#) for guidance on how to keep the dialog moving forward with self-revealing help.

Similarly, it is best to avoid claiming that the user “said” a particular response, since the information you present is actually just a reflection of how the speech recognition performed. For example, use:

```
System:      123457. If this is the correct account number,  
             say Yes. [!!!!]
```

instead of:

```
System:      You said 123457. This is not a valid account  
             number. Please restate your account number.  
             [!!!!]
```

because it is possible that the user actually said 1234567, but the recognition engine did not register the number 6. (Of course, it is also possible that the user actually said the number incorrectly.) Note that the example above is only valid if 6-digit account numbers are legal in the system; if the system required seven digits, then an appropriate reprompt might be:

```
System:      123457. Please restate your 7-digit account  
             number. [!!!!]
```

In this example, the system feeds back the number it interpreted without claiming the user said it, provides the information about the seven-digit requirement, and requests the user to provide the number again.

Designing and Using Grammars

Designing good grammars is an art, not a science. Iterative prototyping is crucial to grammar design. See [“Design Methodology” on page 100](#).

Since only words, phrases, and DTMF key sequences from active grammars are considered as possible speech recognition candidates, what you choose to put in a grammar and when you choose to make each grammar active have a major impact on speech recognition accuracy. In general, you should only enable a grammar when it is appropriate for a user to say something matching that grammar. At all other times, you should disable the grammar to improve recognition response time and recognition accuracy for other grammars. When appropriate, you should reuse grammars to promote application consistency.

Managing Tradeoffs

There are many tradeoffs that you will want to consider in deciding what words and phrases to include in your grammars and when to make each grammar active. Some of the major tradeoffs are:

- [“Word and Phrase Length”](#)
- [“Vocabulary Robustness and Grammar Complexity”](#)
- [“Number of Active Grammars”](#)

Word and Phrase Length

One of the first tradeoffs you are likely to encounter is how long users responses should be. [Table 30 on page 156](#) compares the two schemes.

Table 30. Grammar Word/Phrase Length Tradeoffs

Longer Words and Phrases	Shorter Words and Phrases
<p>Multisyllabic words and phrases generally have greater recognition accuracy because there is greater differentiation between valid utterances.</p> <p>Individual word choice is still important in longer phrases of because the VoiceXML browser’s ability to match a menu choice based on a user utterance of one or more significant words. See “Say What You Hear” on page 82.</p>	<p>Shorter words and phrases are more likely to be misrecognized; when a grammar permits many short user utterances, it is important to minimize acoustic confusability by making them as dissimilar sounding as possible.</p> <p>Monosyllabic words and short words with unstressed vowels are especially prone to be recognized as each other, even though they may look and sound different to a human ear.</p>
<p>Dialogs may be slower.</p>	<p>Dialogs progress faster: choices are read faster, and user responses tend to be shorter.</p>
<p>Users may have difficulty remembering long phrases.</p>	<p>Easier for users to remember.</p>
<p>For full-duplex implementations with recognition barge-in detection, longer words and phrases may induce stuttering and Lombard effects. See “Choosing Full-Duplex (Barge-In) or Half-Duplex Implementation” on page 109.</p>	

Vocabulary Robustness and Grammar Complexity

A related issue is how robust and complex your grammars should be, as illustrated in [Table 31](#).

Table 31. Vocabulary Robustness and Grammar Complexity Tradeoffs

Robust Grammar	Simple Grammar
<p>Inclusion of synonyms and alternative phrases gives users greater freedom of word choice; however, users may incorrectly assume that they can say virtually <i>anything</i>, leading to a large number of out-of-grammar errors.</p>	<p>Narrow list of valid utterances places more constraints on user input.</p>
<p>Grammar files are larger and load slower.</p>	<p>Grammar files are smaller and load more quickly.</p>
<p>Increased chance of recognition errors.</p>	<p>Simple grammars generally have better recognition accuracy.</p>

Number of Active Grammars

Finally, you will want to consider when each grammar should be active, as presented in [Table 32](#).

Table 32. Number of Active Grammars Tradeoffs

More Active Grammars	Fewer Active Grammars
May improve usability, such as by allowing anytime access to items on main menu.	
Increased chance of recognition conflicts.	Less chance of misrecognitions due to recognition conflicts.
Performance can degrade.	Better performance.

Note: You can limit the active grammars to just the ones specified by the current form by using the `<field>` element's modal attribute.

Improving Recognition Accuracy

In general, you can improve recognition accuracy by:

- Simplifying your grammars to minimize the possibility of confusion between words.
- Presenting fewer choices.
- Having fewer active grammars.

For transactions that cannot be undone, you may also want to add confirmation prompts to verify that the user input was accurately recognized.

Matching Partial Phrases

With the VoiceXML browser's "Say What You Hear" algorithm, users can select from a menu choice or option list by speaking the *exact, complete* text for the desired item, or by saying a sequence of one or more words from the choice or option text, as explained in ["Say What You Hear"](#) on page 82.

When designing your grammars, you should bear in mind that if multiple items contain the same key words, users may inadvertently trigger the wrong item when they speak partial text from a choice or option. Consider a menu with the following choices:

```
System: Say one of the following:
        Choose another movie in this theater
        Choose another theater
```

If the user says “Choose another theater”, the VoiceXML browser will match the utterance to the first conforming item, which in this case would be “Choose another movie in this theater.” (“Choose”, “another”, and “theater” are significant words in both menu choices.)

There are two ways to resolve this problem:

- Reverse the order of the items in the menu (so that “Choose another theater” comes first)
- Reword the text of one of the items (for example, change the first item to “Choose another movie”)

Improving Grammar Performance

Grammar design can have a significant impact on response time. The best advice is to thoroughly test your grammars; however, we can offer the following high-level guidelines:

- The longer the user utterance, the longer the response time.
- Increasing the rule depth (that is, using subrules) increases response time.
- Using the repeat operator (*) gives slightly better performance than explicitly calling a subrule multiple times.

[Table 33 on page 159](#) compares the relative performance of various grammar designs:

Table 33. Grammar Design and Response Time

Utterance	Grammar Design (in order of increasing response time by utterance)
<i>one</i>	public <rule> = one;
	public <rule> = <sub>*; <sub> = one;
	public <rule> = <sub>; <sub> = one;
	public <rule> = <sub>; <sub> = <sub2>; <sub2> = one;
	public <rule> = <sub>; <sub> = <sub2>; <sub2> = <sub3>; <sub3> = one;
<i>one one</i>	public <rule> = <sub>*; <sub> = one;
	public <rule> = <sub><sub>; <sub> = one;
<i>one two three four</i>	public <rule> = one two three four;
	public <rule> = <sub1><sub2>; <sub1> = one two; <sub2> = three four;
	public <rule> = <sub1><sub2>; <sub1> = <sub1a><sub1b>; <sub1a> = one; <sub1b> = two; <sub2> = <sub2a><sub2b>; <sub2a> = three; <sub2b> = four;
<i>one one one one one one one one</i>	public <rule> = <sub>*; <sub> = one;
	public <rule> = <sub><sub><sub><sub><sub><sub><sub><sub>; <sub> = one;

Using Boolean and Yes/No Grammars

General Strategy

For the first presentation of a prompt with an expected answer of Yes or No, we generally recommend using the built-in **boolean** grammar. This grammar provides more flexibility in accepting user input than a simple Yes/No grammar would. For example:

```
System:    Do you want more information? <boolean grammar active>
User:      Okay
```

Recovering From a noinput Event

If the system returns a **noinput** event in response to the initial prompt, we recommend that you attempt to recover by switching to a simple Yes/No grammar and a prompt that clearly directs the user to say “Yes” or “No”, as shown here:

```
System:    Do you want more information? <boolean grammar active>
User:      <no response>
System:    Please say Yes or No. <Yes/No grammar active>
User:      Yes.
```

Recovering From a nomatch Event

If the system returns a **nomatch** event based on the user input for the initial prompt, we recommend that you switch to a Yes/No grammar and use a prompt that attempts to confirm whether the user intended to provide a positive or negative response. In their book, *How to Build a Speech Recognition Application — A Style Guide for Telephony Dialogues* (Enterprise Integration Group, San Ramon, CA, 1999), Bruce Balentine and David P. Morgan recommend the phrase, “Was that a Yes?” for this purpose. For example:

```
System:    Do you want more information? <boolean grammar active>
User:      <unintelligible response>
System:    Was that a Yes? <Yes/No grammar active>
User:      Yes.
```

This design minimizes disruptions to the dialog flow because the user's response to the subdialog prompt is the same as the intended response to the prompt that generated the out-of-grammar error.

Note: For consistency's sake, you could make this a subdialog and reuse it for all confirmations.

When Initial Accuracy Is Paramount

If it is more important to get extremely high accuracy on the first presentation of the Yes/No question than it is to accept a broader range of user responses, you could write the initial prompt so that it explicitly directs the user to say Yes or No, and use a Yes/No grammar. For example:

```
System:   Approve transaction? Please say Yes or No. <Yes/No grammar
          active>
User:     Yes.
```

Using the Built-in Phone Grammar

Some people tend to pause at the logical grouping points when speaking telephone numbers, especially if they are having difficulty remembering the number. For example, when speaking a U.S. telephone number, a user might pause after the 3-digit area code, and again after the 3-digit exchange. If the pauses are long enough, they might inadvertently trigger endpoint detection before the user has finished speaking the 10-digit telephone number.

If your application requires users to enter telephone numbers, you will want to take special care to thoroughly test your telephone number collection dialogs; if users experience difficulties, you may want to employ some of the following techniques to ensure that the data is being captured correctly:

- Set the **completetimeout** and **incompletetimeout** properties to a higher than default value (for example, 1150 to 1500 msec), to allow users more time to pause while speaking the telephone number. If you adopt this strategy, you should also test that the latency until the next system prompt does not become too great.
- If the number of digits collected on the initial attempt indicates that the user probably paused too long at one of the logical grouping points, you might try collecting just the remaining digits. For example, if the recognition engine returned only 6 digits, it may be that the user paused too long after speaking the area code plus exchange. In this case, you might prompt only for the final four digits of a 10-digit U.S. telephone number.

- Design your application to require a fixed number of digits (for example, always requiring the area code, even for local telephone numbers).
- Allow or encourage users to use DTMF when entering telephone numbers.
- Have the application read back telephone numbers for user verification.

Testing Grammars

When testing your grammars, you should test words and phrases that are in your grammars, as well as words and phrases that are not. (The purpose of testing “out-of-grammar” words and phrases is to ensure that the speech recognition engine is rejecting these utterances; erroneously accepting these utterances could cause unintended dialog transitions to occur.)

If your application has more than one grammar active concurrently, you should test each grammar separately, and then test them together.

To help identify if there are any words that are consistently misrecognized, you should test your grammar with a group of test subjects that is representative of the demographics and environments of your users. For example, you might want to vary the ambient noise level, gender, age, accent, and level of fluency during desktop testing. When you are ready to deploy your application, you may want to perform additional testing while varying the type of telephone (standard, cordless, cellular, and speaker phone, etc.).

If you discover words or phrases that are consistently problematic, you might need to rephrase some entries.

Remember that testing your grammar is an iterative process. As you make changes, you should go back and retest to verify that all of the valid words and phrases can still be recognized.

Providing Consistent Error Recovery

Error recovery may be even more important for a speech user interface than for a graphical user interface because:

- Speech is transient
- Voice applications generally have little or no external documentation

Managing Errors

Users are less likely to get lost in broad (as opposed to deep) dialogue structures; therefore, try to avoid deeply-nested menu structures.

When designing your application, you should provide consistent opportunities to recover from the types of errors shown in [Table 34](#). Subdialogs are one way to do so.

Table 34. Error Recovery Techniques

Error	Example	Recommended Strategy
<i>System misrecognition of user input.</i>	The number of PIN digits input or recognized does not match the number required.	Reprompt.
<i>System misinterpretation of user intent.</i>	A misrecognition or user error has caused the dialog to progress in a way that is inconsistent with the user's intention.	Feed user input forward to give users feedback (see “Providing Instructional Information” on page 153) and provide a Backup command (see “Adopting a Consistent Set of Always-Active Navigation Commands” on page 125).
<i>User failure to grasp all of the presented information.</i>	The user was momentarily distracted and did not hear all of the menu choices.	Use the introductory prompt to inform the user about the always-active Repeat command. Use self-revealing help to repeat menu choices and to remind the user about the always-active commands, including the Repeat command.

Recovering from Out-of-Grammar Utterances

You might also want to consider writing a subdialog to recover from an out-of-grammar user response to a boolean prompt, as explained in [“Recovering From a nomatch Event” on page 160](#).

If out-of-grammar responses persist, the audio input quality may not be sufficient for recognition to occur. To counter this problem, you could try prompting the user to provide DTMF input.

Confirming User Input

You should always ask for user confirmation if the system cannot undo an action. For example:

```
System:      Transferring $500 from checking to savings. Say
              'Yes' or 'No'. [!!!!]
```

Compound questions pose a tradeoff. For example, asking “Are the name and address correct?” is quicker to answer if the answer is yes for both, but is more confusing for the user and requires an extra dialog to separate out the incorrect answer(s) if the answer for one or more is no. For example:

```
System:      Are the name and address correct? [!!!!]
User:        No.
System:      Which is incorrect: Name, street address, city, state, zip
              code? [!!!!]
User:        Zip code.
System:      Please state the zip code. [!!!!]
```

Understanding Spoke-Too-Soon and Spoke-Way-Too-Soon Errors

In half-duplex systems, the user might cause an error by speaking before hearing the tone that indicates the system is ready for recognition. The user then continues speaking over the tone and into the recognition window. This is called a “*spoke-too-soon*” (STS) error, and is illustrated in [Figure 4 on page 165](#). Because a portion of the user utterance was spoken outside of the recognition window, the input that the speech recognition engine actually received often does not match anything in the active grammars, so the speech recognition engine treats it as an out-of-grammar utterance.

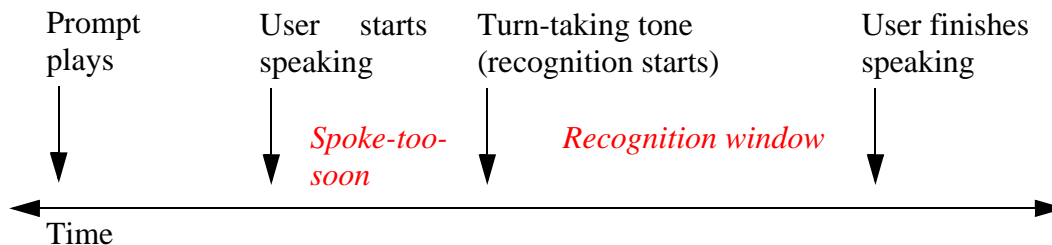


Figure 4. Spoke-Too-Soon (STS) Error

It is also possible for a user to *finish* speaking before the tone sounds; this is called a “*spoke-way-too-soon*” (SWTS) error, and is illustrated in [Figure 5](#). Because the entire user utterance was spoken outside of the recognition window, the speech recognition engine does not actually receive any input and the system will generally time out waiting for the input that the user already gave.

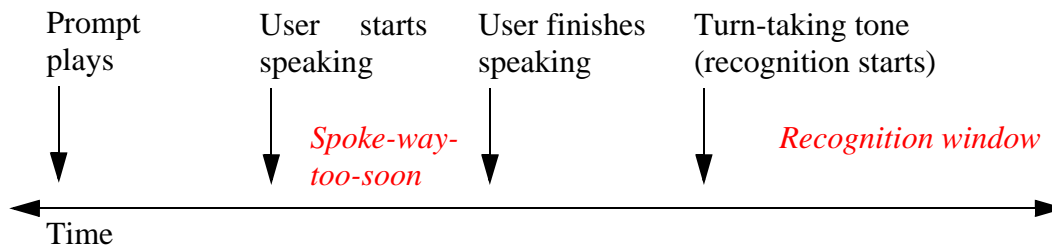


Figure 5. Spoke-Way-Too-Soon (SWTS) Error

Minimizing STS and SWTS Errors

Spoke-too-soon and spoke-way-too-soon errors can be a common source of speech recognition errors, especially in a half-duplex system with excessively long prompts. Here are some things you can do to minimize these types of errors:

- Make your system prompts as short as possible, especially if your application is half-duplex.
- Aggressively trim silence from the end of prerecorded audio files to prevent time lapses between the end of the prompt and the presentation of the beep.

- Place the “Please speak to the system after the beep” prompt at the end of the introductory message, just before you request the initial input from the user.
- If the user commits a spoke-too-soon error, play a message instructing the user to “Please speak to the system after the beep.”
- For spoke-way-too-soon errors, add the instruction “Speak to the system after the beep” to self-revealing help prompts that are triggered by a silence timeout.

Recovering from STS and SWTS Errors

These types of errors can become more frequent as users become expert with a system and rush to provide input. A spoke-to-soon or spoke-way-too-soon error almost never indicates a problem with the user *understanding* the system; therefore, the appropriate system response is to let the user get back to the application as quickly as possible, with a reminder to wait for the tone. For example:

```
System:      Do you want to do another transaction? [!!!!]
User:        <interrupting the beep> Yes.
System:      Remember to wait for the tone. Do you want to do another
              transaction? [!!!!]
User:        <speaking completely over the prompt, resulting in silence
              timeout> Yes.
System:      Please say 'yes' or 'no'. [!!!!]
```

Implications for Self-Revealing Help in Half-Duplex Systems

If a half-duplex system returns a **nomatch** or **noinput** event, you can not be sure whether it was truly an out-of-grammar or timeout, or whether it was caused by a spoke-too-soon or spoke-way-too-soon error. Consequently, you should include early in the self-revealing help sequence a reminder to wait for the tone, along with whatever other help you want to provide for these return codes; this “wait for the tone” reminder is not necessary when the user said a phrase from the Help grammar.

Advanced User Interface Topics

The guidelines presented in the previous sections of this chapter cover the fundamentals of creating a clean, usable speech user interface. You don't have to stop there, though, if you have the resources and motivation to create a more advanced user interface. The advanced user interface topics covered in this section are:

- ["Customizing Expertise Levels"](#) — See [page 167](#).
- ["Selecting an Appropriate User Interface Metaphor"](#) — See [page 168](#).
- ["Controlling the "Lost in Space" Problem"](#) — See [page 170](#).
- ["Managing Audio Lists"](#) — See [page 171](#).
- ["Additional Opportunities for Exploiting Audio Formatting"](#) — See [page 173](#).

Customizing Expertise Levels

You may want to consider designing your VoiceXML application to accommodate multiple levels of user expertise. Depending on the application, this may be as simple as switching prompt styles, or as complex as optimizing dialogs for expert users (for example, to allow expert users to fill multiple form fields with a single utterance by using mixed-initiative dialogs). Another useful technique to speed up the interaction for expert users is to provide a brief welcome prompt, followed by a recognition opportunity; if the user does not provide any input, then provide the list of initial choices. This is described in ["Welcome" on page 140](#).

When multiple expertise levels are available, consider allowing the user to dynamically select the desired level. To do this, you might need to add commands to the always-active set of grammars.

Alternatively, you might allow user behavior to determine the desired level. For example, if a user barges in to interrupt the opening prompt with a valid command, you might treat that user as an expert. (A novice is unlikely to know valid input and is therefore unlikely to accidentally trigger this expert detector.) If you do this, you will also need a way to automatically revert back to "novice" level if it becomes apparent (through **nomatch** or **noinput** events) that the user isn't really an expert.

Because customizing expertise levels requires in-depth knowledge of how users interact with the application differently as they progress from novice to expert, this step should typically be left to the second iteration of the user interface design (see ["Refinement Phase" on page 106](#)).

Selecting an Appropriate User Interface Metaphor

Users develop mental models to help them understand system behavior. The effectiveness of the user interface is often a direct function of the extent to which the user's mental model of the system state matches the system's actual state because mismatches can quickly lead to breakdowns in the user's interaction with the system. You can influence the user's mental model by choosing and using an appropriate user interface metaphor.

You will want to select a user interface metaphor that is intuitive for your application, and consistently adhere to this model in all interactions with the user. The correct user interface metaphor for your application is one that allows users to predict the behavior of the interface in a given situation.

Audio Desktop Metaphor

An Audio Desktop metaphor is often appropriate for interfaces that span multiple applications or modules. Users can access specific information objects by speaking a command, much like users of a visual interface can access specific objects by clicking on a visual icon.

Personified Interface Metaphor

A Personified Interface is one in which the designer has deliberately attempted to give it specific personality traits, including apparent self-awareness and decision-making abilities. Building a personified interface and doing it right is a non-trivial effort whose description is beyond the scope of these guidelines. For more information on this topic, you may want to read Chapter 9 in the Balentine and Morgan book.

Conversational Participants Metaphor

A Conversation Participants metaphor is often appropriate for interfaces in which the user can interact with multiple applications in parallel. For example, a user might interact with both a stock management module and a banking module in the same session. You may elect to assign different “agent personalities” (using audio formatting and perhaps even a human name) to the speech output of the two modules. In this case, you could permit the user to interact with the different modules as if carrying on a three-way conversation. For example:

```
User <speaking to a Banking module, male personality named Paul>: Paul, what's my checking account balance?
Paul: Your checking account balance is $65,439.17.
User <speaking to Stock Management module, female personality named Mary>: Mary, buy 100 shares of IBM.
Mary: Buying 100 shares of IBM. Anything else?
User <speaking to Mary>: No.
```

Note: Be cautious when allowing users to specify agent names in speech interaction. In normal human-to-human conversation, people have a natural tendency to pause after speaking a name as an attention-getting device; this pause lets the other participant in the conversation know that the speaker is getting ready to say something to them. If this pause exceeds about 500 ms (and these pauses often do in natural speech), this pause might trigger endpoint detection in the speech recognition engine, which means that the engine might well miss the rest of the user's utterance.

Remember that while humanizing the interface can result in a more pleasant user experience, it can also lead the user to produce very open-ended responses that are difficult for the system to recognize. When the system has trouble recognizing the user, it doesn't take long for the interface to break down. Building this type of interface requires more extensive usability testing than a directed interface.

Controlling the “Lost in Space” Problem

Users rarely get lost when using simple, straightforward applications. When an application contains deeply-nested menus, though, the user can get disoriented, causing the interaction between the user and the application to break down.

Minimizing the Number of Nested Menus

To help users avoid this “lost in space” dilemma, you should try to minimize the number of nested menus required to reach any given dialog state. In speech interfaces (as in visual interfaces), users are less likely to get lost in a broad structure as opposed to a deep structure.

Using Audio Formatting

In addition, you may want to consider using audio formatting to help orient the user. For example, you might play different introductory or background tunes for different modules of the your application.

Providing State Information

Another technique to consider is providing the user with a consistent means for querying the system about the current state of the dialog. Doing this might require you to create additional always-active commands.

Providing Bookmarks

You might also consider providing a bookmarking capability to enable advanced users to return to a particular dialog state using fewer dialog moves. Doing this might require you to create additional always-active commands.

Managing Audio Lists

In general, audio menus should not contain more than a few items; recommended maximums are described in [Table 29 on page 143](#). There might be times when you have no choice but to present a list of more than the recommended number of items. If so, a good way to manage the list may be by using a “speak to select” strategy or list-scanning commands.

Using a “Speak to Select” Strategy

When presented with a running list of items, users (even first-time users) typically say one of a small number of phrases to interrupt the list when they hear the item they want, including utterances such as “OK”, “Stop”, or “Yes”, or simply repeating the target item. For example:

```
System:      When you hear the name of the desired video, please repeat
              it:
              The Maltese Falcon
              Gone with the Wind
              The Nutty Professor
              The War of the Worlds
User:        Yes.
System:      Adding "The War of the Worlds" to your shopping cart.
```

Note: The prompt still suggests that the user repeat the desired movie name (rather than just saying ‘yes’), because this strategy takes some of the time pressure off the user; the user can choose an item even after several additional selections have been read.

Using List-Scanning Commands

For very long lists in full-duplex systems with barge-in, you may want to consider providing list-scanning commands that permit bi-directional navigation, such as BACKWARD, FORWARD, NEXT, PREVIOUS, TOP, and BOTTOM.

Unless you have previously identified that the user has experience with the system, you should plan to announce these list-scanning commands before presenting the list. You may also want to consider using special audio tones as feedback for the recognition some of these commands. For example:

```
User:      List the stocks in my portfolio.
System:    Your portfolio has fifty stocks. You can use the commands
           FORWARD, BACKWARD, NEXT, PREVIOUS, TOP and BOTTOM to control
           the playback of this list.
           General Motors
           IBM
           General Mills
           Hannaford Brothers
           Eastman Kodak
User:      Backward.
System:    <backward tone>
           Eastman Kodak
           Hannaford Brothers
           General Mills
User:      Forward.
System:    <forward tone>
           General Mills
           Hannaford Brothers
User:      Price.
System:    The current price for Hannaford Brothers is 25 7/8, up 1/8.
User:      Next.
System:    The current price for General Mills is 87 3/4, down 1/4.
User:      Bottom.
System:    <bottom tone>
System:    The current price for Wal-Mart Stores Inc. is 54, unchanged.
User:      Backward.
System:    <backward tone>
           Wal-Mart Stores Inc.
```

Kellogg Company
 Monsanto Company
 <etc.>

Additional Opportunities for Exploiting Audio Formatting

In addition to the opportunities described in [“Applying Audio Formatting” on page 116](#), you may want to consider formatting the system events shown in [Table 35](#) with well-designed audio tones, in much the same way that modern GUIs use tones to complement visual feedback for system events.

Table 35. Advanced Audio Formatting

Event	Description
<i>Toggle-on</i>	User turned on a toggle option
<i>Toggle-off</i>	User turned off a toggle option
<i>Ask-question</i>	System just asked the user a yes/no question
<i>Yes-answer</i>	Confirms that the user answered ‘yes’
<i>No-answer</i>	Confirms that the user answered ‘no’
<i>Alert-user</i>	Alert user about a possible warning
<i>Delete-object</i>	System successfully deleted an object
<i>Task-done</i>	Signals completion of a task (similar to the hourglass in a visual interface disappearing)
<i>Save-object</i>	System successfully saved an object
<i>Open-object</i>	System opened an object (file, folder, dialog, etc.)
<i>Close-object</i>	System closed an object (file, dialog box, menu, etc.)
<i>Select-object</i>	Selected an object
<i>Deselect-object</i>	Deselected an object
<i>Paste-object</i>	Pasted an object
<i>Search-hit</i>	Found a match when searching
<i>Search-miss</i>	Search failed (opposite of search hit)
<i>Help</i>	About to present a help message
<i>Large-movement</i>	Cursor jumped (for example, when navigating between links)

This chapter contains hints and tips for structuring, coding, and testing your VoiceXML applications. Topics include:

- ["VoiceXML Application Structure"](#) — See [page 175](#).
- ["VoiceXML Coding Tips"](#) — See [page 177](#).
- ["XML Issues"](#) — See [page 180](#).
- ["Security Issues"](#) — See [page 180](#).
- ["Desktop Testing"](#) — See [page 181](#).

VoiceXML Application Structure

When deciding how best to structure your VoiceXML applications, you may want to consider the issues discussed in this section.

Deciding How to Group Dialogs

Unlike an HTML page, which is a renderable unit, a VoiceXML document may contain several "interactable" units such as forms, menus, etc. While you could go to extremes and either write your entire VoiceXML application in a single VoiceXML document or have a separate document for each menu or form, most VoiceXML applications will consist of dialogs and supporting code grouped together into a number of files. For example:

- An application root document for defining global variables and global grammars.
- One or more VoiceXML documents containing the application dialogs or subdialogs.
- Grammar and prerecorded audio files, as required by the application dialogs.
- ECMAScript files, if any.
- A site customization file to create global `<menu>` and `<link>` elements for your site, if desired. In some ways, you can think of this as being similar to an application root document, but configured by the system administrator rather than the application developer.
- Server-side logic for accessing information in back-end enterprise databases.

There are no firm rules for deciding how to group dialogs; however, careful consideration of the following issues can help you decide:

- Logical grouping of menus or forms
- Resources they require
- Functions they perform
- Expected frequency of use
- Number of pages you want the VoiceXML browser to request from the Web application server

For example, you might decide to use a separate VoiceXML document for a form or menu that is executed infrequently and contains a large grammar or references large grammar or audio files. This design will allow the large files to be downloaded only when needed.

Similarly, you might decompose a complex dialog sequence into a series of subdialogs. Subdialogs are also useful for creating reusable components.

Deciding Where to Define Grammars

Grammars for VoiceXML applications can be defined in an external file or inline, as explained in [“External and Inline Grammars” on page 74](#). The only difference is the level of interaction between the VoiceXML browser and Web application server, and the resulting application performance. You will want to weigh such factors as:

- grammar size, and its affect on document access time
- the importance of instantaneous response, and the corresponding need to load the grammars up-front

For example, you might decide to define a grammar as an external file if the grammar is large and is in a menu or form that is unlikely to be executed. This design will minimize document access time.

Conversely, you might decide to define a grammar inline if you want it to be instantaneously ready (rather than having to download it from the Web application server) when the user accesses the menu or form. Alternately, you might achieve similar results by fetching and caching the grammars when the application loads, as explained next.

Fetching and Caching Resources for Improved Performance

You can improve performance by using a combination of fetching and caching to prime the cache with all of the referenced audio and/or grammar files before your application starts. To do this, create a VoiceXML application root document that contains a form that will never be visited. In this form, specify all of the audio tags that will be referenced by the application; the files should be referenced using a **fetchhint** value of **prefetch** and a caching policy of **safe**. For example:

```
<?xml version="1.0">
<vxml version="1.0">
  <!-- This form should never be visited. It is defined to prefetch -->
  <!-- a large grammar that will be loaded for password verification -->
  <form id="prefetchGrammar">
    <grammar src="password.gram" type="application/x-jsgf"
    fetchhint="prefetch" caching="safe"/>
  </form>
</vxml>
```

When the first document in the VoiceXML application is loaded, the VoiceXML browser will fetch all of the referenced audio and grammar files and store them in the cache.

VoiceXML Coding Tips

This section documents miscellaneous tips on VoiceXML coding.

Accessing User Utterances

If you want to access the actual recognized user utterance (for example, “a buck twenty five”), you can use the shadow variable for the corresponding field: *name\$.utterance*.

If you want to know how the value was stored (“USD1.25” in this example), you can assign the field to another variable. This technique can be very useful when designing your server-side logic.

For example, the following code tests how currency values are stored:

```
<vxml version="1.0">
  <form id="GrammarTest">
    <var name="Gram" />
    <field name="WhatYouSaid" type="currency">
      <prompt>
        Say some currency
      </prompt>
      <filled>
        <assign name="Gram" expr="WhatYouSaid" />
        <prompt> You said: <value expr="WhatYouSaid" />,
                  which was stored as: <value expr="Gram" />.
                  The shadow variable 'utterance' contains:
                  <value expr="WhatYouSaid$.utterance" />.
        </prompt>
        <clear namelist="WhatYouSaid" />
      </filled>
    </field>
  </form>
</vxml>
```

To see how this works, try running this test code using the following values:

```
sixteen dollars and fifty seven cents
three twenty five
nine million two hundred thousand dollars
a buck twenty five
```

Now examine the output in the **vxml.log** file:

```
V: file:C:/testcases/shadowvars/shadow004.vxml (fetch get)
C: Say some currency
H: sixteen dollars and fifty seven cents
C: You said: 16 dollars and 57 cents,
  which was stored as: USD16.57.
  The shadow variable 'utterance' contains: sixteen dollars and fifty
  seven cents.
C: Say some currency
H: three twenty five
C: You said: 3.25,
  which was stored as: 3.25.
  The shadow variable 'utterance' contains: three twenty five.
C: Say some currency
H: nine million two hundred thousand dollars
C: You said: 9200000 dollars,
  which was stored as: USD9200000.00.
  The shadow variable 'utterance' contains: nine million two hundred
  thousand dollars.
C: Say some currency
H: a buck twenty five
C: You said: 1 dollar and 25 cents,
  which was stored as: USD1.25.
  The shadow variable 'utterance' contains: a buck twenty five.
```

Using <value> within <choice>

Although the VoiceXML browser supports the use of a <value> element within a <choice> element (as documented in the VoiceXML 1.0 specification), this construct is not recommended, nor is it particularly useful.

The grammar is created from the **<choice>** element when the document is loaded. Therefore, the variable referenced in the **<value>** element must be defined before the document is loaded (that is, it must have been defined with application scope in a previously loaded document), and its value will not be reevaluated before the grammar is used.

XML Issues

It is important to remember that VoiceXML is an XML-based application, and therefore subject to standard XML rules, such as those explained here.

Using Relational Operators

You must use escape sequences to specify relational operators (such as greater than, less than, etc.). For example, instead of “<” you must use “<” to represent “less than”.

Specifying Character Encoding

The default character encoding for XML documents is utf8, which has 7-bit ASCII as a proper subset. Character codes in VoiceXML documents that are greater than 127 (such as 0x92, which is a fancy apostrophe) will therefore be interpreted as the first byte of a multi-byte utf8 sequence. You can force the XML parser to use another codepage by specifying the desired encoding as the very first thing in the file:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

Security Issues

Most security issues will apply to the deployment environment, not the desktop environment. However, two issues bear mentioning here to enable you to consider them when performing desktop testing.

Authenticating Users

User authentication is a function handled by the individual VoiceXML application, not the VoiceXML browser. For example, a VoiceXML application could prompt the user for a name and PIN code.

If desired, you can write your application to allow the user to *speak* the name, but require the user to use *DTMF* to enter the PIN code. See [“A Simple VoiceXML Example” on page 69](#) for an example of this.

The application can use this data to authenticate the user against an existing database, similar to the process adopted by many secure visual Web sites.

Using a Proxy Server

To access your application files using a proxy server, you will need to edit the `%IBMVS%\bin\vsaudio.bat` file, search for **“:Execute”** to locate the command that starts the VoiceXML browser, and specify the following Java system properties:

-Dproxy.Set=true -Dproxy.Host=your.proxy.host.com -Dproxy.Port=portnumber

Note: These properties are a feature of the Java HTTP implementation, not the IBM WebSphere Voice Server SDK.

When you run the revised batch file, the VoiceXML browser will use the defined proxy server.

Desktop Testing

The IBM WebSphere Voice Server SDK includes features to facilitate desktop testing of VoiceXML applications.

Simulating DTMF (Telephone Key) Input

You can simulate DTMF input on your desktop by using the DTMF Simulator, which is started automatically by the VoiceXML browser unless you specify the Java system property **-Dvxml.gui=false**.

When the DTMF GUI is enabled, you can simulate a telephone keypress event by pressing the corresponding key on the computer keyboard or clicking on the corresponding button on the DTMF Simulator GUI. For more information, see [“Interactions with the DTMF Simulator” on page 39](#).

Running Text Mode or Automated Tests

If you do not have a microphone on your desktop workstation, you can still invoke the VoiceXML browser using the **vstext** script:

```
"%IBMVS%bin\vstext" URL
```

This script starts the VoiceXML browser in text mode. The VoiceXML browser writes prompts and other output as text in the window from which you started the VoiceXML browser, and accepts input from your keyboard or the DTMF Simulator GUI.

For example, you could perform text mode testing of the restaurant application shown in [“Static Grammars” on page 76](#) by typing the following inputs:

```
Coffee  
Turkey and cheese on rye
```

If you want to perform automated testing, you can redirect input from a text file:

```
"%IBMVS%bin\vstext" URL < inputfile
```

Using a Text File for Input

The input file you create should be an ASCII text file containing one user response per line. Each response should, of course, be appropriate for the state that the dialog will be in at that point.

Numbers in the input file are interpreted as spoken input, not DTMF.

Extraneous spaces and tabs within a line are ignored; however, blank lines are considered input and will usually cause the VoiceXML browser to throw an error.

You cannot use a text file to provide input to VoiceXML `<audio>` or `<record>` elements.

Testing Built-in Field Types

Table 36 provides examples of the types of input you might specify when testing an application that uses the built-in field types.

Table 36. Sample Input for Built-in Field Types

Built-in Field Type	Sample Input
boolean	yes true positive right ok sure affirmative check yep no false negative wrong not nope
currency	three twenty five sixteen dollars and fifty seven cents ten dollars nine million two hundred thousand dollars
date	may fifth march December thirty first two thousand july first in the year of nineteen ninety nine yesterday tomorrow today

Built-in Field Type	Sample Input
digits	0 1 2 3 4 5 6 7 8 9
number	ten million five hundred thousand fifty three negative one point five positive one point five point fifty three one oh oh point five three one hundred oh seven point seven dot six
phone	nine one four nine four five two three eight eight nine zero four nine four five two three eight eight one nine zero four nine four five two three eight eight nine one four nine four five two three eight eight extension sixty three fifty one one eight hundred five five five one two one two one three eight five one three two six ten
time	one o'clock one oh five three fifteen seven thirty half past eight oh four hundred hours sixteen fifty twelve noon midnight now

Timing Issues

Occasionally, the delay in fetching VoiceXML pages may be long enough to cause the response being sent from your input file to arrive before the application is ready to receive it. If this occurs, the application and the input file will be out of synch with each other, and subsequent responses will be directed to the wrong prompt. This will generally cause the remainder of the dialog to result in **nomatch** errors.

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department T01B
3039 Cornwallis Road
Research Triangle Park, NC 27709-2195
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Copyright License

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information in softcopy, the photographs and color illustrations may not appear.

Trademarks

The following terms are trademarks or registered trademarks of the International Business Machines Corporation in the United States, other countries, or both:

DirectTalk

IBM

ViaVoice

WebSphere

Microsoft, Visual Basic, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

This glossary defines terms and abbreviations used in this publication. If you do not find the term you are looking for here, refer to *The IBM Dictionary of Computing*, SC20-1699, New York: McGraw-Hill, copyright 1994 by International Business Machines Corporation. Copies may be purchased from McGraw-Hill or in bookstores.

active grammar

A speech grammar that the speech recognition engine is currently listening for. One or more grammars will be active at any time, and the content of the active grammar(s) defines the user utterances that are valid in a given context.

ANI

Automatic Number Identification. A telephony service that tells the receiving party the telephone number of the calling party.

application

A set of related VoiceXML documents that share the same application root document.

application root document

A document that is loaded when any documents in its application are loaded, and unloaded whenever the dialog transitions to a document in a different application. The application root document may contain grammars that can be active for the duration of the application, and variables that can be accessed by any document in the application.

ASP

Microsoft Active Server Pages. One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. ASPs can be written in various scripting languages, including VBScript (based on Microsoft Visual Basic), JScript (based on JavaScript), and PerlScript (based on Perl).

bail out

The termination of a sequence of self-revealing help prompts, if the user repeatedly fails to provide an appropriate response.

barge-in

A feature of full-duplex environments that allows the user to interrupt computer speech output (audio file and text-to-speech). See also “[full-duplex](#)”.

CGI

Common Gateway Interface. One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. CGI scripts are typically written in Perl, although they can be written in other programming languages.

continuous speech recognition

The IBM WebSphere Voice Server SDK supports “continuous speech recognition”, in which users can speak a string of words at a natural pace, without the need to pause after each word. Contrast with “[discrete speech recognition](#)”.

cookie

Information that a Web server stores on a user’s computer when the user browses a particular Web site. This information helps the Web server track such things as user preferences and data that the user may submit while browsing the site. For example, a cookie may include information about the purchases that the user makes (if the Web site is a shopping site). The use of cookies enables a Web site to become more interactive with its users, especially on future visits.

cut-thru word recognition

See “[barge-in](#)”.

data prompts

Prompts where the user must supply information to fill in the field of a form. Contrast with “[verbatim prompts](#)”.

dialog

The main building block for interaction between the user and the application. VoiceXML supports two types of dialogs: “[form](#)” and “[menu](#)”.

discrete speech recognition

Users must pause briefly after speaking each word, to allow the system to process and recognize the input. Contrast with “[continuous speech recognition](#)”.

DNIS

Dialed Number Identification Service. A telephony service that tells the receiving party what telephone number the caller dialed.

DTMF

Dual Tone Multiple Frequency. The tones generated by pressing keys on a telephone’s keypad.

DTMF Simulator

A GUI tool that enables you to simulate DTMF input when testing your VoiceXML application on your desktop workstation. The VoiceXML browser communicates with the DTMF Simulator to accept DTMF input, and uses that input to fill in forms or select menu items within the VoiceXML application.

echo cancellation

Technology that removes echo sounds from the input data stream before passing what's left (that is, user speech) to the speech recognition engine. In a deployment environment, this is configured on the telephony hardware; if echo cancellation is poor, you may need to turn off “[barge-in](#)” or switch to “[half-duplex](#)”.

ECMAScript

An object-oriented programming language adopted by the European Computer Manufacturer's Association as a standard for performing computations in Web applications. ECMAScript is the official client-side scripting language of VoiceXML. Refer to the *ECMAScript Language Specification*, available at <http://www.ecma.ch/ecma1/stand/ECMA-262.htm>.

event

The VoiceXML browser throws an event when it encounters a **<throw>** element and certain specified conditions occur. Events are caught by **<catch>** elements that can be specified within other VoiceXML elements in which the event can occur, or inherited from higher-level elements. The VoiceXML browser supports a number of predefined events and default event handlers; you can also define your own events and event handlers.

form

One of two basic types of VoiceXML dialogs. Forms allow the user to provide voice or DTMF input by responding to one or more **<field>** elements.

full-duplex

Applications in which the user and computer can speak concurrently. Full-duplex applications use “[echo cancellation](#)” to subtract computer output from the incoming data to determine what was user speech. See also “[barge-in](#)”. Contrast with “[half-duplex](#)”.

grammar

A collection of rules that define the set of all user utterances that can be recognized by the speech recognition engine at a given point in time. The VoiceXML browser makes different grammars active at different points in the dialog, thereby controlling the set of valid utterances that the speech recognition engine is listening for. Grammars support word substitution and word repetition.

GSM

Global System for Mobile Communication. The cellular telephone network.

GUI

Graphical User Interface. A type of computer interface consisting of visual images and printed text. Users can access and manipulate information using a pointing device and keyboard. Contrast with “[speech user interface](#)”.

H.323

Audio communications protocol used by IBM WebSphere Voice Server with ViaVoice Technology.

half-duplex

Applications in which the user should not speak while the computer is speaking, because the speech recognition engine does not receive audio while the computer is speaking. Turn-taking problems can occur when the user speaks before the computer has finished speaking; using a unique tone to indicate the end of computer output can minimize these problems by informing users when they can speak. Contrast with “[full-duplex](#)”.

help mode

A technique for providing general help information explicitly, in a separate dialog. Contrast with “[self-revealing help](#)”. See “[Choosing Help Mode or Self-Revealing Help](#)” on page 131.

II digits

Information Indicator Digits. A telephony service that provides information about the caller’s line (for example, cellular service, payphone, etc.).

JMF

Java Media Framework.

JSGF

Java Speech Grammar Format. Refer to <http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/index.html>.

JSP

JavaServer Pages. One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. JSPs call Java programs, which is executed by the HTTP server.

JVM

Java Virtual Machine.

Lombard speech

The tendency of people to raise their voices in noisy environments, so that they can be heard over the noise.

machine directed

A dialog in which the computer controls interactions. Grammars are only active within their own dialogs. Contrast with “[mixed initiative](#)”.

menu

One of two basic types of VoiceXML dialogs. Menus allow the user to provide voice or DTMF input by selecting one menu choice.

menu flattening

Feature of natural command grammars that enables the system to parse user input and extract multiple tokens. Mixed initiative dialogs can provide similar benefits.

mixed initiative

A dialog in which either the user or the computer can initiate interactions. You can use form-level grammars to allow the user to fill in multiple fields from a single utterance, or document-level grammars to allow the form’s grammars to be active in any dialog in the same VoiceXML document; if the user utterance matches an active grammar outside of the current dialog, the application transitions to that other dialog. Contrast with “[machine directed](#)”.

mixed-mode applications

Applications that mix speech and DTMF input.

mu-law

The compression and expansion algorithm used in primarily in North America and Japan when converting from analog to digital speech data.

multi-modal application

An application that has both a speech and a visual interface.

natural command grammar

A complex grammar that approaches natural language understanding in its lexical and syntactic flexibility, but unambiguously specifies all acceptable user utterances. Contrast with “[natural language understanding \(NLU\)](#)”.

natural language understanding (NLU)

A statistical technique for processing natural language, using text that is representative of expected utterances to create a dictation-like model. NLU does not use grammars; instead, it uses statistical information to tag and analyze key words in an utterance. Contrast with “[natural command grammar](#)”.

out-of-grammar (OOG) utterance

The user input was not in any of the active grammars.

persistence

A property of visual user interfaces is that information is persistent; that is, information remains visible until the user moves to a new visual page or the information changes. Contrast with [“transience”](#).

prompt

Computer spoken output, often directing the user to speak.

pronunciation

A possible phonetic representation of a word that is stored in the speech recognition engine and referenced by one or more words in a grammar. A pronunciation is a string of sounds that represents how a given word is pronounced. A word may have several pronunciations; for example, the word “tomato” may have pronunciations “toe-MAH-toe” and “toe-MAY-toe”.

PSTN

Public Switched Telephone Network.

prosody

The rhythm and pitch of speech, including phrasing, meter, stress, and speech rate.

recognition

When utterances are known and accepted by the speech recognition engine. Only words, phrases, and DTMF key sequences in active grammars can be recognized.

recognition window

The period of time during which the system is listening for user input. In a full-duplex implementation, the system is always listening for input; in a half-duplex implementation or when barge-in is temporarily disabled, a recognition window occurs only when the dialog is in a state where it is ready to accept user input.

SDK

The IBM WebSphere Voice Server Software Developers Kit.

self-revealing help

A technique for providing context-sensitive help implicitly, rather than providing general help using an explicit help mode. Contrast with [“help mode”](#).

servlet

One of many server-side mechanisms for generating dynamic Web content by transmitting data between an HTTP server and an external program. Servlets are dynamically loaded Java-based programs defined by the Java Servlet API (<http://java.sun.com/products/servlet/>). Servlets run inside a JVM on a Java-enabled server.

session

A session consists of all interactions between the VoiceXML browser, the user, and the document server. The session starts when the VoiceXML browser starts, continues through dialogs and the associated document transitions, and ends when the VoiceXML browser exits.

speech browser

See “[VoiceXML browser](#)”.

speech recognition engine

Decodes the audio stream based on the current active grammar(s) and returns the recognition results to the VoiceXML browser, which uses the results to fill in forms or select menu choices or options.

speech user interface

A type of computer interface consisting of spoken text and other audible sounds. Users can access and manipulate information using spoken commands and DTMF. Contrast with “[GUI](#)”. See “[DTMF](#)”.

spoke too soon (STS) error

A recognition error that occurs when the user in a half-duplex application begins speaking before the turn-taking tone sounds and continues speaking over the tone and into the speech recognition window.

spoke way too soon (SWTS) error

A recognition error that occurs when the user in a half-duplex application finishes speaking before the turn-taking tone sounds.

stuttering effect

When a prompt in a full-duplex application keeps playing for more than 300 ms after the user begins speaking, users may interpret this to mean that the system didn’t hear their input. As a result, the users stop what they were saying and start over again. This “stuttering” type of speech makes it difficult for the speech recognition engine to correctly decipher user input.

subdialog

Roughly the equivalent of function or method calls. Subdialogs can be used to provide a disambiguation or confirmation dialog, or to create reusable dialog components.

text-to-speech (TTS) engine

Generates computer synthesized speech output from text input.

token

The smallest unit of meaningful linguistic input. A simple grammar processes one token at a time; contrast with “[menu flattening](#)”, “[natural command grammar](#)”, and “[natural language understanding \(NLU\)](#)”.

transience

A property of speech user interfaces is that information is transient; that is, information is presented sequentially and is quickly replaced by subsequent information. This places a greater mental burden on the user, who must remember more information than they need to when using a visual interface. Contrast with “[persistence](#)”.

turn-taking

The process of alternating who is performing the next action: the user or the computer.

URI

Universal Resource Indicator. The address of a resource on the World Wide Web. For example: **<http://www.ibm.com>**.

URL

Universal Resource Locator. A subset of URI.

User to User Information

ISDN service that provides call set-up information about the calling party.

utterance

Any stream of speech, DTMF input, or extraneous noise between two periods of silence.

verbatim prompts

Menu choices that the user can select by repeating what the system said. Contrast with “[data prompts](#)”.

voice application

An application that accepts spoken input and responds with spoken output.

VoiceXML

Voice eXtensible Markup Language. An XML-based markup language for creating distributed voice applications. Refer to the VoiceXML Forum Web site at **<http://www.voicexml.org>**.

VoiceXML browser

The “interpreter context” as defined in the VoiceXML 1.0 specification. The VoiceXML browser fetches and processes VoiceXML documents and manages the dialog between the application and the user.

Wizard of Oz testing

A testing technique that allows you to use a prototype paper script and two people (a user and a human “wizard” who plays the role of the computer system) to test the dialog and task flow before coding your application. See [“Prototype Phase \(Wizard of Oz Testing\)” on page 104](#).

XML

eXtensible Markup Language. A standard metalanguage for defining markup languages. XML is being developed under the auspices of the World Wide Web Consortium (W3C).