

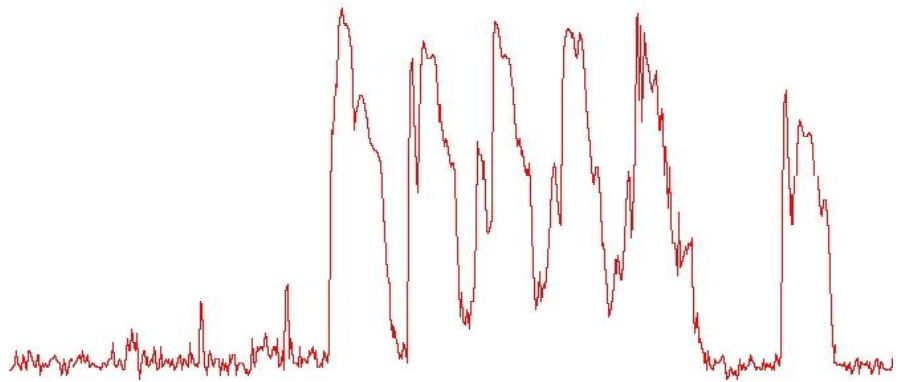
DRAFT



at&t

AT&T Speech Mashups

Application Developer's Guide



D R A F T

Copyright © 2009-2010 AT&T. All rights reserved.

AT&T Speech Mashups, Application Developer's Guide, v. 1010

Printed in USA.

October 2010

All rights reserved.

AT&T, WATSON, AT&T logo, and all other marks contained herein are trademarks of AT&T Intellectual Property and/or AT&T affiliated companies.

Apple and Mac are trademarks of Apple Computer, Inc. registered in the U.S. and other countries.

Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States.

All other trademarks are the property of their respective owners.

No part of this document may be reproduced or transmitted without written permission from AT&T.

Every effort was made to ensure that the information in this document was complete and accurate at the time of printing. However, information is subject to change.

1 About this guide

This guide describes how to incorporate AT&T speech technologies with a web-based application to create a speech mashup. It is written for developers and assumes a knowledge of web applications and the standards used for web applications, including HTTP, XML, JSON, and EMMA.

Information in this guide is divided into the following chapters:

Chapter 3, “Overview,” is a high-level description of the speech mashup architecture with information about connecting to AT&T speech servers, including the WATSON speech recognizer and the Natural Voices TTS (text-to-speech) server. This chapter also includes instructions for registering at the portal, which manages account and connection information and is where you upload and manage grammars needed for applications requiring automatic speech recognition (ASR).

Chapter 4, “Creating a Rule-Based Grammar,” describes the syntax and notations used to create the two rule-based grammars supported by WATSON ASR: XML and WBNF. It also describes how to parse the semantic interpretation so that a specific string is returned, based on the recognized result.

Chapter 5, “Setting up a Statistical Language Model,” outlines the steps for building a statistical language model to recognize the vocabularies too unpredictable to be effectively captured by a rule-based grammar.

Chapter 6, “Uploading, Managing, and Testing Grammars,” describes how to upload grammars for use with the WATSON recognizer and then test how well the speech recognition performs with a specific grammar. General guidelines are also provided for increasing the recognition accuracy.

Chapter 7, “Modifying Output Speech for Natural Voices,” describes how to modify the output speech using SSML tags or to change word pronunciations using phonemic transcriptions.

Chapter 0, “The MIME Multipart/Mixed format for audio/metadata interleaving is documented in the AT&T Internal Addendum to this document.

Building a Speech Mashup Client,” gives instructions on how to build (and modify existing) clients for Java ME and the iPhone as well as a plugin for a Safari browser on a Mac®. It includes information needed to write or modify clients so they can access the WATSON server or the Natural Voices server, or both.

Chapter 9, “Administration & Troubleshooting,” describes how to view log files, change portal account information, and address problems that may occur.

1.1 Getting more help

Development of speech mashups is ongoing. To keep current with updates or request more information, send emails to watsonadm@research.att.com.

1.2 Change History

June 2011:

- > Added a chapter covering the transcription and utterance export facility.

October 2010:

- > Added a chapter describing the WatsonApplet and AudioPlayer applets, for creating speech-enabled web applications for Java-enabled web browsers.
- > Added a chapter describing the container formats used to deliver bookmarked audio streams.

February 2009:

- > The speech mashup manager now manages connections to AT&T's text-to-speech server, Natural Voices, allowing users to convert text to streaming audio. Parameters allow the client to specify a particular voice, sample rate, and other information relevant for speech.

SSML tags, which allow you to modify the text normalization, pronunciation, and prosody in the output speech are documented in Chapter 5.
- > Support has been added for connecting to outside servers in order for an additional processing step to be applied before or after the speech-related task. For example, postprocessing would enable a phone number or other database item to be returned with the recognition, such as for a directory assistance request. For TTS, preprocessing might include applying application-specific text normalization before the text is converted to speech.
- > When creating transcriptions, you can insert annotations into the transcript, (including background speech or non-speech events such as hangups and tones) by clicking a button.
- > Limits have been applied to the amount of space allocated for all grammars associated with a single UUID. The amount of space taken up by current grammars is displayed on the Update account information page.

August 2008: Initial version

2 Contents

1	About this guide	iii
1.1	Getting more help	iv
1.2	Change History	iv
2	Contents	v
3	Overview	1
3.1	Speech mashup architecture.....	1
3.2	What is WATSON ASR?.....	2
3.2.1	Grammars for recognition.....	3
3.3	What is Natural Voices?	4
3.4	The speech mashup portal	5
3.5	What you need to do.....	7
3.5.1	What you need to know.....	7
4	Creating a Rule-Based Grammar	9
4.1	Guidelines and best practices for creating a grammar	9
4.1.1	Pronunciation tags	10
4.1.2	Numbers and digits	11
4.2	Semantic tags	12
4.3	Define statements for controlling the compilation.....	12
4.4	Creating an XML grammar	13
4.4.1	Adding pronunciation tags to an XML grammar	15
4.4.2	Adding semantic tags to an XML grammar.....	15
4.4.3	Using word weighting.....	16
4.4.4	A sample XML grammar.....	16
4.5	Creating a WBNF grammar	18
4.5.1	Rules.....	18
4.5.2	Adding pronunciation tags to a WBNF grammar.....	20
4.5.3	Adding semantic tags to a WBNF grammar	20
4.5.4	A sample WBNF grammar.....	21
5	Setting up a Statistical Language Model	23
6	Uploading, Managing, and Testing Grammars	25
6.1	Creating application directories for grammars	25
6.2	Uploading grammars	27
6.2.1	Uploading grammars interactively.....	28
6.3	Sharing and managing grammars	28
6.3.1	Editing prebuilt or shared grammars	29
6.4	Determining accuracy.....	30
6.4.1	Sending audio files for testing.....	30
6.4.2	Creating transcriptions	33
6.4.3	Comparing transcriptions to utterances.....	33
6.4.4	Exporting transcriptions and utterances	34
6.5	Checklist for improving accuracy	36
6.6	Setting recognizer preferences with a commands file.....	37
6.6.1	Preferences for automatic endpointing.....	38

7	Modifying Output Speech for Natural Voices.....	41
7.1	Using SSML tags	42
7.1.1	SSML syntax	42
7.2	Changing word pronunciations.....	44
7.3	Testing the TTS conversion.....	46
7.4	TTS With Bookmarks / Notifications.....	47
7.4.1	Introduction	47
7.4.2	Simple Format	48
7.4.3	Ogg Format	49
7.4.4	MIME Multipart/Mixed Format.....	50
8	Building a Speech Mashup Client.....	51
8.1	REST API information.....	51
8.1.1	Setting recognizer parameters.....	53
8.1.2	Request API parameters for TTS.....	54
8.2	Sample clients for devices	55
8.2.1	Client for Java ME	55
8.2.2	Native client for the iPhone.....	60
8.2.3	Safari plugin for Mac.....	62
8.3	Applets for Java-enabled browsers.....	64
8.3.1	Introduction	64
8.3.2	Speech Recognition (ASR).....	64
8.3.3	Text-to-Speech (TTS)	66
8.3.4	Loading Applets in Detail.....	69
8.3.5	Client Requirements	71
8.4	Configuring the client for the recognition result.....	72
8.4.1	Setting a threshold.....	73
8.5	Combining speech processing with other processing	74
8.5.1	Processing transaction steps.....	75
9	Administration & Troubleshooting	77
9.1	Viewing log files.....	77
9.2	Updating passwords and other account information.....	77
9.3	Troubleshooting	77
10	Glossary.....	79
11	Index	81

3 Overview

An AT&T® *speech mashup* is a web service that implements speech technologies, including both *automatic speech recognition* (ASR) and text to speech (TTS) for web applications. This enables users of an application to use voice commands to make requests (ASR) or to convert text to audio (TTS). Speech mashups work by relaying audio or text from a web application (any application that understands HTTP) on a mobile device or a web browser to servers at the AT&T network where the appropriate conversion takes place. The result of the process is returned to the web application.

Speech mashups can be created for almost any mobile device, including the iPhone, as well as web browsers running on a PC or Mac®, or any network-enabled device with audio input.

3.1 Speech mashup architecture

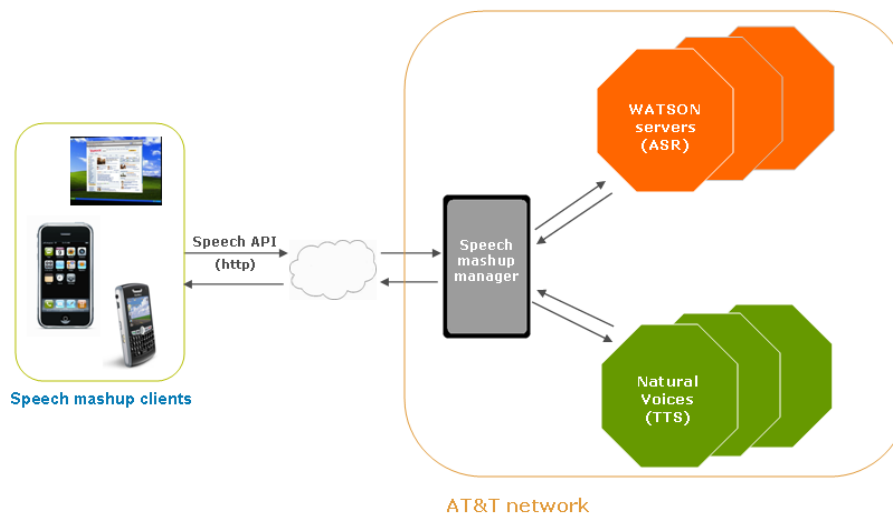


Figure 3.1 In a speech mashup, clients access AT&T speech services through the speech mashup manager

A speech mashup application consists of three main components that enable clients on a browser or mobile device to connect to AT&T speech servers:

- AT&T speech servers, including WATSON servers configured for ASR, and Natural Voices servers, which converts text to speech and returns streaming audio back to the web application.

- > A speech mashup client that relays audio (using HTTP) to the WATSON servers and accepts the recognition result. Examples of clients are available for Java ME devices, the iPhone, and the Safari browser on the Mac OS X.
- > The speech mashup manager (SMM), which opens and manages direct connections to the appropriate AT&T speech servers on behalf of the client, including resolving device dependency issues, and performing authentication and general accounting.

The following diagram details the client elements:

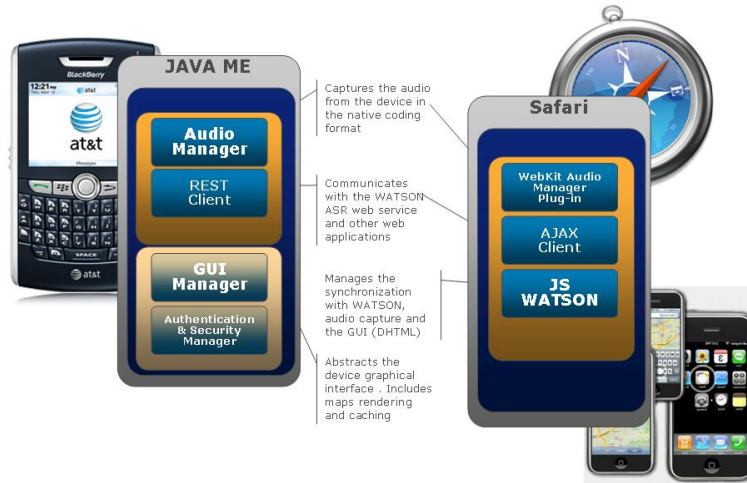


Figure 3.2 Speech mashup clients (available for Java ME, the iPhone, and the Safari Mac browser) interface between the web application and WATSON ASR.

3.2 What is WATSON ASR?

WATSON ASR is the automatic speech recognition component of the WATSON system responsible for converting spoken language to text. This process in WATSON ASR is broken into three main steps: identifying speech features in the incoming audio, mapping those features to basic language sounds (contained in an acoustic model), and matching sounds to phrases and sentences contained in the grammar. The particulars of how this is done is not important to know when creating speech mashups.

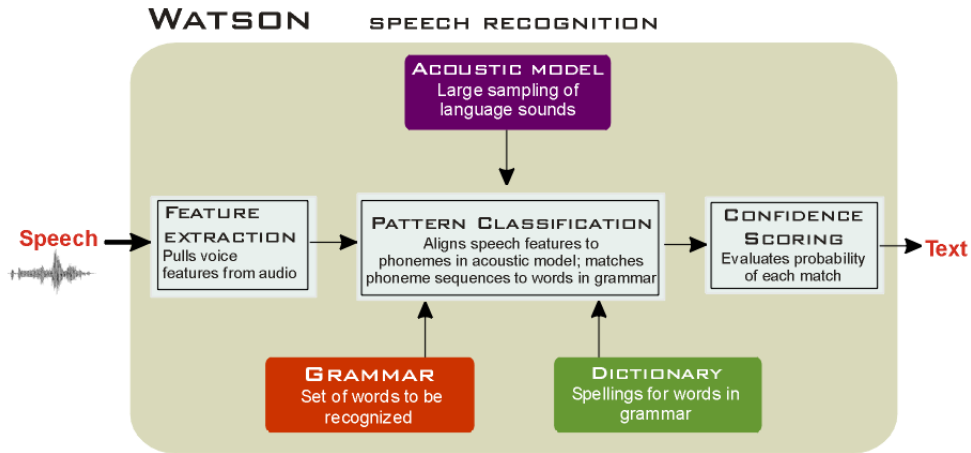


Figure 3.3 ASR converts speech to text based on words, phrases, and sentences in the grammar.

Speech mashups are intended to be easy to develop, so whenever possible individual components and default parameters are provided. WATSON ASR includes a general dictionary and acoustic model so the only ASR component you need to provide is the grammar.

3.2.1 Grammars for recognition

A grammar contains the words, phrases, and sentences that the recognizer will try to recognize. A grammar must be customized for each application to include all possible word sequences that could be uttered within the context of the application.

There are two general types of grammars: *rule-based* and *statistical*. Both types are compiled from a text source.

- In a rule-based grammar, you explicitly define the sentences (and order of words) that can be understood by the recognizer. Words not included in the grammar cannot be recognized.

WATSON ASR supports two grammar formats: the XML standard (W3C) and its own BNF (WBNF). See chapter 2 for specific information on the syntaxes used to create XML and WBNF grammars.

- A statistical language model (SLM) does not use explicit rules, but instead uses the statistical properties of thousands of transcribed utterances to help infer the language.

Constructing an SLM requires a very large set (several tens of thousands) of sentences representative of what a speaker might say in a given context. The data set itself does not need to explicitly contain every sentence that should be recognized because the SLM will model combinations of snippets from each training sentence. In fact the SLM will have probability estimates for all word combinations of all words from the training text, so good vocabulary coverage in the training text is very important.

The procedure for creating an SLM is outlined in chapter 3.

3.3 What is Natural Voices?

Natural Voices converts text to speech. It has built-in rules for normalizing text (such as converting common abbreviations to words and correctly pronouncing numbers) and assigning prosody to make the generated speech sound as natural as possible.

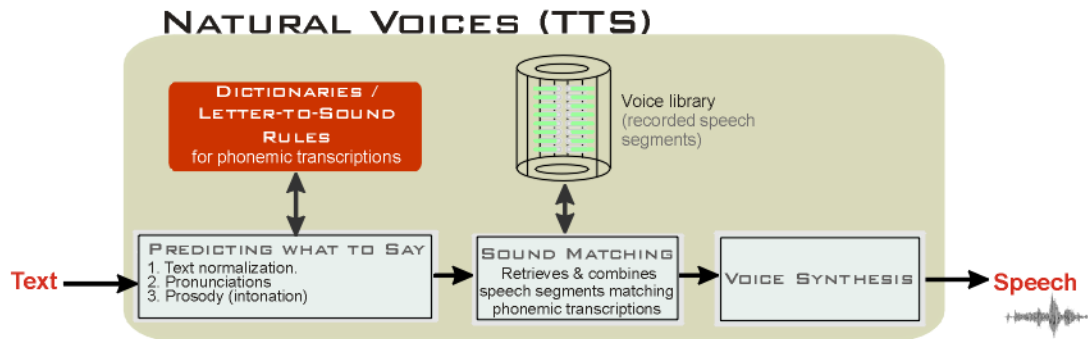


Figure 3.4 Natural Voices TTS Architecture

In addition, Natural Voices will properly interpret Synthesized Speech Markup Language (SSML) tags embedded in the text to more closely control normalization, pronunciation, and prosody.

3.4 The speech mashup portal

The portal is the main interface between developers and the speech mashup, managing accounting and login information, and providing a repository for speech mashup-related files that both the developer and the speech servers (ASR and TTS) can access.

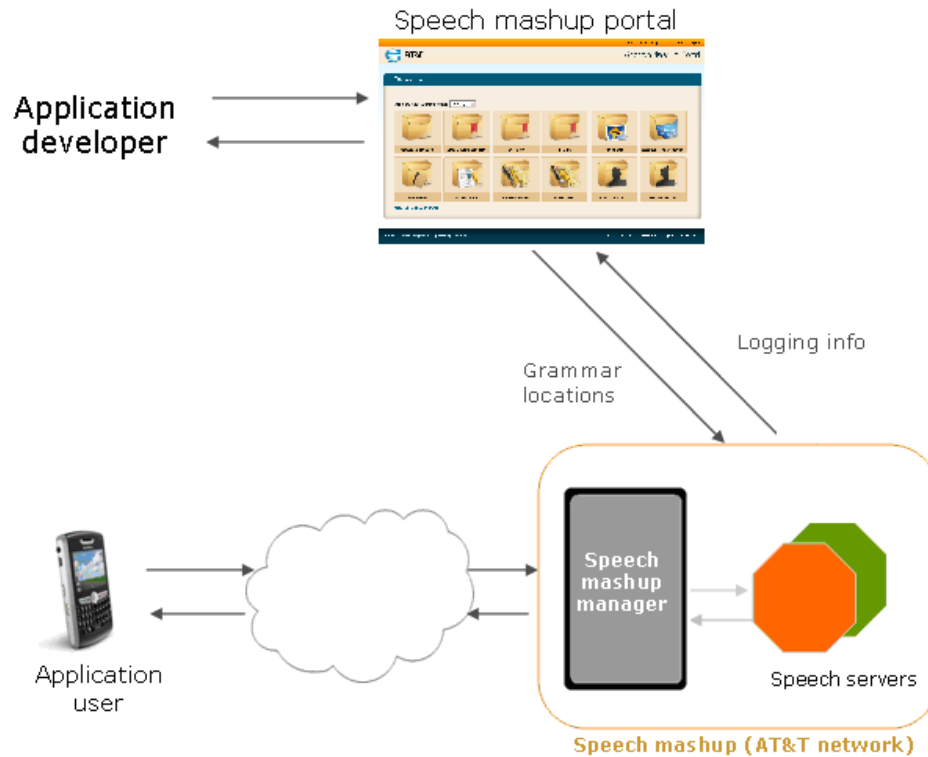


Figure 3.5 and information related to speech mashups are accessible through the speech mashup portal

The portal is where you do the following:

- > Obtain the unique user ID (UUID) to be associated with your account. The UUID is needed for logging into the portal and enables the client to access AT&T speech services.
- > Create application directories for organizing all files and grammars associated with ASR applications.
- > Upload and manage the grammars needed for ASR applications. You store your personal grammars at the portal and also access prebuilt and shared grammars. Grammars are automatically compiled when uploaded.
- > View current and past activity (log files).
- > Make transcriptions of audio files (useful for evaluating the accuracy of the speech recognition).
- > Read and post messages to the Speech Mashup users' board.

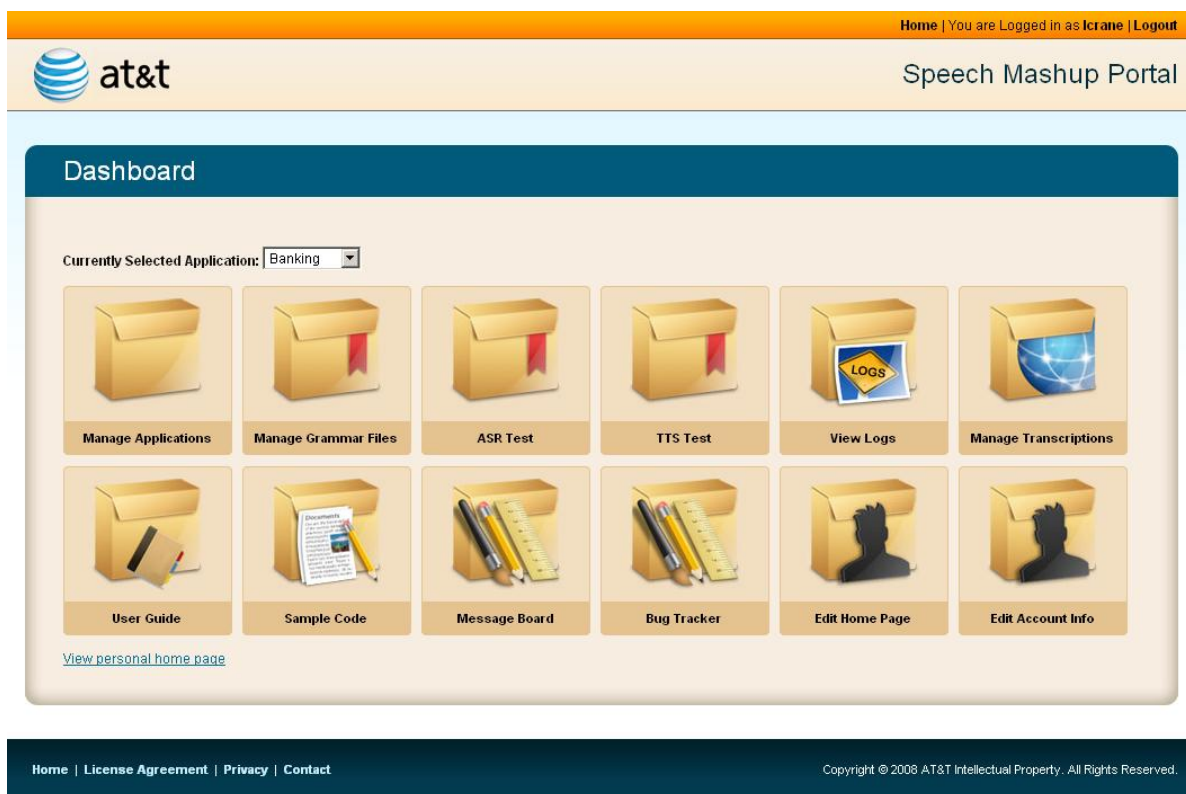


Figure 3.6 Home page of speech mashup portal

You register once to receive a unique UUID (universally unique identifier), which allows you to log into the portal and to write clients that can access the AT&T speech servers.

The UUID is used by the speech mashup to uniquely identify and retrieve your grammars, log files, and transcriptions. Multiple applications can be associated with a single UUID.

3.5 What you need to do

As much as possible, AT&T provides default components and parameter settings to make speech mashup development easy. Building a speech mashup consists of the following steps:

- 1 **Register for a portal account.**
<http://service.research.att.com/smm/>
 You'll receive back an UUID for logging onto the portal and enabling the client to access AT&T servers.
- 2 **Create an application directory for each application or select an existing one.**
 The application directories will contain the grammars, log files, and all other entities associated with a specific application.
- 3 **Create and upload one or more grammars or use a builtin or shared grammar.**
 ASR applications only. Create your own rule-based grammar (XML or WBNF) or SLM and upload it or select a grammar provided on the portal or shared by another user. Uploading a grammar (or a zipped file of multiple grammars) to the portal
- 4 **Build a speech mashup client from existing examples.**
 The client can be written in any suitable programming language (Java, JavaScript, or any other language) depending on the device. Three sample clients can be downloaded: a Java-based client for any Java ME mobile, iPhone native application client, and a Safari Mac OS X plugin.

3.5.1 What you need to know

The following table summarizes the major specifications and requirements for speech mashups.

Table 3.1 Speech mashup specifications	
Speech mashup portal	http://service.research.att.com/smm/
Device support	Java ME devices, iPhone, Safari browser (Mac®)
Audio format (speech mashup portal),	AMR (Adaptive multi-rate coding) AU (μ-law, 16-bit linear) WAV (μ-law, linear) CAF (μ-law, linear) Raw audio (μ-law and 16-bit linear) is also supported, but requires you to provide the sample rate if other than 8000 Hz. (See page 52.)
Output format	ASR: XML (default), JSON, EMMA (recommended) TTS: AMR, AU (μ-law, A-law, 16-bit linear)
Supported languages	ASR: US English (en-us, the default) and the US dialect of Spanish (es-us). TTS: US English
Grammar types	XML (SRGS), WBNF (WATSON BNF), and SLMs (statistical language models)
User ID (UUID)	Allows access to portal and enables the speech mashup client to access WATSON servers. Obtained by registering at the portal.

DRAFT

4 Creating a Rule-Based Grammar

This chapter summarizes how to build rule-based grammars. Although two formats are described here—the standard XML and the WATSON version of BNF—it is recommended that all new grammars be created in XML; WBNF is provided for backward compatibility.

To create a rule-based grammar:

1. In a text editor, create the rules. See “Creating an XML grammar” or “Creating a WBNF grammar” as appropriate.
2. Save the file using the extension `.grxml` for an XML grammar or `.wbnf` for a WBNF grammar.

A set of prebuilt grammars is available from the portal to be used both as standalone grammars and for assembling with other grammars. For a listing, see page 29. Use prebuilt grammars with caution and always thoroughly test as you would with grammars you create.

WBNF grammars can be assembled from other grammars by inserting `#include` statements at the top of the grammar source. Assembling larger grammars from smaller ones allows you to take advantage of existing grammars and not have to rewrite rules.

`#include` statements use the following syntax:

```
#include "FILENAME.wbnf"
```

Note that there is a limit to the size of grammars; exceeding the limit generates an error message. If your combined source and compiled grammars exceed the limit, contact AT&T Research at watsonadm@research.att.com.

4.1 Guidelines and best practices for creating a grammar

Effective recognition depends on careful design of the rules. Follow these guidelines:

- > **Define all possible utterances, but define them as narrowly as possible.**

No word, phrase, or sentence will be recognized unless it is included in the grammar. Out-of-grammar words may be matched to the closest in-grammar word (for example, *New York* for *Newark* if Newark isn't in the grammar) or may not be matched at all, resulting in a recognition error.

> **Account for multiple ways of saying the same information.**

Some information, such as long numbers, dates, or years, may be described in various ways depending on context; 111 could be “one hundred eleven” (a quantity) or “one eleven” for an address; “1234” could be recited as “twelve hundred thirty four” or “one thousand, two hundred eighty-four.” The grammar must take into account various ways of saying the same information.

> **Avoid making broad grammars that can identify any utterance.**

Write the rules to exclude unlikely utterances. The longer the grammar or the more utterances it can recognize, the more inefficient it becomes. You don’t want to waste time searching for sentences that would never be uttered. This includes ungrammatical constructions or extremely unlikely word combinations or values that don’t fall within a reasonable range.

If a grammar becomes too large, consider changing the prompt to make it more directed.

> **Avoid prompts that encourage very short words or similar sounding words.**

Short, one-word responses provide little context that the recognizer can exploit. For short word responses, use the name pron tag (see next section) to increase recognition.

If the grammar contains similar sounding words (*reservation* and *reconfirmation* for example), rewrite the prompt to guide the speaker to use distinct words.

> **Use pronunciation tags when appropriate.**

The acoustic model contains special phone sets for whole-word pronunciations for the following cases: names, alpha, digits, confirm, quantities (whole numbers). See the next section for more information about pronunciation tags.

> **Substitute full words for abbreviations. For example, use *street* and *doctor* rather than St. and Dr.**

4.1.1 Pronunciation tags

Words pronounced in isolation, such as digits and one- or two-word answers (such as names and confirmations) pose a difficult recognition problem because they provide little context for the recognizer to exploit. For such cases, five specialized phone sets are contained in the acoustic model provided with WATSON ASR.

- > **digits**, for the numbers 0-9 when used in isolation or when spoken as part of a telephone, account, and other number
- > **alpha**, for acronyms or when spelling out names or other words
- > **name**, for proper names and other short utterances such as states and cities
- > **quant** (quantity), for natural numbers (10 and above) or utterances referring to quantities
- > **confirm** (confirmation), for yes, no, maybe, and other similar utterances

To make use of these specialized pronunciations, you insert pronunciation tags (pron tags) in the grammar rules. How these tags are represented in the grammar is described in the sections describing the XML and WBNF grammars.

4.1.2 Numbers and digits

The way in which you represent numbers within a grammar depends on the number itself and the context.

As a general rule, use digits for numbers smaller than 10. Using digits automatically causes the compiler to use the digits phone set within the acoustic model (this is the same as specifying the digit pronunciation tag).

Use words for numbers larger than 10 and specify that the quantity pron tag be used.

When you have account, telephone, or other numbers that combine digits and larger numbers, use both digits and words as appropriate. For example, for a 1-800 number, use digits for 1 and 8, and then use the quantity pronunciation tag for “hundred” as shown here:

```
<phone-number> = 1 8 _{ pron-quant hundred } _
```

The following are guidelines for representing numbers and digits in a grammar.

- > Use digits, rather than words, for numbers up to and including 10, such as those making up an account, telephone, or other number.
- > For natural numbers, use the word forms of the number (eleven, twelve . . .; twenty-one, twenty-two, twenty-three . . . etc.)
- > Spell out ordinal numbers such as first and third.
- > For account, phone, and other numbers with a known, fixed number of digits, write the grammar to allow only that number of digits.

Rather than simply allowing multiple digits (<item repeat=1-10>), specify the exact number of digits as shown here:

```
<rule id="digit">
  <one-of>
    <item>1</item>
    <item>2</item>
    <item>3</item>
    <item>4</item>
    <item>5</item>
    <item>6</item>
    <item>7</item>
    <item>8</item>
    <item>9</item>
    <item>0</item>
  </one-of>
</rule>
<rule id="digits">
  <item repeat="10">
    <ruleref id="digit">
  </item>
</rule>
```

4.2 Semantic tags

In an actual application, the recognized utterance is often not used directly. Instead the end application expects a predefined string that determines the next transaction to perform. For example, a fast-food customer might order a drink using any combination of phrases (“I want a coke,” “coke, please,” “give me a coke”), but all the end application needs to know is “coke.”

Translating between the uttered phrase and the string expected by the application is done through the use of *semantic tags* in the grammar. In the previous example, a semantic tag could convert specified phrases to the string expected by the end application (in this case, “coke”).

Semantic tags are implemented in XML using the W3C standard SISR (Semantic Interpretation for Speech Recognition), which uses a script language to return semantic results. See <http://www.w3.org/TR/semantic-interpretation/>.

The way in which the semantic tags are entered into the grammar depends on the grammar type. See “Adding semantic tags to an XML grammar” (see page 15) or “Adding semantic tags to a WBNF grammar” (page 20) as appropriate.

4.3 Define statements for controlling the compilation

Define statements, which can be included in the grammar, offer some control over the way in which grammars are compiled; you can, for instance, substitute a different acoustic model or specify the use of pronunciation tags.

In XML, define statements are specified using the WATSON extension to SRGS `<watson:option>`, which takes a both a name and a value attribute. The following example shows the use of the define statement to change the acoustic model:

```
<watson:option name="am" value="gentel06" />
```

The following table describes the options for both name and value that are most useful for speech mashups.

Table 4.1 Define statement options

name	value
pron	Applies specified pron tag to entire grammar. Can be alpha , digits , name , quant , confirm . See page 10 for more information about pron tags.
lang	ISO country code for language to use. Currently US English (en_us) and the US dialect of Spanish (en_es) are supported.
am	Short name or full path of acoustic model. By default, grammars are compiled with the gentel04 acoustic model. In some cases (such as for short words), gentel06 may provide better recognition.
word_penalty, sil_penalty	Both take float values. If after evaluating the accuracy of a grammar, lowering the word penalty (less than 3) for too many deletions may increase accuracy (for out-of-grammar words). Raising it above 3 can sometimes increase accuracy if there were many insertion errors. If changing the word penalty has no effect, try increasing the silence penalty (set to 4 or 5). Setting the silence penalty is similar to lowering the word insertion penalty.
Grammar types	XML (SRGS) and SLMs (statistical language models). WBNF is also supported for backward compatibility, though it should not be used for creating new grammars.
User ID (UUID)	Allows access to portal and enables the speech mashup client to access WATSON servers. Obtained by registering at portal.

By convention, define statements are inserted at the beginning of the grammar. The syntax for define statements in the WBNF grammar are as follows:

```
#define default_pron_tag      alpha, digit, name, quant, confirm
#define lang                  en_us, en_es
#define word_penalty          float
#define sil_penalty           float
#define start_rule            name of the start (aka root) rule. Use this when the grammar starts
                             with a rule other than <START>
```

4.4 Creating an XML grammar

The XML syntax is described in detail in the Speech Recognition Grammar Specification (SRGS) (<http://www.w3c.org/TR/2002/CR-speech-grammar-20020626>). The following paragraphs summarize the main points. See table 2.2 for a listing of the most common XML operators and special characters.

An XML grammar consists of a header followed by a body and, like HTML and VoiceXML, uses markup case-insensitive tags and plain text. A *tag* is a keyword enclosed by angle brackets (< >). Tags, which appear in nested pairs, are not defined; you define your own.

An XML grammar starts and ends with the grammar tags (<grammar> ... </grammar>) and must contain at least one rule element (<rule> ... </rule>).

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

```
<item weight="0.8">New York </item>
```

The rule name for each rule definition must be unique within the grammar and it cannot contain the following characters or special rules: . : - GARBAGE NULL VOID.

The XML version must be included in the first line (the encoding attribute that is sometimes included is not required when writing an XML grammar for WATSON). A single root must also be defined:

```
<grammar version="1.0" root="typeofLoan">
```

The following is a simple XML grammar that prompts to know whether an account is new or existing (a longer example is given on page 16):

```
<grammar version="1.0" root="typeofLoan">
  <!--Grammar to select a new or existing account -->

  <rule id="typeofLoan">
    <ruleref uri="#hes"/>
    <ruleref uri="#object"/>
    <ruleref uri="#post"/>
  </rule>

  <rule id="hes">
    <item repeat="0-1">
      <one-of>
        <item>huh</item>
        <item>um</item>
      </one-of>
    </item>
  </rule>

  <rule id="object">
    <item>
      <one-of>
        <item>new</item>
        <item>existing</item>
      </one-of>
    </item>
    <item>account</item>
  </rule>

  <rule id="post">
    <item repeat="repeat="0-1">
      <one-of>
        <item>bye</item>
        <item>good-bye</item>
        <item>that's all</item>
        <item>I'm done</item>
      </one-of>
    </item>
  </rule>
</grammar>
```

Table 4.2 XML Notations

Element	Description
"<one-of>"	A set of alternatives ("loan" or "home" or "interest rate").
repeat="0-1"	Optional expression.
repeat="n"	Repeat the expression exactly <i>n</i> times.
repeat="n-m"	Repeat the expression between <i>n</i> and <i>m</i> times.
repeat="n-"	Repeat the expression <i>n</i> times or more.
Special characters	Function
"<!-- -->"	Enclose a comment.
Special rules	Function
GARBAGE	A rule that matches any speech up until the next rule match, the next token, or until the end of spoken input.
NULL	Defines a rule that matches if the speaker doesn't say anything.
VOID	Defines a rule that can never be spoken. Inserting VOID into a sequence automatically makes that sequence unspeakable.

4.4.1 Adding pronunciation tags to an XML grammar

For an XML grammar, pron (pronunciation) tags have the following form:

```
<item pron="name-of-phone-set"> term </item>
```

Where *name of phone set* can be one of the following: **digits**, **alpha**, **name**, **quant**, or **confirm**, and where *term* is the word to which the pron tag applies.

In this sample XML grammar, the "pron" attribute for <item> causes the compiler to use a digit transcription for '0' if one is available but use the default pronunciation for 1.

```
<grammar root="start">
  <rule id="start">
    <item repeat="1-" repeat-prob="1.0">
      <one-of>
        <item pron="digits"> 0 </item>
        <item> 1 </item>
      </one-of>
    </item>
  </rule>
</grammar>
```

4.4.2 Adding semantic tags to an XML grammar

To add semantic tags to an XML grammar, insert the string **tag-format="semantics/1.0"** in the first <grammar> line (<grammar tag-format="semantics/1.0" root="object">). Then add <tag> elements to override the returned value of a grammar component using a script. Do not insert spaces between the double quotes.

```
<tag> out="tag-content"</tag>
```

In the following example, the rule “object” contains tags that specify the returned string depending on what was matched.

```
<rule id="object">
  <one-of>
    <item>home      <tag> out="newloan"  </tag> </item>
    <item>refinancing <tag> out="refi"      </tag> </item>
    <item>refinance  <tag> out="refi"      </tag> </item>
    <item>loan       <tag> out="newloan"  </tag> </item>
    <item>interest   <tag> out="rates"    </tag> </item>
    <item>rate       <tag> out="rates"    </tag> </item>
    <item>rates      <tag> out="rates"    </tag> </item>
  </one-of>
</rule>
```

Thus the same string (**refi**) is returned depending on whether the speaker says refinancing or refinance; the string **rates** is returned if the speaker says *interest*, *rate*, or *rates*.

4.4.3 Using word weighting

XML grammars support word weights so that the grammar can better model the word statistics of an actual application. Insert weight attributes within the nested `<item>` tags, and place a floating-point value for a weight, as shown here:

```
<item>
  <one-of>
    <item weight="0.8">New York </item>
    <item weight="0.2"> Newark </item>
  </one-of>
</item>
```

4.4.4 A sample XML grammar

The following example shows a possible XML syntax for the prompt: “You have reached the Home Mortgage Department. Are you calling about buying a new home, refinancing an existing one, or are you calling to get the current interest rates?”

```
<grammar version="1.0" root="typeofLoan" tag-format="semantics/1.0">
  <!--Grammar to select to new loans, refinancing, interest rates -->
  <rule id="typeofLoan">
    <ruleref uri="#hes"/>
    <ruleref uri="#preamble"/>
    <ruleref uri="#object"/>
    <ruleref uri="#post"/>
    <tag>
      out = rules.object;
    </tag>
  </rule>
  <rule id="hes">
    <item repeat="0-1">
      <one-of>
```

```

        <item>huh</item>
        <item>um</item>
    </one-of>
</item>
</rule>

<rule id="preamble">
    <item repeat="0-1">
        <one-of>
            <item>hi</item>
            <item>please</item>
            <item>I want</item>
            <item>I want to</item>
            <item>I'm interested in</item>
            <item>I'm calling about</item>
            <item>just</item>
            <item>just give me</item>
        </one-of>
    </item>
</rule>

<rule id="object">
    <one-of>
        <item>home</item>
        <item>new home          <tag> out="newloan" </tag> </item>
        <item>new loan          <tag> out="newloan" </tag> </item>
        <item>existing loan      <tag> out="refi" </tag> </item>
        <item>existing home loan <tag> out="refi" </tag> </item>
        <item>existing           <tag> out="refi" </tag> </item>
        <item>refinancing        <tag> out="refi" </tag> </item>
        <item>refinancing my home loan <tag> out="refi" </tag> </item>
        <item>refinance my home loan <tag> out="refi" </tag> </item>
        <item>refinance my loan   <tag> out="refi" </tag> </item>
        <item>refinancing an existing loan <tag> out="refi" </tag> </item>
        <item>refinance          <tag> out="refi" </tag> </item>
        <item>loan</item>
        <item>interest           <tag> out="rates" </tag> </item>
        <item>interest rate       <tag> out="rates" </tag> </item>
        <item>interest rates      <tag> out="rates" </tag> </item>
        <item>rate                <tag> out="rates" </tag> </item>
        <item>current rate        <tag> out="rates" </tag> </item>
        <item>current rates       <tag> out="rates" </tag> </item>
        <item>current interest rates <tag> out="rates" </tag> </item>
    </one-of>
</rule>

<rule id="post">
    <item repeat="0-1">
        <one-of>
            <item>bye</item>
            <item>good-bye</item>
            <item>that's all</item>
            <item>I'm done</item>
        </one-of>
    </item>
</rule>

```

```
</grammar>
```

This grammar is for demonstration only. In practice, this grammar might elicit a range of responses difficult to predict (note the preamble rule); for this reason, you might redo the application as a series of directed prompts (“Are you calling about buying a new home?”) or consider doing an SLM.

4.5 Creating a WBNF grammar

This section describes the WBNF syntax, which is WATSON’s variation of the standard BNF. Because support for this syntax is being discontinued, it is recommended that all new rule-based grammars be created using XLM.

At its simplest, WBNF source text is composed of rules, which themselves are composed of words, operators, and other rules. Rule names are enclosed in angle brackets (< >) and are separated from the matching expression by =. A semicolon denotes the end of a rule. The syntax is shown here:

```
<rulename> = <word, rule, or expression> | < word, rule, or expression> ;
```

Elements to the right of the rule name are logically AND’ed together, with the pipe symbol | to separate alternate responses (see table 2.3 for other operators and special characters):

```
<loan-type> = new | existing ;
```

Words are tokens representing the speech part of the grammar. All words have to match some speech. As much as possible they are written out as regular words, but they could be arbitrary sequences of letters and digits.

Words are separated by white space and are not case sensitive (though the case used in the source will be preserved in the recognition string). Words are tokens representing the speech part of the grammar, and are also called **terminals**, because, unlike rules, they cannot be expanded further. Terminals never appear on the left side.

Optional words are enclosed in square brackets ([]), and parentheses are used to group words and clarify meaning. In this example, the parentheses makes clear that “loan” can be preceded by one of several qualifiers:

```
( new | existing | old ) [home] loan
```

4.5.1 Rules

By convention, WBNF grammars begin with the <START> rule (the compiler issues a warning message if it’s missing). (Rule names are tokens between angle brackets.) Avoiding recursion is recommended or the compilation may fail. Individual tokens within a rule are separated by white space.

A rule is often called a non-terminal, because it can be expanded into terminals (words) and other rules. Only grammars that can be fully expanded to terminals can be compiled. Partial grammars, grammars containing missing rules, cannot be compiled.

In WBNF, all rules have public scope.

Whether a given grammar source text contains a complete grammar may depend on the choice of the start rule. Not all rules defined in the source need to be reachable from the start rule; they can be left dangling. They should be commented out for clarity.

Here's a simple three-rule grammar (the two forward slashes in the first line indicate comments):

```
// Are you calling about a new account or an existing account?
<START>          = [<hes>] [<preamble>] <account-type> ;
<preamble>       = ([I'm calling] about ) | hi ;
<account-type> = (a new | an existing) (account | one) |
                  (new | existing) [account | one] ;
```

Simple grammars such as this one could be written on one line, with only the <START> rule, though placing each rule on its own line clarifies the grammar structure.

The above grammar can result in the following utterances (as well as others):

```
I'm calling about a new account, About a new account,
I'm calling about an existing account, About an existing account,
A new account, An existing account, New, Existing
```

All utterances listed here would be considered as a possible candidate at the beginning of the prompt. As the recognizer works to match phonemes to the input audio, paths get ruled out.

Table 4.3 WBNF Notations

Special characters	Function
	(White space, including spaces, tabs, and new lines.) Separates words.
< >	Delimits a rule name.
;	Terminates a rule.
//	Indicates a comment (C++ style); anything after // is ignored.
/* */	Indicates a comment (C style); all following text is ignored. As in C, these delimiters do not nest.
" "	Sets off words containing special characters that would otherwise be treated as control characters. White space is not allowed between the double quotes.
\	Performs the same function as quotation marks when placed before a non-white space special character.
Special words	Function
...	Wildcard word (also called garbage) that matches any speech. Often used for keyword spotting, when all text except keywords are replaced by wildcards. The normal recognition string will contain "..." tokens where speech was matched, but the speech itself is not decoded or cannot be reported.
_garbage	Same as a wildcard (matches any speech), but is also insignificant: it is suppressed from the normal recognition string.
_silence	Matches non-speech, in particular silence. It is insignificant and does not appear in the normal recognition string. There is usually no need to add explicit silences to a grammar since WATSON ASR inserts optional silences at the beginning and end of the grammar as well as between all words.
_epsilon	Represents the null word, i.e., the absence of a word. It is unusual to find this word explicitly in a grammar, but it is often used by the compiler so you may come across it when examining lists of grammar symbols or examining the compiled g.fsm.

Operators are evaluated in the following order:

- > Parenthesized and optional expressions
- > Repeated expressions (those that use * or +)
- > Expression sequences
- > Expression alternatives (OR)

4.5.2 Adding pronunciation tags to a WBNF grammar

For a WBNF grammar, you can use pronunciation (pron) tags in two ways: by setting a global pron tag that covers the entire grammar, or by associating pron tags only with specific words.

To set a global pron, use a define statement at the beginning of the grammar:

```
#define default_pron_tag name-of-phone-set
```

Where *name-of-phone-set* can be **name**, **digits**, **quant**, **confirm**, or **alpha**.

For example, for a list of names you would use `#define default_pron_tag name`.

To specify a pron tag for individual words, insert a pron tag in the appropriate place in the grammar, as follows:

```
_ { pron-name-of-phone-set term } _ For example: _ { pron-name Reid } _
```

where *term* is replaced by the word or term to which the tag is applied.

The following shows a simple account number grammar, which uses the digits phone set.

```
<START> = <Digit> <Digit> <Digit> <Digit> <Digit> <Digit> <Digit> <Digit> <Digit> <Digit> ;
<Digit> = _ { pron-digit 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 } _ ;
```

4.5.3 Adding semantic tags to a WBNF grammar

A semantic tag replaces a recognized utterance with a predefined string that is expected by the end application.

To insert semantic tags, include an expression in the `<START>` rule according to the syntax shown here, where the expression is enclosed by the characters `_ {` and `_`:

```
<START> = [<hes>] [<preamble>] _ { "$=$a" <request> } _ [<closing>] ;
```

where `<request>` is a rule name you define within the grammar. `<request>` is just an example; you can assign any rule name that makes sense for your application.

In the body of the grammar, include the rule name definition, specifying both the string to be forwarded to the end application and the rule whose results are to be replaced by the string. Use double quotes around the string expression and do not include spaces between the double quotes. The syntax is as follows:

```
<rule-name> = _ { "$a='string'" <name-of-input-rule> } _
```

where *string* is the text of the semantic string sent to the end application, and where the rule name that follows is the rule whose returned utterance will be replaced by the semantic tag.

In the following example, any utterance allowed by the rule <newloan> (e.g., “I want a new loan,” “I’m calling about a new loan,” “new loan,” etc.) will be replaced by the string newloan.

```
<request>      = _{"$a='newloan'" <newloan> }_
```

The sample WBNF grammar shown starting on page 21 shows the use of semantic tags.

The portal contains a set of builtin WBNF grammars that can be used, after being tested and modified for the intended purpose, as standalone grammars or incorporated with other grammars. See page 29 for a listing.

4.5.4 A sample WBNF grammar

The sample WBNF grammar selects a mortgage type. For additional WBNF grammar examples, see the prebuilt grammars provided on the portal.

```
//This is for the loan-type prompt: "You have reached the Home
//Mortgage Department.
//Are you calling about buying a new home, refinancing an existing
//home, or are you calling to get the current interest rates?"

<START>  = [<hes>] [<preamble>] _{ "$=$a" <request> }_ [<closing>] ;

<hes>    = uh | um | ahhh;

<preamble> = hi | please |
            ([I want] to) |
            ([I'm interested] in) |
            ([I'm calling] about [getting]) |
            (give me) ;

<request> = _{"$a='newloan'" <newloan> }_ |
            _{"$a='existingloan'" <existingloan> }_ |
            _{"$a='interestrate'" <interestrate> }_ ;

<newloan> = [(buy | buying)] [a] new [(loan | home)] ;

<existingloan> = refinancing [my | a | an] [existing] [(home | loan)] |
                [refinancing] [my | a | an] existing (home | loan) |
                [refinance] [my | a | an] existing (home | loan) |
                refinance [my | a ] [(home | loan)] |
                [refinance] [my | a | an] [existing] [(home | loan)] ;

<interestrate> = [current] interest (rate | rates) |
                 [current] [interest] (rate | rates) |
                 current [interest] (rate | rates) ;

<closing>    = that's all | done | ok | thank you |
                I'm done | good-bye ;
```

This grammar is meant only as a demonstration. In practice, this grammar might elicit a range of responses difficult to predict (note the preamble rule); for this reason, you might redo the application as a series of directed prompts (“Are you calling about buying a new home?”).

DRAFT

5 Setting up a Statistical Language Model

When it's not possible to constrain speakers to a set of responses and the utterances are apt to be too unpredictable to be defined by rules, use a statistical language model (SLM). Instead of relying on hand-written rules that explicitly spell out which utterances can be recognized, an SLM derives its rules from the statistical properties of thousands of training sentences (the training *corpus*). By analyzing the training data, the SLM is able to estimate the probability that certain word sequences will appear together.

When passed a new utterance, the SLM can provide an *a priori* probability for it based on the patterns it detected in the training data. Such a grammar is much more flexible than a rule-based one.

Setting up a statistical language model requires training data, and lots of it. A minimum of 10,000 utterances is recommended.

The steps to setting up an SLM are the following:

1. Collecting sample data from the customer.

Sample data should include thousands of transcribed utterances (at least 10,000) and should be representative of the responses expected in the final application.

2. Obtaining transcriptions of the training data.

The file of transcribed utterances should include only text, and each utterance should be on a single line that ends with a carriage return.

Do not include punctuation. Remove from the file any non-language information, such as silences, hesitations, coughs, and other artifacts of spontaneous spoken language.

While not required, the filenaming convention is to append the extension `.train` to the text file.

3. Reviewing the transcription file and strengthening it if necessary.

Strengthening may be needed if critical words are not well represented in the training data. For instance, in service calls to a gas company, the words “emergency” or “gas leak” are very important, but if they appear only a handful of times out of 10,000 utterances, you should physically edit the file to add additional utterances containing those terms (25 times out of 10,000 should be adequate).

Experiment with the utterances to see if whether inserting additional words into likely and representative phrases prevent the utterance from being recognized.

4. Assigning semantic meaning to the text.

SLMs do not use semantic tags. To get the same effect, the SLM can be used in collaboration with AT&T's Natural Language Understanding (NLU) tools, which can be used to build classifiers for translating text to the normalized strings expected by the application.

5. Setting aside data for testing.

Before compiling an SLM, set aside a portion (a minimum of 10% is recommended) of the audio for testing purposes.

6. Uploading the SLM. See page 25.

6 Uploading, Managing, and Testing Grammars

Grammars belong to three contexts:

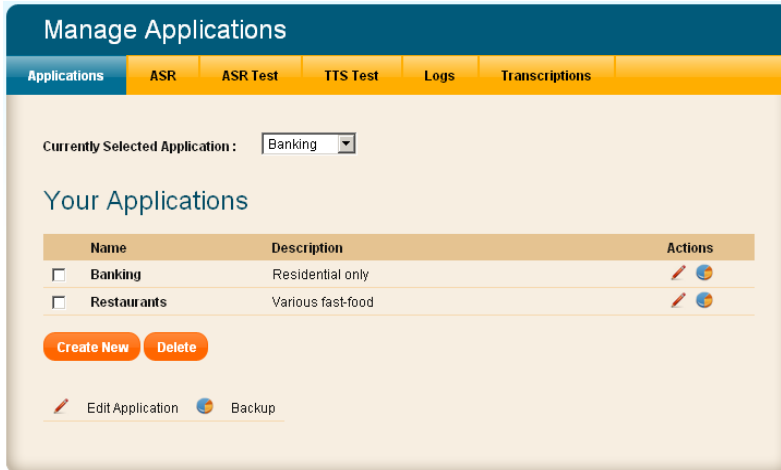
- > **My grammars.** These are your personal grammars that you create and control. They are not accessible to anyone else.
- > **Shared grammars.** Shared grammars are grammars created by others and then made available to all speech mashup users. You can view (and copy) these files but cannot rename or delete them. Grammars you create and share with others are listed in both your personal grammars and under shared grammars.
- > **Built-in grammars** are included with the speech mashup and include grammars for US cities, digits, account numbers, and names. See Table 4.1 for a listing.

Grammars you create must be uploaded to the portal to be made accessible to the WATSON ASR servers; uploading a grammar automatically compiles it. You can upload grammars either through the portal or by using `wget`.

6.1 Creating application directories for grammars

Grammars you upload must be associated with an *application directory*, which serves as a central location for all grammars, log files, and audio associated with a specific application. You cannot upload a grammar until you create an application directory for it.

To create a new application directory, select **Manage applications** from the portal's home screen to open the following dialog:



From the Manage Applications screen, you edit, back up, or delete current application directories, or create new ones.

Use Backup option (the pie-shaped icon under “Actions”) to create a zipped file of all application directory contents.

To create a new application directory:

1. Click Create New from Manage applications screen.

If combining recognition with other processing (see page 74), enter the URL of servers to connect to.

Specifies how long to wait before considering a new utterance to be part of a new conversation. Default is 5 minutes.

2. Enter a unique name.
3. (Optional). Enter a short and long description. The short description is shown in the table of applications; the longer one is revealed by mousing over the application directory name.
4. Click Create.

Warning: Deleting an application directory deletes all grammars, log files, and audio stored in it.

6.2 Uploading grammars

You upload grammars in two ways: either from the portal or by using `wget`. In both cases, uploading the grammar automatically compiles it.

From the portal, you upload a grammar (or multiple grammars zipped together) as follows:

1. Select the appropriate application directory from the dropdown list.

Currently Selected Application : Restaurants

2. Select the **Manage Grammar Files** option from the main screen.
3. In the text box at the bottom of the screen, enter or browse for the filename. Click Submit Upload.

The screenshot shows the 'Manage Grammars and Dictionaries' interface. It features a top navigation bar with tabs for Applications, ASR, ASR Test, TTS Test, Logs, and Transcriptions. Below this, a 'Currently Selected Application' dropdown is set to 'Restaurants'. The main area is divided into two sections: 'Grammars' and 'Dictionaries'. The 'Grammars' section contains a table with columns 'Filename', 'Comp', 'Cmde', and 'Shared'. It lists 'burger-menu' (no, no, no) and 'pizza-grammar' (yes, no, no). Below the table are buttons for 'Upload', 'Delete', 'Rename', 'Edit', 'View', and a 'private' status dropdown. Further down are buttons for 'Compile', 'Watson Cmds', 'Share Grm Only', 'Share Grm & Src', and 'Unshare'. The 'Dictionaries' section has a 'Name' input field and buttons for 'Upload', 'Delete', 'Rename', and 'Edit'. At the bottom, there is a 'Grammar Details / Compilation Options / Compilation Log' section. It includes a 'Language' dropdown (English, Spanish), an 'Acoustic Model' dropdown (gentel04), and a 'Sel. Dicts' section with 'en-US' and 'en-US-its' options. A 'Save' button is also present. Annotations with orange lines point to various elements: 'Click to upload (& compile)' points to the 'Upload' button; 'Select the application directory in which to store the grammar' points to the 'Currently Selected Application' dropdown; 'Select My Grammars (private), Shared Grammars, or Builtin Grammars' points to the 'private' status dropdown; 'Change recognizer parameters, if needed. See page 37.' points to the 'Language' and 'Acoustic Model' dropdowns; and 'Change grammar compilation options' points to the 'Sel. Dicts' section.

Note: It's recommended that you keep a copy of each grammar on your local system. To back up an entire application directory, use the Backup option on the **Manage Applications** page.

The grammar will be automatically compiled when uploaded (if you're uploading a zipped file, the files will be unzipped first and then compiled).

A file that does not compile correctly is shown in red; the reason for the error is shown at the top of the screen. For more information about a compilation error, including the file line on which the error occurs, select the file and click **View Log File**.

6.2.1 Uploading grammars interactively

Rather than using the portal, you can upload grammars non-interactively, i.e., without having to be logged in. This can be useful for applications where grammars are generated by an automated process, and you want these grammars to be uploaded automatically by the same process that generated them.

To use this feature, use `wget` or some other tool that lets you send a file to an HTTP server using a POST request. With `wget`, the command to upload a grammar looks like this:

```
wget --post-file=your_grammar_source_file \
    --header 'Content-Type: text/plain' \
    --server-response \
    'https://service.research.att.com/smm/grammarUpload?username=YourUserName&
password=YourPassword&appname=YourApplication&filename=YourGrammarName'
```

The `--header` option is necessary because when `Content-Type` is unspecified, `wget` sets it to `application/x-www-form-urlencoded`, making the Tomcat server parse the request body as a form submittal—not what you want when trying to upload a file. The `grammarUpload` servlet ignores the `Content-Type`, so you can set it to whatever you want, as long as it isn't `application/x-www-form-urlencoded`.

The `--server-response` option makes `wget` print the response header on stdout; you want that because this is where `grammarUpload` reports success or failure.

If the compilation fails, the servlet adds “X-Compilation-Failure: Grammar compilation failed; see response body for details” to the response header, and it returns the compilation error log in the response body. Note that the HTTP status code is 200 in case of a compilation failure; while this is inconsistent, the reason is that `wget` refuses to download the response body if the HTTP status code signals an error, so in order to return the error log, the servlet has to return a status code of 200 and signal the error some other way.

When uploading a very large grammar, it is possible for the request to appear to fail, because `wget` or some proxy between `wget` and the server may decide to terminate a connection after it has been idle for a long time. Compiling large grammars can take a very long time; if this problem occurs, you can add `keepalive=true` to the request URL; the servlet will write a dot to the response every 10 seconds, thus keeping the connection alive indefinitely. The upload and compilation result will be reported in the response body instead of the response header in this case. You may want to add something like “-O response_file” to the `wget` command line to specify what file to save the response body to.

6.3 Sharing and managing grammars

You can share any grammar you create so that others can also use it. Any grammar you share is available to everybody else. (Future versions will support user groups so you can share with a specific group of users.)

To share a grammar, go to **Manage Grammars**, select the appropriate application directory and then the grammar (make sure the context **My Grammars** is selected), and assign it a title and a description (click **Update Description**); a grammar cannot be shared unless it has a title and description. Then click **Make Shared**. To unshare a grammar, select the grammar and click **Remove Shared**.

6.3.1 Editing prebuilt or shared grammars

When using a shared grammar (which can be unshared at any time), it's recommended that you copy the grammar into a text editor and then re-upload them as one of your personal grammars. Although personal grammars may have the same name as prebuilt or shared grammars (the recognizer looks first for personal grammars, then shared grammars, and then prebuilt ones), it's probably good practice to uniquely name each grammar.

Table 6.1 Builtin Grammars

citystate.xml	Contains all cities in the US. The source is not available.
en-us-time.wbnf	Returns time in the format of <i>hhmmx</i> , where <i>hh</i> is the hour, <i>mm</i> is the minutes, and <i>x</i> is one of the following: a (am) or p (pm), h (24-hour time), or ? (unknown).
en-us-helpcancel.wbnf	Returns help or cancel .
es-us-boolean.wbnf	Spanish boolean grammar.
es-us-phone.wbnf	Spanish phone number grammar that returns a string of 10 digits (area code followed by phone number) or 7 digits (phone number alone).
names2k.wbnf	A grammar containing two hundred names.
en-us-currency.wbnf	Currency grammar that returns a value formatted as <i>mmm.nn</i> where <i>mmm</i> is the number of dollars (3 digits or less). <i>nn</i> is the number of cents (always 2 digits). The standard specifies an optional currency type as a 3-character field preceding the amount, unless it is ambiguous. Does not include the currency type.
alphadigits.wbnf	7 alpha-digits grammar.
es-us-number.wbnf	Spanish numbers grammar that returns a series of digits optionally including a + or - sign, and/or a decimal point.
en-us-exit.wbnf	Grammar for recognizing the word exit .
en-us-helpcancel.wbnf	Spanish grammar for recognizing the words help and cancel .
digits_es.wbnf	Spanish digit loop grammar.
es-us-currency.wbnf	Spanish currency grammar that returns a value formatted as <i>mmm.nn</i> , <i>mmm</i> is the number of dollars and may be 3 digits or fewer. <i>nn</i> is the number of cents and is always 2 digits. The standard specifies an optional currency type as a 3-character field preceding the amount, unless it is ambiguous. This grammar does not include the currency type.
en-us-number.wbnf	Numbers grammar returns a series of digits optionally including a + or - sign, and/or a decimal point.
en-us-helpcancelexit.wbnf	Returns words help , cancel , and exit .

es-us-cancel.wbnf	Spanish grammar for recognizing cancel .
en-us-cancel.wbnf	Grammar for recognizing cancel .
es-us-help.wbnf	Spanish grammar for recognizing help .
numbers-es.wbnf	Spanish numbers grammar that returns a series of digits optionally including a + or - sign, and/or a decimal point.
en-us-boolean.wbnf	Boolean grammar.
en-us-date.wbnf	Date grammar. Returns an 8-digit number, <code>yyyymmdd</code> with ? for missing fields Example: july 4th: <code>????0704</code> , november: the first: january 1st 1970: <code>19700101</code>
es-us-helpexit.wbnf	Returns words help , cancel , and exit .

6.4 Determining accuracy

To evaluate how well speech recognition is performing for a specific application, you need to compare the recognized results returned from the WATSON recognizer to what was actually said. This is a three-step process:

1. Collect sample audio files and forward them to WATSON ASR for recognition (you can use `wget` or another tool). You will need to make recordings for this step. The more sample audio files, the better; several hundred are recommended.
2. Create transcriptions for each test utterance. You can use the portal's **View/enter transcriptions** to do so.
3. Compare the transcriptions to the recognizer outputs to determine the percentage of correct and incorrect recognitions.

Each step is described in more detail in the following sections.

6.4.1 Sending audio files for testing

To test how well the grammar can recognize utterances, create recordings of utterances that people might say in the context of the application. Send these audio files to the WATSON servers and then evaluate the word and sentence accuracy (the percentage of words and sentences that were correctly recognized). You might find when making recordings, people use different words than you anticipated and that may not be included in the grammar.

You can send audio files to WATSON ASR using a client if you have one already, or use the `wget` command to send audio. The `wget` command is standard in many modern UNIX and UNIX-like systems, including the Cygwin environment for Microsoft® Windows®. In case your system does not include `wget`, the source code is available at <http://www.gnu.org/software/wget/>, and executables for Microsoft Windows can be found at <http://pages.interlog.com/~tcharron/wgetwin.html>, and for Mac OS X at <http://www.statusq.org/archives/2005/02/22/610/>.

For testing a grammar, `wget` requires the following:

- > Name of audio file and content type. What you enter for the content type (audio/amr in the example starting on the next page) does not matter (though it should be descriptive), as long as it is not the wget default, which is application/x-www-form-urlencoded.
- > Your UUID.
- > The grammar name, without the extension. The grammar must already be uploaded to the portal and have compiled correctly.
- > The format type of the result (if other than XML).
- > (Optional) An output filename.

The following is a sample use of wget using one of the prebuilt grammars (en-us-date). To send multiple files, write a simple script. (You can find scripts containing for the following sample, as well as the sample-date.amr file used there, in the SpeechMashupGuide.zip file that you download by using the portal's Sample Code link. Note that you have to edit the script to put in your own UUID before you can run it.)

```
wget \
--post-file=sample-date.amr \
--header 'Content-Type: audio/amr' \
--server-response
'http://service.research.att.com/smm/watson?cmd=rawoneshot&grammar=en-us-
date&uuid=<your own UUID>&appname=<application ID>&resultFormat=emma' \
-O response.emma
```

The output produced onscreen will look similar to the following:

```
--16:16:08--
http://service.research.att.com/smm/watson?cmd=rawoneshot&grammar
=en-us-date&uuid=<your own UUID>&appname=<application
ID>&resultFormat=emma
=> `response.emma'
Resolving service.research.att.com... 192.20.225.56
Connecting to service.research.att.com|192.20.225.56|:80...
connected.
HTTP request sent, awaiting response...
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/xml;charset=UTF-8
Content-Length: 2138
Date: Thu, 31 Jul 2008 20:16:07 GMT
Connection: keep-alive
Set-Cookie: JSESSIONID=BC6CF8CBED6E65C4281FD8026BC75BBB;
domain=service.research.att.com; path=/smm
Length: 2,138 (2.1K) [application/xml]

100%[=====
=====>]
2,138      --.-K/s

16:16:08 (5.08 MB/s) - `response.emma' saved [2138/2138]
```

The returned output will be similar to the following (EMMA format):

```
<?xml version="1.0" encoding="UTF-8"?>
<emma:emma version="1.0"
```

```

xmlns:emma="http://www.w3.org/2003/04/emma"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2003/04/emma
http://www.w3.org/TR/WD-emma-20070409/emma.xsd"
xmlns="http://www.example.com/example">
<emma:grammar id="gram1"
  ref="smm:grammar=en-us-date&UID=[your_own_UUID]"/>
<emma:model id="model1"
  ref="smm:file=en-us-date.xsd&UID=[your_own_UUID]"/>
<emma:info>
  <uuid>[you_own_UUID]</uuid>
  <watson>
    <version>watson-6.1.3700</version>
    <time>2008-08-07 23:08:49.951</time>
    <session_id>20080807-230849-00000002</session_id>
    <hostname>ss-2</hostname>
  </watson>
</emma:info>
<emma:one-of id="one-of1"
  emma:medium="acoustic"
  emma:mode="voice"
  emma:function="dialog"
  emma:verbal="true"
  emma:lang="en-US"
  emma:start="1218164929760"
  emma:end="1218164933360"
  emma:grammar-ref="gram1"

emma:signal="smm:UID=[you_own_UUID]&file=/l/u205/speechmashu
ps/pino/def001/audio/20080807/audio-222478.amr"
  emma:signal-size="5766"
  emma:media-type="audio/amr; rate=8000"
  emma:source="smm:platform=null&device_id=null"
  emma:process="smm:type=asr&version=watson-6.1.3700"
  emma:duration="3600"
  emma:model-ref="model1"
  emma:dialog-turn="20080807-230849-00000002:1">
<emma:interpretation id="nbest1"
  emma:confidence="0.5"
  emma:tokens="July thirty first 2 thousand 8">
  <![CDATA[<$='????'+$m+$d> <$m> <$='07'> July </$='07'> </$m>
<$d> <$='31'> thirty first </$='31'> </$d> </$='????'+$m+$d>
<$=$y+$m+$d> <$y> <$=$2+'00'> <$2> <$='20'> 2 thousand </$='20'>
</$2> </$=$2+'00'> <$=$2+$c7> <$c7> <$='0'+$1> <$1> 8 </$1>
</$='0'+$1> </$c7> </$=$2+$c7> </$y> </$=$y+$m+$d> ]]>
</emma:interpretation>
</emma:one-of>
</emma:emma>

```

Note that you must specify a Content-Type request header (`--header 'Content-Type: audio/amr'` in the example), but it does not matter what value you set it to, as long as it is not `application/x-www-form-urlencoded`.

The reason is that that specific content type is handled specially by web servers; they assume that the request body contains data from an HTML form being submitted, and attempt to parse it accordingly, and the servlet or CGI program that processes the request never gets to see the raw request body. Since the default Content-Type header sent by `wget` is `application/x-www-form-urlencoded`, you must use `--header` to set it to

something else, but it does not have to actually match the type of audio you are sending; the speech mashup portal determines the audio format using the audioFormat request parameter or by inspecting the audio data itself, not using the Content-Type header.

6.4.2 Creating transcriptions

Creating transcriptions using the portal requires a recent version of the Java Runtime environment. Download the most recent version from <http://java.sun.com>.

To create a transcript that can later be compared to the recognized output:

1. Log into the portal and click **View/enter transcriptions**.
2. Navigate to the appropriate date (click **<<** or **>>**).
3. Click **▶** next to an audio file to listen to it.
4. Type the complete utterance into the transcription field (or copy the reco text if the match is exact).
5. Click a save button. This saves the transcription to a database for later retrieval.

The screenshot shows the 'Manage Transcriptions' web interface. The interface has a top navigation bar with tabs: Applications, ASR, ASR Test, TTS Test, Logs, and Transcriptions (which is active). Below the navigation bar, there is a section for 'Currently Selected Application : def001'. This section includes a 'Date' field (12/10/2008), a 'Batch' field (65 of 65), and an 'id' field. Below these fields are buttons for '<< Save', 'Save', 'Save >>', and '>>'. The main part of the interface is a table with columns: id, time, grammar, reco, length, gender, and transcription. The table contains one row with the following data: id: 282465, time: 23:54:23, grammar: /data1/fm/business.search.lm, reco: mexican restaurants, length: 3.5, gender: M, and transcription: Mexican restaurants. Below the table are buttons for '<< << Save', 'Save', 'Save >>', and '>>'. At the bottom of the interface, there is a section for 'Background Speech' with buttons for 'Empty', 'Foreign', 'Hengup', 'Human Noise', 'Non-Human Noise', 'Not Understandable', and 'Touchtone'. Below this section are buttons for 'Gender' (M, F, B) and 'Age' (A, C, B). Annotations with orange lines point to various parts of the interface: 'Enter new date' points to the 'Date' field; 'Session number (utterances by a single user, or a series of single-sentence utterances make up a batch)' points to the 'Batch' field; 'Enter ID of transcription when searching.' points to the 'id' field; 'Buttons for inserting annotations into transcription)' points to the 'Background Speech' buttons; 'Play bar for saving transcription or advancing/returning to other batches' points to the '<< << Save', 'Save', 'Save >>', and '>>' buttons; 'Play audio' points to the '▶' button next to the audio file; and 'Enter (or copy) transcription here' points to the 'transcription' field.

6.4.3 Comparing transcriptions to utterances

To obtain the accuracy rate, compare each actual utterance to the transcription created for it.

One way is to create two text files, one for the utterances captured from the recognized results (use the string from the reco field or the tokens string if using EMMA) and the other for the transcriptions, and then compare and score the files using a tool such as the Speech Recognition Scoring Package (SCORE) available from NIST (http://www.nist.gov/speech/tools/score_362tarZ.htm). This tool will return both the word and sentence accuracy.

6.4.4 Exporting transcriptions and utterances

Having created transcriptions for a set of utterances, you will probably need to export them so you can use them in downstream processes for tuning language models and evaluating application performance. The mashup portal provides a facility for this through the URL <http://service.research.att.com/smm/utterances.jsp>.

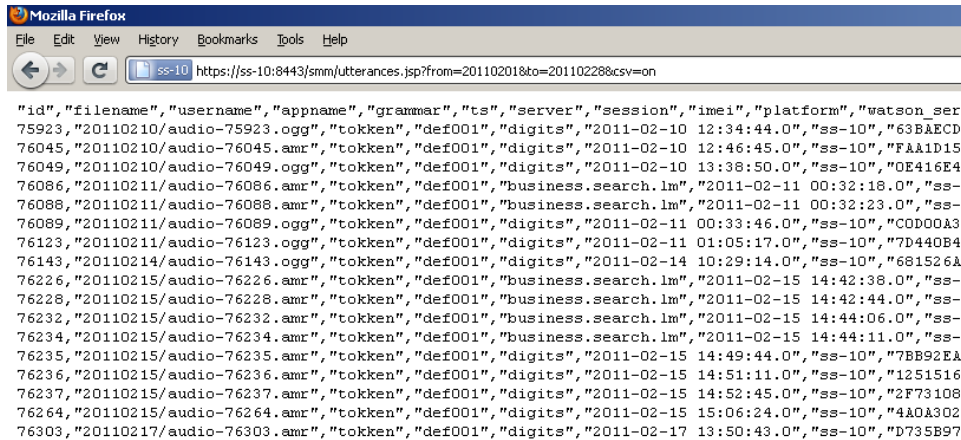
When you access this link while logged into the portal, you will be presented with the following page:

Use the “currently selected application” drop-down to select the application whose utterance records you want to export, and optionally specify the start and end dates. (Note that the end date is inclusive, so in order to dump the whole month of January, 2011, you would specify the start and end dates as shown in the screen shot, above.) Leaving the start date, or the end date, or both blank will cause the export not to be constrained in that direction, so you can create dumps of everything up to a certain date, everything from a certain date, or everything for all dates.

By default, the output generated by utterances.jsp is a pipe-delimited text file, that is, it contains one line for each utterance record, and each record’s individual fields are separated from each other using pipe characters (vertical bar, ASCII/Unicode 124). By checking the “generate CSV” check box, the output format is changed to comma-separated values, which can be easily imported into spreadsheet programs. In either case, the text is encoded using ISO-8859-1 (Latin-1) encoding.

```
id|filename|username|appname|grammar|ts|server|session|imei|platform|watson_server|watson_session|
75923|20110210/audio-75923.ogg|token|def001|digits|2011-02-10 12:34:44.0|ss-10|63BAECD6434E6C981
76045|20110210/audio-76045.amr|token|def001|digits|2011-02-10 12:46:45.0|ss-10|FAA1D1510A0E2463E
76049|20110210/audio-76049.ogg|token|def001|digits|2011-02-10 13:38:50.0|ss-10|0E416E4AB2E59FCF6
76086|20110211/audio-76086.amr|token|def001|business.search.lm|2011-02-11 00:32:18.0|ss-10|Sess-
76088|20110211/audio-76088.amr|token|def001|business.search.lm|2011-02-11 00:32:23.0|ss-10|Sess-
76089|20110211/audio-76089.ogg|token|def001|digits|2011-02-11 00:33:46.0|ss-10|COD00A38280A1444C
76123|20110211/audio-76123.ogg|token|def001|digits|2011-02-11 01:05:17.0|ss-10|7D440B46A82710EC1
76143|20110214/audio-76143.ogg|token|def001|digits|2011-02-14 10:29:14.0|ss-10|681526AAFB748FCDE
76226|20110215/audio-76226.amr|token|def001|business.search.lm|2011-02-15 14:42:38.0|ss-10|Sess-
76228|20110215/audio-76228.amr|token|def001|business.search.lm|2011-02-15 14:42:44.0|ss-10|Sess-
76232|20110215/audio-76232.amr|token|def001|business.search.lm|2011-02-15 14:44:06.0|ss-10|Sess-
76234|20110215/audio-76234.amr|token|def001|business.search.lm|2011-02-15 14:44:11.0|ss-10|Sess-
76235|20110215/audio-76235.amr|token|def001|digits|2011-02-15 14:49:44.0|ss-10|7BB92EA5C552579B7
76236|20110215/audio-76236.amr|token|def001|digits|2011-02-15 14:51:11.0|ss-10|12515163684452B62
76237|20110215/audio-76237.amr|token|def001|digits|2011-02-15 14:52:45.0|ss-10|2F73108DB1C8AB83C
76264|20110215/audio-76264.amr|token|def001|digits|2011-02-15 15:06:24.0|ss-10|4A0A3026D4592AD97
76303|20110217/audio-76303.amr|token|def001|digits|2011-02-17 13:50:43.0|ss-10|D735B975B1E6C4F65
```

pipe-delimited utterance records dump



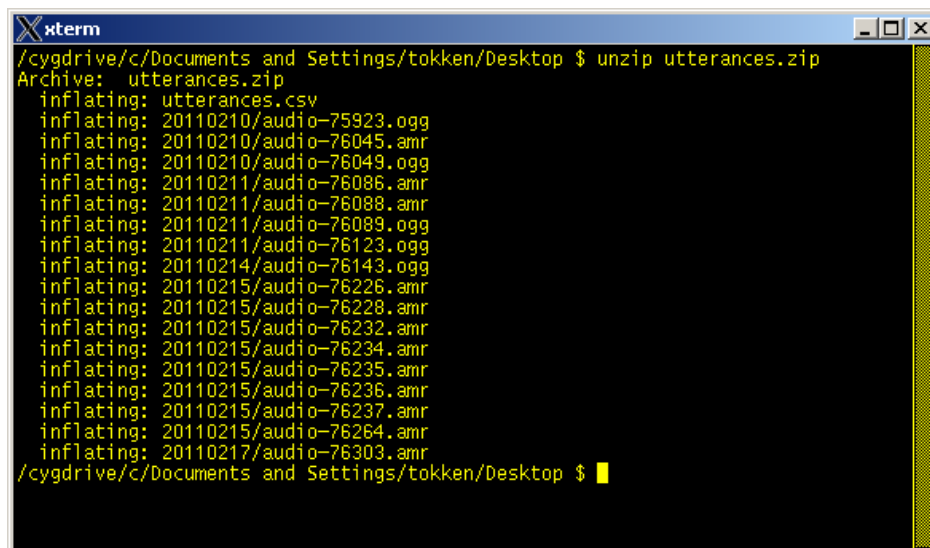
```

"id","filename","username","appname","grammar","ts","server","session","imei","platform","watson_serv
75923","20110210/audio-75923.ogg","tokken","def001","digits","2011-02-10 12:34:44.0","ss-10","63BÆCDe
76045","20110210/audio-76045.amr","tokken","def001","digits","2011-02-10 12:46:45.0","ss-10","FAA1D15J
76049","20110210/audio-76049.ogg","tokken","def001","digits","2011-02-10 13:38:50.0","ss-10","0E416E4J
76086","20110211/audio-76086.amr","tokken","def001","business.search.lm","2011-02-11 00:32:18.0","ss-1
76088","20110211/audio-76088.amr","tokken","def001","business.search.lm","2011-02-11 00:32:23.0","ss-1
76089","20110211/audio-76089.ogg","tokken","def001","digits","2011-02-11 00:33:46.0","ss-10","C0D00A3E
76123","20110211/audio-76123.ogg","tokken","def001","digits","2011-02-11 01:05:17.0","ss-10","7D440B4E
76143","20110214/audio-76143.ogg","tokken","def001","digits","2011-02-14 10:29:14.0","ss-10","681526AJ
76226","20110215/audio-76226.amr","tokken","def001","business.search.lm","2011-02-15 14:42:38.0","ss-1
76228","20110215/audio-76228.amr","tokken","def001","business.search.lm","2011-02-15 14:42:44.0","ss-1
76232","20110215/audio-76232.amr","tokken","def001","business.search.lm","2011-02-15 14:44:06.0","ss-1
76234","20110215/audio-76234.amr","tokken","def001","business.search.lm","2011-02-15 14:44:11.0","ss-1
76235","20110215/audio-76235.amr","tokken","def001","digits","2011-02-15 14:49:44.0","ss-10","7BB92EAE
76236","20110215/audio-76236.amr","tokken","def001","digits","2011-02-15 14:51:11.0","ss-10","12151516
76237","20110215/audio-76237.amr","tokken","def001","digits","2011-02-15 14:52:45.0","ss-10","2F73108I
76264","20110215/audio-76264.amr","tokken","def001","digits","2011-02-15 15:06:24.0","ss-10","4A0A302E
76303","20110217/audio-76303.amr","tokken","def001","digits","2011-02-17 13:50:43.0","ss-10","D735B97E

```

CSV utterance records dump

The utterances.jsp page will usually be used to obtain the recognition and transcriptions for recorded utterances, and possibly other meta-data that is stored in the utterance records, but it also lets you dump the utterances themselves, that is, the audio files, in addition to the utterance database records. To do this, check the “Create ZIP file containing audio” check box; this will cause your browser to download a ZIP file containing the dump of the utterance records, plus all the corresponding audio files.



```

Xterm
/cygdrive/c/Documents and Settings/tokken/Desktop $ unzip utterances.zip
Archive:  utterances.zip
  inflating: utterances.csv
  inflating: 20110210/audio-75923.ogg
  inflating: 20110210/audio-76045.amr
  inflating: 20110210/audio-76049.ogg
  inflating: 20110211/audio-76086.amr
  inflating: 20110211/audio-76088.amr
  inflating: 20110211/audio-76089.ogg
  inflating: 20110211/audio-76123.ogg
  inflating: 20110214/audio-76143.ogg
  inflating: 20110215/audio-76226.amr
  inflating: 20110215/audio-76228.amr
  inflating: 20110215/audio-76232.amr
  inflating: 20110215/audio-76234.amr
  inflating: 20110215/audio-76235.amr
  inflating: 20110215/audio-76236.amr
  inflating: 20110215/audio-76237.amr
  inflating: 20110215/audio-76264.amr
  inflating: 20110217/audio-76303.amr
/cygdrive/c/Documents and Settings/tokken/Desktop $

```

Combined utterance records and audio files dump

You may need to make these utterance dumps a part of a regular batch process, in which case having to log into a web interface and manually filling out and submitting a form would be inconvenient. To help with this situation, utterances.jsp also supports being invoked outside of the portal context, using a tool like wget:

```
wget -O utterances.zip \
  'https://service.research.att.com/smm/utterances.jsp
  ?username=your-username
  &password=your-password
  &appname=selected-appname
  &from=start-date-YYYYMMDD
  &to=end-date-YYYYMMDD
  &csv=(on|off)
  &audio=(on|off) '
```

Note that there are line breaks in the URL in the example above that were inserted for readability only; when you create the request URL, everything should be in one line with no intervening white space.

Note that you must use HTTPS, not HTTP; the portal will automatically send a redirect if you try to access utterances.jsp over HTTP, but you have to use HTTPS from the start in order to prevent your password from being sent in the clear.

The username and password parameters should be set to the same values that you use to log into the portal; all the other parameters correspond to the fields in the form that utterances.jsp displays in its interactive mode.

6.5 Checklist for improving accuracy

If your results are not good because many no-matches are being reported or many words are not being recognized, try one or more of the following solutions:

- > Check that unrecognized words are contained in the grammar.
- > Adjust the word penalty. For errors caused because the recognizer is inserting too many words that are not there, increase the penalty. If the recognizer is missing words that are there, decrease the penalty.

You adjust the word (and silence) penalties using define statements in a rule-based grammar. See page 12. If adjusting the word penalty doesn't work, try adjusting the silence penalty.

- > (XML or SLM only) If the recognizer is substituting a referenced word with a different word, use word weighting to favor or discourage individual words. Insert weight attributes (floating-point values) within nested <item> tags as shown here:

```
<item>
  <one-of>
    <item weight="0.8"> New York </item>
    <item weight="0.2"> Newark </item>
  </one-of>
</item>
```

- > Use pron tags if appropriate.

Pronunciation tags should be used if the grammar includes digits, names, quantities (whole numbers), spellings, and short confirmations.

See “Adding pronunciation tags to an XML grammar” (page 15) or “Adding pronunciation tags to a WBNF grammar” (page 20) as appropriate.

- > In the grammar, remove or insert disfluencies.
Disfluencies, which are the natural irregularities and hesitations in the smooth flow of speech, are often accounted for in a grammar since they are a part of spoken language. Adding a rule for disfluencies often improves recognition; normally one rule for disfluencies at the beginning of a grammar is enough. However, in some cases, removing disfluencies can simplify a grammar, making it more efficient and increasing recognition.
- > Check the audio files for audibility. There may be a problem with the audio rather than with the recognition.
- > Some voices are simply hard to recognize, particularly those that are mumbled or heavily accented. It might not be possible to get good results with some voices.

Any time you update the grammar, re-upload the file for the changes to take effect.

6.6 Setting recognizer preferences with a commands file

Depending on your grammar, adjusting recognizer parameters can improve performance. Currently, you can adjust the following parameters for each grammar.

- > Speed vs accuracy, which controls the tradeoff between fast processing and accurate recognition. Better accuracy requires more CPU but at a cost of slower processing speed. A value of **0** is the fastest, and **1.0** the most accurate. The default is **.5**.
- > Sensitivity, which controls how sensitive the recognizer is when determining that audio is speech. Use a lower value for noisy environments and a higher one for low-noise situations. The default is **50**, and the supported range is **1-100**.
- > Number of results returned. By default the recognizer returns a single result. However, in some cases (such as when the result is cross-checked against a database), you may want more than one result to increase the chances that the correct response is returned.

You set recognizer parameters by creating a WATSON commands file from the **Manage Grammars** dialog (select WATSON commands file from the dialog).

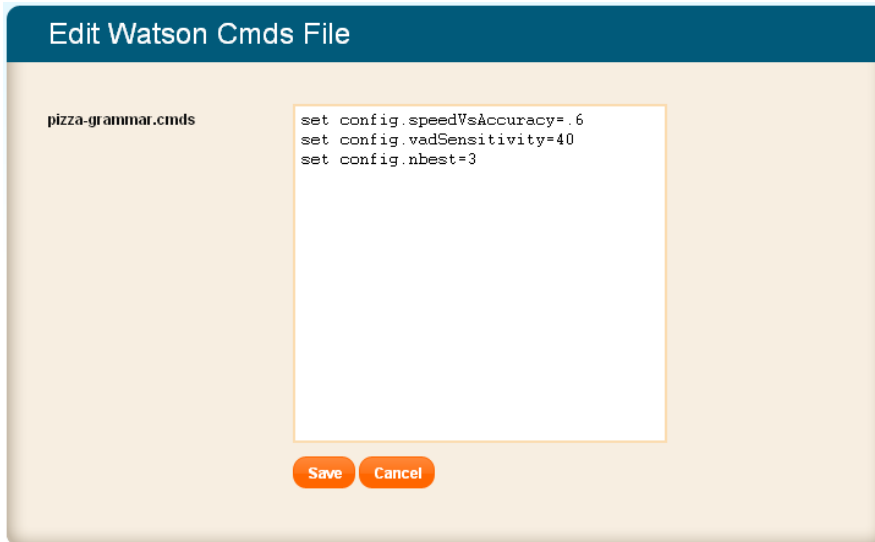
Each command uses the following syntax:

```
set name=value
```

where *name* can be one of the following parameters and *value* is a value appropriate to the parameter.

<code>config.speedVsAccuracy</code>	Value range: 0.0 – 1.0 (default is .5)
-------------------------------------	--

<code>config.vadSensitivity</code>	Value range: 1-100 (default is 50)
<code>config.nbest</code>	Value range: n (default is 1)



Note: These recognizer parameters are the same that can be set through the REST API control parameter (see page 54). Note that the REST control parameters override the settings contained within the WATSON commands file.

6.6.1 Preferences for automatic endpointing

In many usage scenarios, it is desirable to let the ASR engine itself decide when the user starts and finishes speaking, rather than requiring the user to signal these events by pressing a button.

WATSON supports automatic endpointing through its “timeout” parameters. To configure a grammar to be used with endpointing, add the following statements to the grammar’s associated *.cmds file:

```
activateEvh "timeouts"
activateEvh "speechstart-hmm"
timeouts.firstTimeout = 400
timeouts.secondTimeout = 500
```

After *firstTimeout* milliseconds of silence after speech start, WATSON will generate a result, and if the result’s confidence score is above the recognition threshold, it will return this result to the client; if the score is below the threshold, it will continue processing audio.

After *secondTimeout* milliseconds of silence after speech start, WATSON will generate a result and return it to the client, even if its confidence score is below the recognition threshold.

It is possible to specify the timeout parameters in the ASR request instead of using the *.cmds file; this can be convenient for testing, when the ability to quickly tweak parameters is important. To do this, you would add the parameter setting statements using the mashup’s *control* parameter, by adding the following to the request URL:

```
...&control=activateEvh+%22timeouts%22%3BactivateEvh+%22speechstart-  
hmm%22%3Btimeouts.firstTimeout+=+400%3Btimeouts.secondTimeout+=+500
```

(Note: the above is URL-encoded, so the + signs represent spaces, %22 represents the double quote character, and %3B represents the semicolon, used for separating statements.)

This mechanism can also be used to override settings from the *.cmds file.

7 Modifying Output Speech for Natural Voices

AT&T's Natural Voices converts text to speech using a large sampling of previously recorded sound segments that it combines on the fly to form any word or phrase. There are four main steps to converting text to speech:

- > Normalizing the text
Text contains more than just words. Non-words—abbreviations & acronyms, numbers, symbols (\$, &, /)—need to be interpreted or fully expanded into words. For example, does Dr. mean drive or doctor; does “1/2” mean “one-half,” “January 2”, or something else (page “1 of 2”)? The context can resolve this problem.
- > Retrieving the pronunciation for each word.
Natural Voices provides a large, general English dictionary from which it retrieves phonetic spellings.
- > Combining sound segments.
The phonemes that make up a word are matched to sound segments, which are then combined into natural sounding phrases.
- > Synthesizing the speech.
The sound segments combined in the previous step are converted to a speech signal using the specified language, voice, and speaking rate.

The way in which the input text sounds is based on default rules applied by the Natural Voices server at each step. However, you can apply different rules to change the way in which the text is normalized, or modify the prosody (the intonations, rhythm, pauses, of spoken speech) or pronunciation of individual words.

To modify the speech, do one (or both) of the following:

- > Spell out numbers and abbreviations to get the exact pronunciation.
For example, for “1/2” to be pronounced “one of two” (instead of as one half), type out the words rather than using numbers).
- > Insert SSML tags as described in the next section. SSML tags can modify the normalization, prosody, and pronunciation.

7.1 Using SSML tags

SSML is a standardized XML markup language for modifying the way text is processed by TTS engines. The SSML tags are instructions for normalizing text and controlling emphasis and other speaking qualities (prosody). The tags, which are inserted into the text, either manually or automatically by an application, do the following:

- > Normalize text by interpreting symbols, abbreviations, and acronyms according to context.
- > Substitute a word for another. For instance, you may want all instances of “America” to be replaced with “United States”.
- > Resolve ambiguities. Is “lead” to be pronounced “led” or “lead”?
- > Achieve natural phrasing by applying stress, intonation, and pitch where appropriate, or inserting pauses.
- > Change the volume and speaking rate.

Tags are not always the best choice. Punctuation (commas and periods) can usually be inserted more quickly and easily than SSML tags to create pauses. In fact, whenever possible, use punctuation (commas and periods) for pauses, or use text formatting for text contained in tables.

SSML tags can be inserted into the text, or you can set up a preprocessing step where the tags are inserted automatically by a program running on a different server.

See Table 5.1 for a quick listing of SSML tags. See For detailed information about XML and SSML, see [World Wide Web Consortium](#) (W3C).

7.1.1 SSML syntax

SSML tags must be set off with an opening and closing tag as shown in this example, in which the emphasis tag makes “this” be pronounced louder than surrounding words.

Make `<emphasis> this </emphasis>` more prominent.

A small number of empty tags don’t require both an opening and closing tag (such as the `<break/>` tag, which inserts a space between two words).

Some SSML tags take an attribute and a value for even more control. For example, you can control the length of a pause:

Begin `<Break time="100ms"/>` now. (or Begin `<Break time="3s"/>` now.)

Table 7.1 SSML tags

<say-as> tags <i>value can be:</i>	<say as> tags provide hints on which text expansions are more likely given the context. Syntax is: <code><say-as interpret-as="value"> text </say-as></code>
"acronym"	Pronounces each letter individually. Useful for acronyms embedded in all uppercase text. Same as <code>spell</code> tag.
"address"	Identifies text as an address. Pauses are inserted between address components, and building numbers are spoken in pairs, e.g., <i>1250 Elm</i> is read <i>twelve fifty elm</i> .
"currency"	US currency only. Expands \$ and decimal numbers appropriately.
"date"	Treats the text as a date. The US English default is "mdy" (with slashes); use format attribute to change to "dmy". <code><say-as interpret-as="date" format="dmy"> 1/2/2008 </say-as></code>
"ignore-case"	Ignores capitalization of words. Useful for all-uppercase input (the default is to spell out all-uppercase words).
"lines"	Treats end-of-line as end-of-sentence, so each line is read separately. Useful for lists, tables, and poems, and provides an alternative to using <code><s></code> to force sentence breaks.
"literal"	Passes the string through literally (as is), exactly as typed and without expansion (<i>Main St.</i> would be pronounced <i>Main s t</i>).
"math"	Treats text as a mathematical expression, correctly interpreting plus, minus, and division signs.
"measurement"	Treats text as a measurement, interpreting single quotes as "feet" and double quotes as "inches."
"number"	Specifies a fraction (using fraction attribute) or separately pronounces each digit (using digits attribute). <code><say-as interpret-as="number" format="fraction digits"> text </say-as></code>
"spell"	Reads characters & digits separately (<i>Hello</i> , becomes <i>H E L L O</i>). Same as <code>acronym</code> .
"telephone"	Treats input number as a telephone number, speaking each digit separately and applying prosodic phrasing to group digits.
"time"	Pronounces time as hours, minutes, or seconds.
Structure tags	Structure tags consists of opening and closing tags surrounding text being synthesized: <code><tag-name> text </tag-name></code>
< speak >	Identifies the language to use. <code><speak xml:lang="en_us"></code> , which specifies US English. Other options are en_uk for UK English, en_es for the US dialect of Spanish, and es_es for European Spanish.
< voice >	Identifies specific voice to use. <code><voice name="mike"> text </voice></code> <code><voice name="cyrstal"> text </voice></code> Can also identify a language: <code><voice xml:lang="es_us"> 1 2 3 </voice></code>
< paragraph >	(or <code><p></code>). Signals start and end of a paragraph without regard to other formatting hints. Use when the start and end of a paragraph are not clear. Can also change language: <code><p xml:lang="en_us"></code>
< sentence >	(or <code><s></code>). Sets off text that is to be read in sentence structure. Can also change language: <code><s xml:lang="es_us"></code>
< mark / >	Causes a bookmark notification to be sent. <code><mark name="bm1"/></code> causes the client to receive a bookmark notification with the text "bm1". See section 7.4 for details.
Prosody tags	Prosody tags (except for <code><break/></code>) consist of an opening and closing tag.

Table 7.1 SSML tags

<code><break/></code>	<p>Inserts a pause of the specified length. Use with the strength or time attribute.</p> <p><code><break strength="level"/></code> where <i>level</i> can be none, x-weak, weak, medium, strong, or x-strong</p> <p><code><break time="ns nms"/></code> where <i>n</i> is the number of seconds or milliseconds</p> <p>For slight pauses with no intonation change, use the silence phoneme (<code><phoneme alphabet="darpa" ph="pau"/></code>).</p>
<code><prosody volume=level></code>	<p>Adjusts loudness of the audio output (<code><prosody volume="level n n%"> text</prosody></code>):</p> <p><code><prosody volume=level></code> where <i>level</i> can be silent, x-soft, soft, medium, loud, x-loud, default (return to normal)</p> <p><code><prosody volume=n></code> or <i>n</i> can be a number between 1 and 100 (or above) for a relative change from current volume.</p> <p><code><prosody volume=n%></code> A minus or plus makes the change in volume a variation from the default, not current, volume</p>
<code><prosody rate=value></code>	<p>Varies the speaking rate to be faster or slower (<code><prosody rate="value"> text</prosody></code>):</p> <p>where <i>value</i> can be x-fast, fast, medium, slow, x-slow, or default (to return to a normal rate). <i>value</i> can also be a number, where .5 is half the default rate, 2 is double the default rate, and values such as .8 and 1.2 are somewhere in between.</p>
<code><emphasis level="level"></code>	<p>Applies more or less emphasis (<code><emphasis level="level"> text</emphasis></code>)</p> <p>where <i>level</i> can be strong, moderate, none, or reduced.</p>
Phoneme tag	<p>Specifies a particular pronunciation for a single instance of a word. No part-of-speech rules are applied.</p> <p><code><phoneme alphabet="darpa" ph="f aa 1 dh er 0"/></code>.</p> <p>For a listing of the phonemes, see table 5.2.</p>

7.2 Changing word pronunciations

Natural Voices includes a standard dictionary that provides pronunciations for common English words. However, some words, especially technical terms, jargon, or regional pronunciations, may not be included in the provided dictionary.

When a word is not included or when you prefer a different pronunciation than the one provided, you can use the `<phoneme>` tag to specify the exact pronunciation using the phonemic spelling.

Natural Voices uses the DARPAbet phoneme set as shown in table 5.2.

Table 7.2 DARPAbet phoneme set

Phoneme	Example	Transcription	More Examples
aa	B <u>o</u> b	/b <u>aa</u> b 1/	knot
ae	b <u>a</u> t	/b <u>ae</u> t 1/	bad, gnat
ah	b <u>u</u> t	/b <u>ah</u> t 1/	cub, tuck, bud
ao	b <u>o</u> ught	/b <u>ao</u> t 1/	saw, caught
aw	d <u>o</u> wn	/d <u>aw</u> n 1/	about, how
ax	<u>a</u> bout	/ax 0 b aw t 1/	on <u>io</u> n, <u>u</u> pon
ay	b <u>i</u> te	/b <u>ay</u> t 1/	high, psy <u>ch</u> o
b	<u>b</u> et	/b <u>eh</u> t 1/	
ch	<u>ch</u> urch	/ch er <u>ch</u> 1/	
d	<u>d</u> ig	/d ih g 1/	
dh	<u>th</u> at	/dh ae t 1/	
eh	b <u>e</u> t	/b <u>eh</u> t 1/	mess, led
er	b <u>i</u> rd	/b <u>er</u> d 1/	curb
ey	b <u>a</u> it	/b <u>ey</u> t 1/	laid, eight
f	f <u>o</u> g	/f ao g 1/	phone
g	<u>g</u> ot	/g aa t 1/	
hh	<u>h</u> ot	/hh aa t 1/	
ih	b <u>i</u> t	/b <u>ih</u> t 1/	hip, in
iy	b <u>e</u> at	/b <u>iy</u> t 1/	heap, teal
jh	<u>j</u> ump	/jh ah m p 1/	
k	<u>c</u> at	/k ae t 1/	kick
l	<u>l</u> ot	/l aa t 1/	
m	<u>m</u> om	/m aa m 1/	
n	<u>n</u> od	/n aa d 1/	noun
ng	s <u>ing</u>	/s ih <u>ng</u> 1/	
ow	b <u>o</u> at	/b <u>ow</u> t 1/	sew, coal
oy	b <u>o</u> y	/b <u>oy</u> 1/	foil
p	<u>p</u> ot	/p aa t 1/	
r	<u>r</u> at	/r ae t 1/	
s	<u>s</u> it	/s ih t 1/	
sh	<u>sh</u> ut	/sh ah t 1/	
t	<u>t</u> op	/t aa p 1/	
th	<u>th</u> ick	/th ih k 1/	
uh	b <u>o</u> ok	/b <u>uh</u> k 1/	full
uw	b <u>o</u> ot	/b <u>uw</u> t 1/	fool, cr <u>u</u> de, do
v	<u>v</u> at	/v ae t 1/	
w	<u>w</u> on	/w ah n 1/	
y	<u>y</u> ou	/y uw 1/	

z	zoo	/z uw 1/	xylophone
zh	mea <u>s</u> ure	/m eh zh 1 er 0/	

7.3 Testing the TTS conversion

The TTS functionality can be accessed via the `/smm/tts` servlet, which takes the following parameters:

<code>uuid</code>	The UUID for authenticating with the mashup server.
<code>appname</code>	The application name under which the server will write log messages.
<code>text</code>	Text to speak. Note: Text may also be supplied as the body of a POST request (see below).
<code>audioFormat</code>	Format for the returned digital audio. Possible values are amr (AMR narrow-band), mulaw (AU with μ -law encoding), alaw (AU with A-law encoding), and linear (AU with 16-bit linear encoding). The default is amr .
<code>sampleRate</code>	Desired sample rate for the returned digital audio. The default is 8000 Hz. Note that AMR-NB has a fixed sample rate of 8000 Hz, so specifying a different <code>sampleRate</code> will produce odd-sounding results.
<code>volume</code>	The volume of the generated audio. The valid range for this parameter is 0 to 500; the default value is 100.
<code>voice</code>	Voice to use. Currently there are four options: mike (male, English), crystal (female, English), alberto (male, Spanish), and rosa (female, Spanish). The default is crystal .
<code>ssml</code>	Specifies whether or not the text to speak contains SSML mark-up; this may be true or false . The default is false .
<code>notifications</code>	Specifies which types of notifications should be returned, and optionally also specifies the desired audio/bookmark container format. See section 7.4 for details. The notifications can be any combination of bookmark , phoneme , viseme , and word . In addition, the container format may optionally be specified using one of simple , ogg , or multipart , with simple being the default. The individual parts of this parameter are separated by commas. For example, to select bookmark and word notifications, with the ogg container format, use notifications=bookmark,word,ogg .

You can exercise this API using `wget`, like this:

```
wget -O output_file
'http://<SMM_SERVER>/smm/tts?text=Hello+world&audioFormat=mulaw'
```

If the text to be spoken is long or contains characters that have to be percent-encoded in a URL

(i.e., anything other than the letters a-z and A-Z, digits 0-9, and hyphen, period, underscore, and tilde), it is probably more convenient to send the text using an HTTP POST request instead of using a GET with the **text** parameter; here's how to do this using `wget`:

```
wget -O output_file --post-file=text_file --header 'Content-Type: text/plain' 'http://<SMM_SERVER>/smm/tts?audioFormat=mulaw'
```

Note: The `--header` option is necessary because when Content-Type is unspecified, `wget` sets it to `application/x-www-form-urlencoded`, and that makes the Tomcat server parse the request body as a form submittal, which is not what you want when you're trying to upload a file. The `tts` servlet ignores the Content-Type, so you can set it to whatever you want, as long as it isn't `application/x-www-form-urlencoded`. When sending text in an encoding other than ISO-8859-1, you should use a content type that specifies the encoding, e.g., `"text/plain; charset=UTF-8"` or `"application/xml; charset=ISO-8859-5"`.

7.4 TTS With Bookmarks / Notifications

7.4.1 Introduction

In addition to the synthesized speech itself, the NaturalVoices TTS engine can generate notification events that provide real-time information about the progress of the speech.

One type of notification are *bookmarks*, generated using the SSML `<mark/>` tag. These notifications can be inserted in the text wherever necessary, and can be used, for example, to tell an e-book reader when to scroll the text (by inserting a bookmark at the end of each line) or turn to the next page (by inserting a bookmark at each page break).

The other types of notification are *phoneme*, *viseme*, and *word* notifications. These inform the client about the individual phonemes, visemes (lip movements), and words that are being spoken. **NOTE: phoneme and viseme notifications are currently only available to AT&T personnel.**

The format of a phoneme notification is `"NOTIFICATION:PHONEME:phoneme:duration:stress"`; the format of a viseme notification is `"NOTIFICATION:VISEME:next-id:current-id:duration:feature"`; the format of a word notification is `"NOTIFICATION:WORD:speak-handle:character-offset:word-length:sentence-length:flags:part-of-speech"`. Please refer to the NaturalVoices manual for more detailed information.

The format of a bookmark notification is simply the string specified in the `<mark>` tag's *name* attribute.

The mashup server delivers all these notifications over the same channel as the audio. Since most audio formats do not support meta-data embedded at arbitrary points inside the audio stream, the mashup server needs to use a higher-level "container" stream for combining audio and notifications. This chapter describes the supported container formats.

The formats described in this section are designed to allow interleaving of streaming digital audio with some type of metadata, such as bookmarks for text-to-speech (TTS) applications, or gestures for multi-modal applications combining automatic speech recognition (ASR) with other input modes. The current reference implementations were written for a TTS-with-bookmarks demo, and some of the language reflects this, but

other types of metadata can be transmitted using the same formats by defining a different structure for the metadata. The code that generates and parses the interleaved audio/metadata streams is agnostic of this structure and treats the metadata as a generic byte stream.

7.4.2 Simple Format

The “Simple” format was designed to allow interleaving of audio and metadata with minimal overhead, in terms of CPU usage, latency, and network bandwidth requirements. Its main drawback is that it is not compatible with any established standards, so all tools to inspect or manipulate it need to be written from scratch.

When transmitted over HTTP, a Simple stream should be identified using Content-Type: application/x-bookmarked-audio (TBD – this is what the current SimpleContainerOutputStream implementation uses, but should this be changed to something more generic, e.g. application/x-audio-with-interleaved-metadata?).

A Simple stream starts with the four bytes 0x41, 0x4D, 0x53, 0x21, representing the four-character string “AMS!” in ASCII encoding. This is followed by zero or more chunks, whose layout is defined below. There is no end-of-stream marker; readers are expected to read Simple streams, chunk by chunk, until they encounter end-of-file, or until some point identified by the transport protocol, e.g. the optional Content-Length header in HTTP messages.

A chunk consists of a two-byte header, followed by the chunk data. The two header bytes should be interpreted as a 16-bit integer in network byte order (most significant byte first). The two most significant bits of this integer indicate the chunk type; currently defined are 00=audio, 01=bookmark, 10=reserved, 11=reserved; the remaining 14 bits should be interpreted as an unsigned integer indicating the number of chunk data bytes that follow.

A reader can reconstruct the audio stream by simply concatenating the chunk data from all audio (type 00) chunks.

Bookmark (type 01) chunks have bodies that consist of a four-byte time stamp, followed by a text string in ISO-8859-1 encoding. (Other types of data can of course also be sent by simply defining a different encoding for the bookmark text.) The time stamp is to be interpreted as a signed integer in network byte order, representing a number of milliseconds from the beginning of the audio stream.

Bookmarks are encoded one per chunk, so even if two bookmarks are to occur simultaneously, they must still be sent in two separate chunks, although this limitation can be worked around by defining application-specific encodings.

Example:

The following is a “simple” stream generated for the text “Hello, world.”, converted to speech with bookmarks at the beginning, between the two words, and at the end. The speech is encoded using AMR-NB encoding with the highest bit rate. The chunk headers are highlighted in the dump listing.

The stream is identified by the “AMS!” magic number:

```
00000000 41 4d 53 21                                AMS!
```

The first bookmark:

(Note that the timestamp is 0x00000000, or 0 ms from the start of audio.)

```
00000000      40 12 00 00 00 00 66 69 72 73 74 20      @.....first
00000010 62 6f 6f 6b 6d 61 72 6b      bookmark
```

Several audio chunks:

```
00000010      00 26 23 21 41 4d 52 0a      .&#!AMR.
00000020 3c 91 17 16 be 66 78 00 00 01 e7 af 00 00 00 00 <...>fx...g/....
00000030 80 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000040 00 20 3c 48 77 24 96 66 78 00 00 01 e7 ba 00 00 . <Hw$.fx...g:...
00000050 00 00 c0 00 00 00 00 00 00 00 00 00 00 00 ..@.....
00000060 00 00 00 20 3c 5b 02 9d 86 ba 8a 00 10 02 0c 3f ... <[.....?
00000070 19 90 88 2a 9a 52 65 96 00 1b fd f5 2d 13 2c b9 ...*.Re...}u-.,9
00000080 20 1e 97 f0 00 20 3c e0 54 2c 93 7b 38 ae a0 92 ..p. <`T,.{8. .
00000090 c0 bb 6a 8e 78 bb bf ff f9 30 ba 02 65 fc 7f fe @;j.x;?y0::e|.~
000000a0 f1 7f c4 39 97 00 00 20 3c aa ea 75 14 a9 79 ff q.D9... <*ju.)y.
000000b0 e1 fd 30 e5 f5 14 bc fe 1f 1b 4f e9 af c3 7d 4b a}0eu.<~.0i/C}K
000000c0 9a e9 51 a7 d3 dc 7a d0 .iQ'S\zP
```

The second bookmark:

(Note that the timestamp is 0x0000021c, 540 ms from the start of audio, while only 100 ms of audio have been sent so far. Meta-data, whether bookmarks or gestures, does not have to be aligned with the audio stream; bookmark packets should typically appear early so that a TTS playback client can prepare to show them on time, while gesture packets may have to appear late, if the gesture information provided by the operating system has more latency than the audio stream.)

```
000000c0      40 13 00 00 02 1c 73 65      @.....se
000000d0 63 6f 6e 64 20 62 6f 6f 6b 6d 61 72 6b      cond bookmark
More audio:
000000d0      00 20 3c      . <
000000e0 48 7e 6b b2 b8 3e 03 69 73 85 38 3b 0a 5e c0 99 H~k28>.is.8;.^@.
000000f0 29 c5 5e 3e 29 aa e5 d8 f5 2e 9b 10 70 99 f0 00 )E^>)*eXu...p.p.
00000100 20 3c 09 7f c6 60 7f b8 67 e7 9d e1 b5 7d bb b5 <..F`.8gg.a5};5
00000110 42 11 fc 19 1d 26 d6 e7 a2 c4 ca d5 19 ca d4 6b B.|...&Vg"DJU.JTk
00000120 40 00 20 3c 54 4c 1d e9 83 5d f0 56 89 86 68 00 @. <TL.i.]pV..h.
00000130 6e 59 d6 57 4e fa 20 80 09 00 8c 31 7e 62 e1 24 nYVWNz ....1~ba$
00000140 ba 4f e0 00 20 3c 20 44 02 09 87 20 79 81 fc d4 :O`. < D... y.|T
00000150 29 7d 71 d2 e8 a8 2c 83 3a 01 34 44 60 18 6e 4b )}qRh(,...4D`.nK
00000160 70 78 18 5c f0      px.\p
```

...and so on.

Note that most of the audio chunks in this example are 32 bytes long, which corresponds to one 20 ms AMR-NB frame. Using a larger buffer, these chunks could be combined into larger ones, which would reduce the transmission overhead caused by the chunk headers, but would also introduce additional latency because the sender would have to queue up more audio before being able to flush the buffer. The trade-off between bandwidth and latency is one that needs to be determined on an application-by-application basis.

7.4.3 Ogg Format

The Ogg format for audio/metadata interleaving is documented in the AT&T Internal Addendum to this document.

7.4.4 MIME Multipart/Mixed Format

The MIME Multipart/Mixed format for audio/metadata interleaving is documented in the AT&T Internal Addendum to this document.

8 Building a Speech Mashup Client

A speech mashup client is the part of a speech-enabled application that runs on the user's mobile device or Mac (or, in principle, any voice-enabled and networked device). Its role is to capture and relay speech or text to the speech mashup manager (or *SMM*), which in turn handles authentication, accounting, and communication with the AT&T speech servers. The SMM returns the results from the speech related task (either ASR or TTS) to the client. For an ASR task, the result format is in simple XML or JSON formats, or using the proposed EMMA standard (<http://www.w3.org/TR/2007/WD-emma-20070409/>). For a TTS task, an audio stream is returned.

You can create your own client from scratch, modify a client to incorporate code provided in this chapter for relaying audio or text and receiving back the results, or download one of the following from the Speech Mashup portal web site through the Sample Code link:

- > A Java ME client that can be used for most Java-enabled mobile devices.
- > A native client for the iPhone.
- > A Safari plugin for the Macintosh. The plugin records audio and communicates with the SMM, it can be controlled through a JavaScript API, making it easy to create speech-enabled web pages.

The client communicates with the SMM using a REST (Representational State Transfer) API.

8.1 REST API information

The speech mashup manager provided by AT&T adheres to REST principles of statelessness, ensuring that each request made contains all relevant information, including the information returned from previous requests made earlier in the same session. (Thus multiple searches assume the same location, relayed at the beginning of the search.)

However, in order to support clients that cannot hold an HTTP request output stream open to send real-time audio, the speech mashup manager also supports a stateful mode of operation, where multiple chunks of audio are sent using multiple HTTP requests; in this mode, the server will maintain enough state to join the chunks of audio and present them to the WATSON ASR server as a continuous stream. The recognition results will be returned to the client once it sends the final request of the sequence.

The REST API used for making requests follows the *name=value* format. Table 6.1 lists the commands, which are URL-encoded, to directly control the recognizer actions. See Table 6.3 for commands to control the TTS conversion.

Table 8.1 Request API parameters for ASR

Parameter	Value	Description
uuid	string	Required. Unique user ID assigned at registration.
resultFormat	string	Optional. Result format, which can be EMMA, JSON, XML. Defaults to XML .
appname	string	Required. Name of application directory.
cmd	string	<p>Required. One of the following command strings:</p> <p>oneshot, rawoneshot Starts ASR in stateless mode; the request body will contain the entire audio stream.</p> <p>start Starts ASR in stateful mode; the audio stream will be sent using one or more <i>audio</i> or <i>rawaudio</i> requests.</p> <p>stop Stops stateful ASR and returns the result.</p> <p>audio, rawaudio Sends a chunk of audio for stateful ASR</p> <p>The “raw” versions of <i>oneshot</i> and <i>audio</i> include the audio unencoded; the non-“raw” versions include the audio hex-encoded, for the benefit of JavaScript clients that cannot manipulate raw bytes.</p>
audioFormat	String	<p>Optional. This specifies the format of the audio data supplied by the client. Possible values are:</p> <p>amr Adaptive Multi-Rate (AMR), narrow-band only</p> <p>au Sun AU, μ-law or 16-bit linear</p> <p>caf Apple Core Audio Format, μ-law or linear</p> <p>wav Microsoft/IBM Wave, μ-law or linear</p> <p>mulaw Raw μ-law</p> <p>linsbe Raw 16-bit linear, signed, big-endian</p> <p>linsle Raw 16-bit linear, signed, little-endian</p> <p>linube Raw 16-bit linear, unsigned, big-endian</p> <p>linule Raw 16-bit linear, unsigned, little-endian</p> <p>auto Any of the above; will look for amr, caf, wav, or au header, and if none is found, will use statistical analysis of the first 1024 bytes of audio to determine the format (raw μ-law or any of the above raw 16-bit linear varieties); falls back on header-less amr if the statistical analysis fails.</p> <p>If unspecified, this option defaults to auto.</p>
control	string	Optional. Parameters to control the operation of the speech recognizer. See next section for a description of recognizer controls.
grammar	string	Required. Name of grammar. In case of grammars with the same name, the recognizer searches first for personal grammars then for shared grammars and lastly for prebuilt grammars.
sampleRate	integer	Optional. The audio data sample rate. This only needs to be specified for audio data where the sample rate isn’t encoded in the data itself, i.e. raw μ -law or linear; for formats that do include the sample rate, this parameter is ignored. Defaults to 8000 Hz.
platform	string	Optional (but recommended). Some text to identify the make, model, and version of the client’s hardware platform,

Table 8.1 Request API parameters for ASR

		e.g. "BlackBerry 8800".
client	string	Optional. Some text that identifies the client software version, e.g. "Hotel Finder 2.0".
imei	string	Optional. If the client is a mobile device, clients may set this to the device's International Mobile Equipment Identity code, to allow application developers to distinguish individual users. For production applications, this parameter should not be used, or at least not without the user's explicit consent, because of privacy considerations.
field	string	Optional. For clients that use ASR in multiple contexts, e.g., first to recognize a location and next a business name, this parameter distinguishes the specific context. This is often redundant since the grammar will usually be specific to the context in question, but it can be helpful for ASR tuning.

8.1.1 Setting recognizer parameters

Within the client, you can control the following three recognizer settings, which are set using the `control` parameter described in table 6.2:

- > Speed-vs-accuracy, which represents the tradeoff between accuracy (how well the recognizer matched the actual utterance) to the amount of CPU (represented in time) required to achieve the accuracy. Increasing accuracy slows the speed of processing.
- > Sensitivity, which determines whether a sound is judged more or less likely to be speech rather than random noise. The higher the sensitivity, the quicker the recognizer is to judge noise to be speech.
- > Number of returned results. The default is for one result to be returned. However, in some cases, such as when the application cross-checks results against a database, you may want more than one result to increase the chances that the correct result is returned to the client.

Additional parameter controls will be added in future versions of the speech mashup.

You control recognizer parameters using the REST API's `control` parameter with the `set WATSON` command:

```
set name=value
```

where *name* can be one of the parameters in the following table and *value* is a value appropriate to the parameter. Since all speech mashup commands must be URL-encoded (see <http://en.wikipedia.org/wiki/Percent-encoding> for details), the following is an example of how to change a parameter setting:

```
&control=set+name=value
```

Table 8.2 Recognizer parameters

Parameter	Value range	Description
<code>config.speedVsAccuracy</code>	0.0-1.0	Controls the speed-vs-accuracy trade-off, with 0 being the fastest, and 1.0 the most accurate recognition. The default is .5 .
<code>config.vadSensitivity</code>	1-100	Controls how sensitive the recognizer is when determining that audio is speech. Use a lower value for noisy environments & a higher one for low-noise situations. The default is 50 .
<code>config.nbest</code>	<i>n</i>	Number of best results forwarded to the client. Default is 1 .

8.1.2 Request API parameters for TTS

The following parameters apply for the Natural Voices server. If there is any conflict between these parameters and SSML tags, the SSML tags have priority.

Table 8.3 Request API parameters for TTS

Parameter	Value	Description
<code>uuid</code>	string	Required. Unique user ID assigned at registration.
<code>text</code>	string	Optional. Text may also be supplied in the body of a POST request.
<code>audioFormat</code>	string	Optional. This specifies the format of the audio data supplied by the client. Possible values are: amr Adaptive multi-rate (AMR), narrow-band only mulaw AU with μ -law encoding alaw AU with A-law encoding linear AU, 16-bit linear The default is amr .
<code>voice</code>	string	Optional. Crystal (default) or Mike.
<code>sampleRate</code>	integer	Optional. The audio data sample rate. Defaults to 8000 Hz. Note that AMR-NB has a fixed sample rate of 8000 Hz, so specifying a different sampleRate will produce odd-sounding results.
<code>ssml</code>	True or False	Optional. Set this parameter to True when text contains SSML tags. (When set to the default, False, each word is pronounced, including SSML tags).

8.2 Sample clients for devices

The two clients and the Safari plugin are available for downloading from the Speech Mashup Portal (use the link *Sample code for clients* in the portal menu). The following sections give a high-level description of each.

8.2.1 Client for Java ME

Audio capture in the Java ME environment is performed using the Mobile Media API (MMAPI, specified in JSR-135).

Recording is performed using code similar to this:

```
import java.io.ByteArrayOutputStream;
import javax.microedition.media.Manager;
import javax.microedition.media.Player;
import javax.microedition.media.control.RecordControl;
// Start recording
Player player = Manager.createPlayer("capture://audio?encoding=amr");
player.realize();
RecordControl rc = (RecordControl) player.getControl("RecordControl");
ByteArrayOutputStream bos = new ByteArrayOutputStream();
rc.setRecordStream(bos);
rc.startRecord();
player.start();
// ...now recording; digitized audio is written to ByteArrayOutputStream bos
// Stop recording
RecordControl rc = (RecordControl) player.getControl("RecordControl");
player.stop();
rc.stopRecord();
rc.commit();
player.close();
byte[] audio = bos.toByteArray();
```

The first object created is the Player. This is the generic MMAPI name for a multimedia device; the precise type of device created is determined by the URL passed to `Manager.createPlayer()`. URLs starting with "capture:" indicate recording devices, such as audio or video recorders, or still image capture devices. The "encoding=amr" parameter is optional; on devices that support multiple audio formats, this can be used to select a specific one. If available, AMR (adaptive multi-rate compression; see also http://en.wikipedia.org/wiki/Adaptive_multi-rate_compression) with fixed mode AMR_12.20 (12.20 kbit/s) is the format of choice, because it compresses speech significantly better than other common formats, and since bandwidth on the cellular data channel can be very limited, high compression can be crucial to providing quick response times.

You control the Player by obtaining its RecordControl, which is where to attach a destination for the recorded data -- a ByteArrayOutputStream in the above example code. Another possibility is to use the `setRecordLocation()` method, which takes a URL argument and can be used to direct the data to a file on the local file system (via the use of a "file:" URL) or to another multimedia device.

Since data will not be manipulated or saved locally and since the volume of data will be small (a few kilobytes for a typical utterance in AMR format; a few dozen kilobytes in the case of uncompressed 16-bit PCM), it can be conveniently captured to memory. Saving to memory also avoids the security issues associated with multimedia- and filesystem-access APIs.

Audio is sent to the WATSON recognizer using the widely used `HttpConnection` and `HttpsConnection` interfaces.

The servlet is invoked with one or more request parameters, and optionally a request body containing audio. There is one required parameter, **cmd** (command), which can have the values **start**, **audio**, **rawaudio**, **stop**, **oneshot**, and **rawoneshot**. If the command is **start**, two additional parameters are supported: **control** is an optional chunk of text that is sent to WATSON recognizer verbatim to initialize the instance, and **grammar** is a required parameter that specifies the grammar to be used.

Once the **start** command is sent, and the servlet has opened and initialized a connection to WATSON server, the client may start sending **audio** or **rawaudio** commands. These are POST requests containing hex-encoded (**audio**) or raw (**rawaudio**) audio in the request body; the servlet will decode the audio, if necessary, and forward it to WATSON. Finally, once audio capture is finished, the client sends a **stop** command, which causes the servlet to send a "stop" message to the WATSON server and then wait until it receives a Notify Message with an Event Type of "phrase_result"; this message is then formatted as XML and returned to the client.

The **oneshot** and **rawoneshot** commands combine the entire **start/audio/stop** or **start/rawaudio/stop** sequence in one command; they require the same parameters as the **start** command, and they expect the audio data in a POST request body, like the **audio** and **rawaudio** commands.

WMMJavaMEClient demonstrates all these tasks. It sends a POST request with a **rawoneshot** command, and captures the response's body in a byte array:

```
import java.io.ByteArrayInputStream;
import org.xmlpull.v1.MXParser;
import org.xmlpull.v1.XmlPullParser;
// Send a POST to the WMMServlet, with the uuid, command,
// and grammar passed as query string parameters in
// the request URL, and the captured audio passed in the request body
byte[] audio = recStream.toByteArray();
InputStream is = new ByteArrayInputStream(audio);
String args = "?uuid=[your_own_UUID]" +
    "&appname=[application name]" +
    "&cmd=rawoneshot" +
    "&grammar=ypc-citystate-gram";
String url = "http://service.research.att.com/smm/watson" + args;
HttpConnection con = null;
OutputStream os = null;
byte[] data = null;
String encoding = null;
String text = null;
boolean success = false;
try {
    con = (HttpConnection) Connector.open(url);
    con.setRequestMethod("POST");
    con.setRequestProperty("Content-Type", "application/octet-stream");
    os = con.openOutputStream();
    byte[] buf = new byte[8192];
    int n;
    while ((n = is.read(buf)) != -1)
        os.write(buf, 0, n);
    is = null;
    os.close();
    os = null;
    int resCode = con.getResponseCode();
    if (resCode == 200) {
        is = con.openInputStream();
```

```

        encoding = con.getEncoding();
        if (encoding == null)
            encoding = "UTF-8";
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        while ((n = is.read(buf)) != -1)
            bos.write(buf, 0, n);
        data = bos.toByteArray();
        success = true;
    } else {
        text = "" + resCode + " " + con.getResponseMessage();
    }
} catch (IOException e) {
    text = e.getMessage();
} finally {
    if (os != null)
        try {
            os.close();
        } catch (IOException e) {}
    if (is != null)
        try {
            is.close();
        } catch (IOException e) {}
    if (con != null)
        try {
            con.close();
        } catch (IOException e) {}
}
}
}

```

If the request was successful, i.e., no exceptions were thrown and the HTTP response code was 200, it captures the response body in a byte array named **data**; it will contain something like this:

```

<NotifyMsg>
  <type>9</type>
  <rawText>event: phrase_result
reco: Florham Park New Jersey
interpretation: Florham Park New Jersey
norm_score: 53
grammar:
/n/u205/watsonadm/ypc/lm/./CityStateGram/xml.bmul2.0_bflr10.0_sil4.0/ac0.1_st0.1/USA/a.1
m
trigger: userTimeout
audio_start_sample: 5521
audio_stop_sample: 17040
audio_processed: 0
frame_count: 300
decoded_frame_count: 300
audio_seconds: 0
clock_seconds: 2.70673
cpu_seconds: 0.44
reco_start_time: 1196204960.538
uttr_end_time: 1196204963.245
uttr_end_audio_time: 3
prompt_audio_time: 0
slot: _city = 97 : 0
slot: _state = NJ : 0

```

```

</rawText>
<evType>phrase_result</evType>
<evData>
  <entry>
    <key>trigger</key>
    <value>userTimeout</value>
  </entry>
  <entry>
    <key>uttr_end_audio_time</key>
    <value>3</value>
  </entry>
  <entry>
    <key>slot</key>
    <value>{_city=97 : 0, _state=NJ : 0}</value>
  </entry>
  <entry>
    <key>cpu_seconds</key>
    <value>0.44</value>
  </entry>
  <entry>
    <key>norm_score</key>
    <value>53</value>
  </entry>
  <entry>
    <key>reco</key>
    <value>Florham Park New Jersey</value>
  </entry>
  <entry>
    <key>frame_count</key>
    <value>300</value>
  </entry>
  <entry>
    <key>audio_start_sample</key>
    <value>5521</value>
  </entry>
  <entry>
    <key>audio_stop_sample</key>
    <value>17040</value>
  </entry>
  <entry>
    <key>audio_seconds</key>
    <value>0</value>
  </entry>
  <entry>
    <key>interpretation</key>
    <value>Florham Park New Jersey</value>
  </entry>
  <entry>
    <key>audio_processed</key>
    <value>0</value>
  </entry>
  <entry>
    <key>grammar</key>
    <value>/n/u205/watsonadm/ypc/lm/./CityStateGram/xml.bmul2.0_bflr10.0_sil4.0/ac0.1_st0.1/
    USA/a.lm</value>
  </entry>
  <entry>
    <key>uttr_end_time</key>
    <value>1196204963.245</value>
  </entry>
  <entry>
    <key>event</key>
    <value>phrase_result</value>
  </entry>
  <entry>
    <key>decoded_frame_count</key>
    <value>300</value>

```



```

</entry>
<entry>
  <key>clock_seconds</key>
  <value>2.70673</value>
</entry>
<entry>
  <key>prompt_audio_time</key>
  <value>0</value>
</entry>
<entry>
  <key>reco_start_time</key>
  <value>1196204960.538</value>
</entry>
</evData>
</NotifyMsg>

```

The remaining code parses this XML, looking for a **key** element with content **reco**, and then extracting the content of the **value** element that follows. The code does this using [MXParser](#), which is a lightweight [XmlPullParser](#) implementation; its small memory footprint and minimal CPU usage make it a perfect fit for mobile devices, although for those used to the more common SAX or DOM parsers, it will take a bit of getting used to:

```

// Extract city/state information from XML
if (success) {
    XmlPullParser parser = new MXParser();
    parser.setInput(new InputStreamReader(new ByteArrayInputStream(data), encoding));
    int eventType;
    String currKey = null;
    String reco = null;
    String value = null;
    while ((eventType = parser.next()) != XmlPullParser.END_DOCUMENT) {
        if (eventType == XmlPullParser.START_TAG) {
            value = null;
        } else if (eventType == XmlPullParser.END_TAG) {
            String name = parser.getName();
            if (name.equals("key"))
                currKey = value;
            else if (name.equals("value") && currKey.equals("reco"))
                reco = value;
        } else if (eventType == XmlPullParser.TEXT) {
            String t = parser.getText();
            if (value == null)
                value = t;
            else
                value += t;
        }
    }
    text = "reco: " + reco;
}

```

The **text** variable now holds the recognized text.

The `SpeechMashupGuide.zip` file that accompanies this manual contains the full source code for `WMMJavaMEClient` (click the Sample Code link on the portal home page). The full code is more sophisticated than what was shown above; instead of storing the audio on the client and sending it only after recording has ended, it sends the audio in real time; this leads to a significant increase in complexity, including a custom implementation of `HttpConnection` for platforms without proper HTTP/1.1 support for “chunked” requests bodies. The complexity is worthwhile, though, since it can dramatically improve end-to-end response times.

8.2.2 Native client for the iPhone

Click the Sample Code link on the portal home page for a zipped file that contains source code for a native iPhone application, called SMMDemo, that demonstrates how to use the mashup portal from that platform.

To build and run SMMDemo, first unpack the SMMDemo.tgz package; to do this, copy it to a location of your choice, then open a terminal window, and execute these commands:

```
cd <location.of.your.choice>
tar xvfz SMMDemo.tgz
```

The iPhone SDK does not provide access to the iPhone's built-in AMR audio encoder. The formats that are supported all generate much larger audio streams, which can be a problem when those audio streams have to be transmitted over the cellular data channel (EDGE or 3G), where available data bandwidth can be severely limited. In order to work around this, SMMDemo performs AMR encoding in software, using version 6.10 of the 3GPP audio codec.

To download this code, open a web browser, and go to the following URL:

http://www.3gpp.org/ftp/Specs/archive/26_series/26.104/26104-610.zip

Save the 26104-610.zip file to <location.of.your.choice>/SMMDemo.

Next, execute these commands in the same terminal as before:

```
cd SMMDemo
./setup.sh
```

The setup.sh script will unpack the 26104.zip file, apply patches to some of them, and copy the files required to build SMMDemo to the SMMDemo/Classes directory.

Once setup.sh has finished, the contents of the SMMDemo directory should look like this:

README.txt	Brief reminder of how to get the contents of the SMMDemo.tgz package ready for use in Xcode
26104-610.zip	3GPP AMR codec download
26104-610.doc	3GPP AMR codec documentation
26104-610_ANSI_C_source_code.zip	3GPP AMR codec source code
encoder.c.patch	patches to 3GPP code
interf_enc.c.patch	
interf_enc.h.patch	
rom_enc.h.patch	
setup.sh	script to unpack and patch 3GPP code
MainWindow.xib	
SMMDemoViewController.xib	Interface Builder files that define the application's UI layout, and its connections with the actual code
SMMDemo_Prefix.pch	iPhone application boilerplate
main.m	

Info.plist	
c-code/	Directory containing unpatched 3GPP AMR codec source files, extracted from 26104-610_ANSI_C_source_code.zip
Classes/AMREncoderWrapper.h	Interface to 3GPP codec
Classes/AMREncoderWrapper.m	
Classes/PCMRecorder.h	Interface to iPhone audio capture
Classes/PCMRecorder.m	
Classes/SMMDemo.h	This ties together the audio recording, audio level monitoring, and audio streaming; its <code>receiveResponse</code> method receives the response from the mashup manager and handles it.
Classes/SMMDemo.m	
Classes/SMMDemoAppDelegate.h	Interface Builder boilerplate
Classes/SMMDemoAppDelegate.m	
Classes/SMMDemoViewController.h	Event handling code for <code>SMMDemoViewController.xib</code>
Classes/SMMDemoViewController.m	
Classes/StreamSocket.h	HTTP interface to mashup manager: uses a raw socket and explicitly created HTTP POST request to stream audio to the server and receive its response
Classes/StreamSocket.m	
Classes/encoder.c	3GPP AMR codec, patched
Classes/interf_dec.c	
Classes/interf_dec.h	
Classes/interf_enc.c	
Classes/interf_enc.h	
Classes/interf_rom.h	
Classes/rom_dec.h	
Classes/rom_enc.h	
Classes/sp_dec.c	
Classes/sp_dec.h	
Classes/sp_enc.h	
Classes/sp_enc32.c	
Classes/sp_enc32.h	
Classes/typedef.h	
SMMDemo.xcodeproj/	The Xcode project. Open this from Xcode to start building and running <code>SMMDemo</code> .

Once the source code is set up as shown above, you can open `SMMDemo.xcodeproj` in Xcode, build it, and run it on the iPhone simulator or on an actual iPhone.

Overview of the main SMMDemo source files:

PCMRecorder.m: This file uses the AudioToolbox API to record 16-bit linear PCM at 8000 Hz. The *init* method initializes an *AudioStreamBasicDescription* record with the appropriate parameters, *activate* activates the audio queue using *AudioQueueNewInput()* and allocates buffers using *AudioQueueAllocateBuffer()* and *AudioQueueEnqueueBuffer()*; and *start* and *stop* control recording using *AudioQueueStart()* and *AudioQueueStop()*, respectively.

AMREncoderWrapper.m: This file is a wrapper around the 3GPP PCM-to-AMR encoder. It is used when SMMDemo is in “amr” mode; in “pcm” mode, the output from PCMRecorder is sent to the speech mashup manager as-is.

StreamSocket.m: Implements sending an HTTP POST request with HTTP/1.1-compliant “chunked” request body encoding; this allows streaming audio, where the audio length is not known in advance. (This code uses *[NSStream getStreamsToHost]*, that is, a raw socket, instead of *NSMutableURLRequest*; the latter has the *setHTTPBodyStream* method that directly supports “chunked” request bodies, but unfortunately, the API for setting up the input stream consumed by the *NSMutableURLRequest* is actually more complex than the sample code provided with this document.)

The *initStreamSocket* and *openConnection* methods initialize and open the *StreamSocket* instance; *initPostMessage* and *sendHeaders* create and send the HTTP request header (NOTE: *initPostMessage* contains the speech mashup manager’s URL, hard-coded in the *startPost* variable); *setResponseObserver* registers the callback that will be invoked when the response from the speech mashup manager is received; the *sendAudioBytes* method should be called to receive the audio from the iPhone’s microphone (either directly, in case PCM mode is used, or after having been encoded by the AMR encoder, in case AMR mode is used; *closePost* finishes the current POST request, and *closeConnection* closes the current HTTP connection.

SMMDemo.m: The main class, which handles high-level events and responds to various callbacks. Of interest for testers is the *recordingType* variable in the *loadingComplete* method; this variable can be used to select AMR or PCM recording, by setting the variable to “amr” or “pcm”, respectively. AMR encoding is more compact, requiring 1525 bytes per second vs. 16000 bytes per second for PCM, but AMR encoding requires a lot of CPU time, and on fast networks, response time may be better with PCM. On cellular networks, where available data bandwidth can be very limited, AMR will typically give faster response times.

To make SMMDemo with the speech mashup manager of your choice, change the URL in *[StreamSocket initPostMessage]* according to your speech mashup manager’s base URL and parameters; see page 51 for details.

8.2.3 Safari plugin for Mac

The *SpeechMashupGuide.zip* file that accompanies this manual contains source code for a plug-in that can be used with Safari under Mac OS X. The plug-in can be controlled using JavaScript calls, and handles both recording audio and communicating with the server.

The code is in the *Safari-Plugin* folder, and includes a Project for use with Xcode, and a Makefile for use with the GNU toolchain.

When the plug-in is installed under /Library/Internet Plug-Ins or \${HOME}/Library/Internet Plug-Ins, it will register itself to handle the audio/watson MIME type. To use it in a web page, add this somewhere within the page body:

```
<embed name="audio" id="audio" type="audio/watson"
height="1" width="1">
```

Your JavaScript code can get access to the plugin by using `getElementById()`:

```
var plugin = document.getElementById("audio");
```

The plugin supports the following methods:

- > `startAudioRecording()` Starts capturing audio
- > `stopAudioRecording()` Stops capturing audio
- > `resetAudioRecording()` Reinitializes the audio recorder
- > `playRecordedAudio()` Plays back audio that has been recorded so far.
- > `asr(grammar, uuid)` Calls the portal server to do speech recognition on the audio that has been recorded so far; the result from the speech recognizer is returned as a String. The grammar and uuid parameter values are passed to the portal as the corresponding request parameters.
- > `asrWithParams(grammar, uuid, params)` Like `asr()`; the additional `params` parameter is appended to the request URL, and can be used to add an arbitrary set of additional parameters; it is useful for specifying platform, client, field, etc.
- > `asrAsync(grammar, uuid, callback)` Asynchronous speech recognition: this call is like `asr()`, except it doesn't wait for the response from the portal, but instead returns immediately. This method is useful for ensuring your web page stays responsive while waiting for the speech recognizer to finish. Once the result is available, a global function whose name is given by the `callback` parameter will be invoked, with a single String-valued parameter containing the speech recognizer's result.
- > `asrAsyncWithParams(grammar, uuid, callback, params)` This call is like `asrAsync()`; the additional `params` parameter is appended to the request URL, and can be used to add an arbitrary set of additional parameters; it is useful for specifying platform, client, field, etc.

- > `setBaseURL(url)` By default, the plugin tries to connect to the portal at `http://service.research.att.com/smm/watson`; using `setBaseURL()`, you can point it at a different server.
- > `getBaseURL()` Returns the base URL set with `setBaseURL()`.

The `plugin_test.html` page in Safari-Plugin demonstrates how to use `startAudioRecording()`, `stopAudioRecording()`, `asrWithParams()`, and `asrAsyncWithParams()`; it is also useful for testing the plug-in itself under control of the Xcode debugger.

8.3 Applets for Java-enabled browsers

8.3.1 Introduction

To use speech recognition (ASR) and text-to-speech (TTS) in web applications on Java-enabled browsers, you can use Java applets. We provide such applets, called `WatsonApplet` for ASR and `AudioPlayer` for TTS. The following sections show how to load these applets, how to control them using JavaScript, and specify the browser and JVM requirements for clients.

8.3.2 Speech Recognition (ASR)

The ASR applet, `WatsonApplet`, is provided in the `SignedApplets.jar` file, which you can download at <http://service.research.att.com/smm/SignedApplets.jar>.

In HTML 4, you can load this applet using the tag

```
<applet name="WatsonApplet"
        archive="SignedApplets.jar"
        code="com.att.speechmashups.applet.WatsonApplet.class"
        width="0"
        height="0"
        mayscript>
```

This tag, and an alternative way of loading the applet in strict XHTML and HTML 5 environments, are explained in detail in section 4.

Once you have embedded `WatsonApplet` in an HTML page, you can control it from JavaScript. The applet will appear in the DOM under the name you provided as the *name* attribute in the `<applet>` tag or `deployJava.writeAppletTag()` call, so, using the name from the example above, you can get the applet instance like

```
var asrApplet = document.getElementsByName("WatsonApplet")[0];
```

or simply

```
var asrApplet = document.WatsonApplet;
```

Before using the applet to do ASR, you must tell it which server to talk to, which user ID (UUID) and grammar to use, etc. You do this by creating a request URL, as described on page 51ff.

For example, using the mash-up server at <http://service.research.att.com/smm/watson>, with the UUID 0123456789ABCDEF0123456789ABCDEF, the “digits” grammar, and the “def001” application ID, you’d build the URL like

```
var url = "http://service.research.att.com/smm/watson"
+ "?uuid=0123456789ABCDEF0123456789ABCDEF"
+ "&grammar=digits"
+ "&appname=def001"
+ "&resultFormat=json";
```

This done, you pass the URL to the applet:

```
asrApplet.setUrl(url);
```

You have to do this only once, unless your page uses multiple grammars, in which case you may have to pass a new URL before each ASR request.

Note that the example above also specifies that the ASR result should be returned in JSON format; this is generally recommended since JSON is easy to work with in JavaScript. However, if you require other result formats such as EMMA, those are also available; see table 8.1 in chapter 8.

Having set the URL, you are now ready to perform ASR. To initiate this, call the applet’s `startRecording()` method before the user starts speaking, and `stopRecording()` after they are done:

```
<script>
    function watsonCallback(result, session) {
        var resultObj = eval('(' + result + ')');
        alert("ASR Result: " + resultObj.results[0].reco);
    }
</script>

<form>
    <input type="button" value="Press & Hold to Record"
        onmousedown="asrApplet.startRecording()"
        onmouseup="asrApplet.stopRecording()" />
</form>
```

The `startRecording()` and `stopRecording()` methods both return instantly. After the `startRecording()` call, the applet starts a background thread that reads audio from the system’s default audio input device and sends it to the server in real time, and this continues until `stopRecording()` is called, or until the ASR engine times out and returns a result on its own initiative¹.

Once the ASR engine returns a result, the applet delivers it to the JavaScript environment by calling a global function named *watsonCallback*. This function receives two parameters, the first one being the result itself (in whatever format was requested using the *resultFormat* parameter in the request URL), and the second one being the HTTP session ID.

In order to simplify the code, it is also possible to obtain the ASR result and HTTP session ID synchronously; this way the use of a callback can be avoided and the whole flow of control handled in one function, at the cost of blocking the JavaScript engine while the applet waits for the result.

¹ See section 6.3.1 for details on how to accomplish this.

To use this mode, call `startRecording()` with the parameter *false*:

```
asrApplet.startRecording(false);
```

Calling `startRecording(true)` is equivalent to calling `startRecording()` without a parameter.

If you call `startRecording(false)`, the applet will not try to call `watsonCallback()`; instead, the JavaScript code must call the applet's `getResult()` and `getSession()` functions to get the ASR result and HTTP session ID. Both of these functions will block, if necessary, until the required datum is available.

Note that the HTTP session ID returned by the applet's `getSession()` method or passed to the `watsonCallback()` function is a different entity than the WATSON session ID that is returned within the ASR result object. The former can be useful to establish a group of ASR requests, while the latter is used primarily to locate WATSON logs for debugging.

In order to end an HTTP session and force a new one to be created, call the applet's `invalidateSession()` method.

WatsonApplet API Summary:

```
// Methods
setUrl(url)                // set request URL; see sec. 8.1 for details
startRecording()           // start recording, asynchronous results
startRecording(async)      // start recording
stopRecording()            // stop recording
getResult()               // get ASR result (synchronous)
getSession()              // get HTTP session (synchronous)
invalidateSession()        // invalidate HTTP session for ASR

// Callbacks
// This should be defined at the global scope
// of the document containing the applet
watsonCallback(result, session) // asynchronous result delivery
```

8.3.3 Text-to-Speech (TTS)

The TTS applet, `AudioPlayer`, is provided in the `Applets.jar` file, which you can download at <http://service.research.att.com/smm/Applets.jar>.

Note that this applet is not specifically a text-to-speech applet: it can be used to play .wav and .au audio files from arbitrary URLs. To use it for TTS, you'll just ask it to play a URL that points at the Mash-up Server's TTS servlet. This is explained below.

(The applet can even play file: URLs, but you must load it from the `SignedApplets.jar` file in that case, since unsigned applets aren't allowed to access the local filesystem. The `SignedApplets.jar` file can be downloaded at <http://service.research.att.com/smm/SignedApplets.jar>. [TODO: Recognizing containers from file: URLs by their extensions or magic numbers, if the OS doesn't provide useful MIME types on its own])

In HTML 4, you can load this applet using the tag

```
<applet name="AudioPlayer"
        archive="Applets.jar"
        code="com.att.speechmashups.applet.AudioPlayer2.class"
        width="0"
```



```
height="0"
mayscript>
```

This tag, and an alternative way of loading the applet in strict XHTML and HTML 5 environments, are explained in detail in section 4.

Once you have embedded `AudioPlayer` in an HTML page, you can control it from JavaScript. The applet will appear in the DOM under the name you provided as the *name* attribute in the `<applet>` tag or `deployJava.writeAppletTag()` call, so, using the name from the example above, you can get the applet instance like

```
var ttsApplet = document.getElementsByName("AudioPlayer")[0];
```

or simply

```
var ttsApplet = document.AudioPlayer;
```

Next, you must construct the URL that represents the audio you want to play. The Mash-up Portal's TTS servlet is `http://service.research.att.com/smm/tts`, and to this URL you must add parameters to specify your identity (your UUID), the text to speak, the desired audio format, etc. Here is the full list:

uuid (required): your Universally Unique Identifier, obtained from the portal

appName (required): the application where the servlet's log messages get written to in the portal

voice (optional): one of `crystal`, `crystal16`, `mike`, `mike16`, `rosa`, `rosa16`, `alberto`, or `alberto16`; default: `crystal`. Crystal and Rosa are female voices, Mike and Alberto are male; Crystal and Mike are English while Rosa and Alberto are Spanish. The versions whose names end in 16 are 16-bit voices; the others are 8-bit.

text (optional): the text to speak. If unspecified, the applet will assume the request is a POST and try to read the text from the raw request body. Note that this is different than HTML form POST submission, which uses a URL-encoded request body, and can send multiple parameters in the body.

audioFormat (optional): `mulaw`, `alaw`, `linear`, or `amr`; default: `amr`,

sampleRate (optional): in Hz; default: 8000, must be 8000 for `amr`

volume (optional): 0..500, default: 100

notifications (optional): a set of zero or more, comma-separated, of: `bookmark`, `phoneme`, `viseme`, `word`, `simple`, `ogg`, and `multipart`; the first four select the types of notifications to be returned, while the last three specify the container format to be used for multiplexing the audio and notifications (see page 47ff for details). If unspecified, defaults to no notifications and no container.

Having constructed the URL, you can cause playback to begin by calling one of the applet's `play()` or `playWithPOST()` methods:

```
ttsApplet.play(url)
ttsApplet.play(url, closure)
ttsApplet.playWithPOST(url, text)
ttsApplet.playWithPOST(url, text, closure)
```

In all of these, the *url* parameter is the URL discussed above; the *text* parameter to `playWithPOST` is the text to be passed in the POST request body (note: do not also use a *text* parameter in the URL, because it will take precedence over the request body); and the optional *closure* parameter is an arbitrary character string that will be passed back to any notification callbacks the applet may invoke. (The notification callbacks are discussed later in this section.)

The applet has four additional methods to control its behavior: `stop()` cancels playback; `pause()` pauses playback but does not cancel it; `resume()` resumes a paused playback; and `setVolume()` controls playback volume [TODO: Not yet implemented].

When playback is started, the applet will invoke the global function *playbackBeganCallback* at the moment the audio actually begins to play, or it will call *playbackFailedCallback* if playback couldn't start for any reason (usually an incorrect URL). When playback finishes, whether because it reached the end or because it was cancelled using the `stop()` method, the global function *playbackEndedCallback* will be invoked.

All of these callbacks are passed one parameter, the *closure* that was passed to `play()` or `playWithPost()`. If no closure was passed in, the callback will receive a *null*.

Note that it is not required to actually define any of these callbacks if you're not interested in feedback about the state of playback.

The applet is also capable of returning notifications at the beginning of each phoneme, word, or viseme, and it is capable of returning bookmark notifications at specially marked positions in the text. These notifications are passed to the global *bookmarkCallback* function. This function receives two parameters: the first is the bookmark text, and the second is the closure. In the case of a bookmark notification, the bookmark text is whatever was in the `<mark>` tag's *name* attribute in the source text; for phoneme, viseme, and word notifications, the bookmark text consists of the word PHONEME, VISEME, or WORD, respectively, followed by the notification details, with all fields separated by colons. (See section 7.4.1 for details.)

[TODO:end-to-end bookmarks example]

AudioPlayer API Summary:

```
// Methods
play(url)                                // start playback using GET; closure = null
play(url, closure)                       // start playback using GET
playWithPOST(url, text)                  // start playback using POST; closure = null
playWithPOST(url, text, closure)         // start playback using POST
stop()                                  // stop playback
pause()                                 // pause playback
resume()                                // resume playback after pause
setVolume(volume)                       // set playback volume

// Callbacks
// These should be defined at the global scope
// of the document containing the applet
playbackFailedCallback(closure)          // called when playback fails
playbackBeganCallback(closure)           // called when playback begins
playbackEndedCallback(closure)           // called when playback ends
bookmarkCallback(text, closure)          // called for bookmarks and for phoneme,
                                          // viseme, and word notifications
```

8.3.4 Loading Applets in Detail

The ASR applet, WatsonApplet, and the TTS applet, AudioPlayer, are provided in two jar files, Applets.jar and SignedApplets.jar. Applets.jar contains AudioPlayer only, while SignedApplets.jar contains AudioPlayer and WatsonApplet, both with digital signatures applied.

The digital signatures enable the applets to access audio input devices and the local filesystem. WatsonApplet needs to be able to record audio and would therefore be unusable without a digital signature; AudioPlayer needs a digital signature only in order to be able to play audio from file URLs, but can be used without a digital signature for playing audio from HTTP or HTTPS URLs.

In HTML 4, you can load WatsonApplet applet using the tag

```
<applet name="WatsonApplet"
        archive="SignedApplets.jar"
        code="com.att.speechmashups.applet.WatsonApplet.class"
        width="0"
        height="0"
        mayscript>
```

For AudioPlayer, use the tag

```
<applet name="AudioPlayer"
        archive="Applets.jar"
        code="com.att.speechmashups.applet.AudioPlayer2.class"
        width="0"
        height="0"
        mayscript>
```

The *name* attribute defines the name under which the applet will be exposed in the browser's DOM for access by JavaScript. You may use any name you like.

The *archive* attribute is a URL that the applet's jar can be downloaded from. The URL can be **relative**, as in the example above, which means a location relative to the directory containing the HTML page, or, if the path starts with a slash character, it means a location relative to the web server's document root; or the URL can be **absolute**, where it includes a protocol and host, e.g.

<http://service.research.att.com/smm/Applets.jar>, allowing the jar file to be downloaded from different servers or hosts as well.

It is recommended that the jar file be hosted on the same server as the HTML page that loads it, and that any external JavaScript code that accesses the applet be on that same server as well. Using multiple servers for the various parts of the web application may cause the applet to fail because of the browser's security model.

When using absolute URLs, it is also recommended to load the applet using the same protocol (HTTP vs. HTTPS) as its containing page.

The *code* attribute is the name of the applet's main Java class file. It should be exactly as shown above.

The *width* and *height* attributes tell the browser the physical dimensions of the applet on the HTML page. Since WatsonApplet has no visual user interface, you should set these dimensions to 0. Making the applet invisible by placing it in a hidden <div> tag is not recommended, as some browsers may not load it in that case.

The *mayscript* attribute tells the browser that the applet may try to invoke functions in the JavaScript environment (callbacks). WatsonApplet needs this capability in order to be able to return ASR results asynchronously, and AudioPlayer needs it in order to invoke event notification callbacks (audio start, audio end, bookmarks).

In XHTML and HTML 5, the `<applet>` tag is no longer defined. While some browsers may continue to support it anyway, it is no longer safe to just assume that it will be available. The new way of supporting such browsers is the Java Deployment Toolkit (JDT). Using this toolkit requires two elements in the HTML page: first, loading the toolkit itself, which is a JavaScript program:

```
<script src="http://www.java.com/js/deployJava.js"></script>
```

Place this somewhere in the page's `<head>` element to ensure it is loaded before the page's main content.

Second, load the WatsonApplet, by using the JDT to generate the appropriate browser-specific `<object>` tag:

```
<script>
  var attributes =
    {   name:"WatsonApplet",
        archive:"SignedApplets.jar",
        code:"com.att.speechmashups.applet.WatsonApplet.class",
        width:0,
        height:0,
        mayscript:true
    };
  deployJava.writeAppletTag(attributes, null);
</script>
```

To load AudioPlayer, use this code instead:

```
<script>
  var attributes =
    {   name:"AudioPlayer",
        archive:"Applets.jar",
        code:"com.att.speechmashups.applet.AudioPlayer2.class",
        width:0,
        height:0,
        mayscript:true
    };
  deployJava.writeAppletTag(attributes, null);
</script>
```

The *name*, *archive*, *code*, *width*, *height*, and *mayscript* attributes correspond to those in the `<applet>` tag example shown at the beginning of this chapter.

Just as in the case of using an `<applet>` tag, it is recommended to load the applet from the same server as the containing HTML page and any JavaScript code that needs to access the applet. Using multiple servers may cause problems with the browser's security model. For the same reason, it is also recommended to use the same protocol (HTTP vs. HTTPS) for the HTML, applet, and JavaScript code. Loading the JDT from its master server at www.java.com should not cause problems, but you may still have to make sure the protocol matches the one that's used to load the containing page. For performance reasons, it may be preferable to host a copy of `deployJava.js` on your own web server, in case of congestion or outage at www.java.com (it happens!).

8.3.5 Client Requirements

On the client side (i.e. the user's), you will need a web browser that is capable of running Java applets, a Java Virtual Machine, and the Java Plug-in. It can be used under Microsoft Windows, Mac OS X, and Linux:

In Windows, you will need to install a JVM on the client's PC. If one isn't available already, or is too old (you need version 1.5 or later), download the latest version of the JRE from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, and install it. This will also install the Java Plug-in.

The applet has been tested successfully in Internet Explorer, Firefox, Opera, Chrome, and Safari with this setup.

In Mac OS X, Java and the Java Plug-in are part of the base system, and WatsonApplet and AudioPlayer work without any specific preparations on the user's machine. It has been tested successfully in Safari, Firefox, and Opera.

In Linux, you will need to install the Oracle (formerly Sun) JVM on your system if it isn't available already. If in doubt, run `java -version`; the result should look something like

```
java version "1.6.0_21"  
Java(TM) SE Runtime Environment (build 1.6.0_21-b06)  
Java HotSpot(TM) Server VM (build 17.0-b16, mixed mode)
```

If it doesn't, download the latest JRE from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, and install it.

To install the Java Plug-in, you need to create a link:

```
cd  
mkdir -p .mozilla/plugins  
cd .mozilla/plugins  
ln -s $JRE_HOME/lib/i386/libnpjp2.so .
```

In case the browser is already running, restart it to make sure it sees the plug-in.

The applet has been successfully tested with Firefox in this setup.

If the applet is loaded from SignedApplets.jar, you will probably get a security warning from the Java Plug-in when it is first loaded. It should look something like this:



The reason for this is that the applet is signed using a self-signed certificate. Just check the “Always trust content from this publisher” check box and click Run, and the applet will load. You will not get the warning again when you load the applet again later.

8.4 Configuring the client for the recognition result

The result returned by WATSON ASR contains the recognition result and detailed information about the parameter value and settings used during the recognition. However, you will need to write the client to identify the following information, depending on the application.

- > Recognition result or the semantic interpretation.

The semantic interpretation or the recognition string determines the next step for the web application.

For the semantic interpretation, look for the `interpretation` field; for the exact recognized string, look for either the `reco` field (XML or JSON) or `token` (EMMA).

- > Confidence score, which is a measure of how confident the recognizer is that the final result matches the original utterance. You can use the confidence score to establish a threshold for accepting or rejecting a result. In cases when the confidence score is low, you may not want the result used at all (and instead request that the user resubmit the request). For more information about setting the threshold, see the next page.

The following is sample EMMA output returned from the recognizer:

```
<?xml version="1.0" encoding="UTF-8"?>
<emma:emma version="1.0"
  xmlns:emma="http://www.w3.org/2003/04/emma"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2003/04/emma
    http://www.w3.org/TR/WD-emma-20070409/emma.xsd"
  xmlns="http://www.example.com/example">
  <emma:grammar id="gram1"
    ref="smm:grammar=en-us-date&UID=[your own UUID]"/>
  <emma:model id="modell"
    ref="smm:file=en-us-date.xsd&UID=[your own UUID]"/>
  <emma:info>
    <uuid>[your own UUID]</uuid>
    <watson>
      <version>watson-6.1.3655</version>
      <time>2008-08-31 13:01:32.338</time>
      <session_id>20080807-130131-00000652</session_id>
      <hostname>ss-2</hostname>
    </watson>
  </emma:info>
  <emma:one-of id="one-of1"
    emma:medium="acoustic"
    emma:mode="voice"
    emma:function="dialog"
    emma:verbal="true"
    emma:lang="en-US"
    emma:start="1218128492125"
    emma:end="1218128495725"
    emma:grammar-ref="gram1"

    emma:signal="smm:UID=[your_own_UUID]&file=/1/u205/speechmashu
ps/pino/def001/audio/20080807/audio-222262.amr"
    emma:signal-size="5766"
    emma:media-type="audio/amr; rate=8000"
    emma:source="smm:platform=null&device_id=null"
    emma:process="smm:type=asr&version=watson-6.1.3655"
    emma:duration="3600"
    emma:model-ref="modell"
    emma:dialog-turn="20080807-130131-00000652:1">
  <emma:interpretation id="nbest1"
    emma:confidence="0.5"
    emma:tokens="July thirty first 2 thousand 8">
    <![CDATA[<=$='???'+'$m+$d> <$m> <=$='07'> July <=$='07'> </$m> <$d>
<=$='31'> thirty first </$='31'> </$d> </$='???'+'$m+$d> <=$y+$m+$d>
<$y> <=$2+'00'> <$2> <=$='20'> 2 thousand </$='20'> </$2> </$=$2+'00'>
<=$2+$c7> <$c7> <=$='0'+$1> <$1> 8 </$1> </$='0'+$1> </$c7> </$=$2+$c7>
</$y> </$=$y+$m+$d> ]]>
  </emma:interpretation>
</emma:one-of>
</emma:emma>
```



Confidence score

Recognized utterance
(for best result)

8.4.1 Setting a threshold

For each recognized result, the recognizer assigns a confidence score based on how confident it is that the recognized hypothesis is correct. You may want to set a threshold in your application so that results below a certain threshold are not accepted.

If the client is to accept all recognized results, it is not necessary to set a threshold.

For each grammar or application, you'll need to determine the optimum threshold. It varies between grammars and will be known only through trial and error.

8.5 Combining speech processing with other processing

The SMM can connect to another server before or after the speech-related task so that the extra processing can be applied before or after the speech-related task. This can be useful, for example, when a recognition is used to look up a phone number or other information from a database, or when text normalization specific to an application should be applied before the text-to-speech conversion.

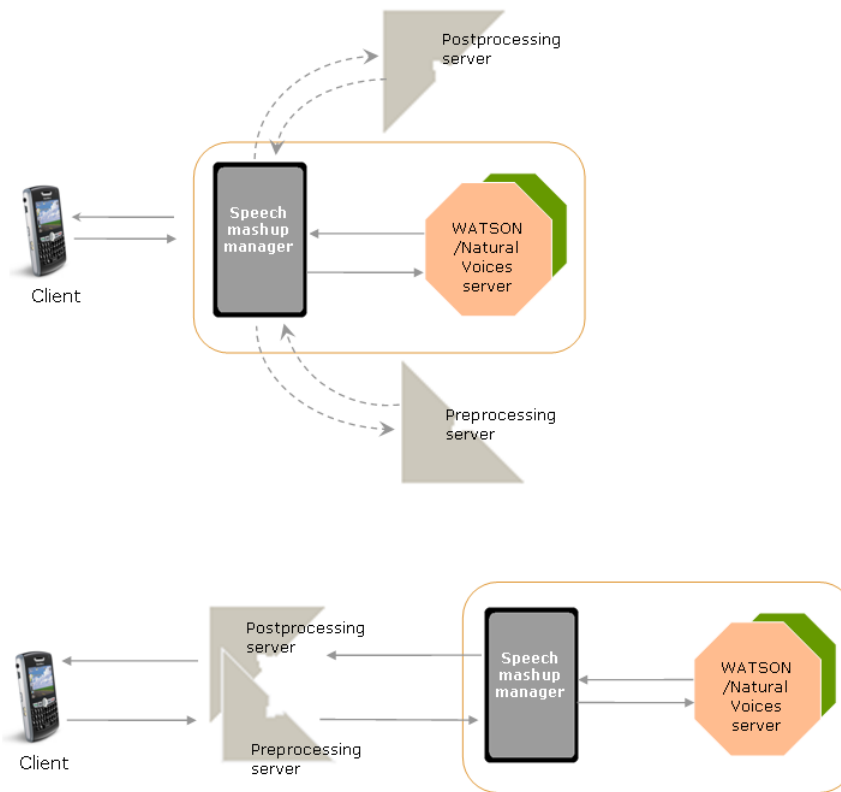


Figure 8.1 Pre- or post-processing performed by another server can be applied before or after the speech task. The connection can be opened by the SMM (top flow) or by the other server (bottom)

Since the SMM is already connected to the Internet, having it open connections to other servers makes it unnecessary for the client to open a second connection, an especially useful feature for mobile devices, where opening Internet connections tends to be slow.

Note: This section describes how the SMM connects to another server; however, you can also have the other server connect to the SMM to combine processing. The other server connects to the SMM using the SMM's URL, essentially becoming the "client" from the SMM's perspective, which you are given when registering at the Speech Mashup Portal (<http://service.research.att.com/smm/watson> followed by `?uuid=<uuid>&cmd=<command>&grammar=<grammar>`, for example, <http://service.research.att.com/smm/watson?uuid=>).

For the SMM to open a connection to another server or servers (in case you're doing both a pre- and a post-processing step), you need to enter the server's URL in the new application directory dialog (see page 25) either for pre-or post-processing, or both.

Instructions to the pre- or post-processing server are handled via X-headers relayed by the client. X-Param- response headers enable the preprocessing server to override parameters from the original request.

In addition, an X-WhatNext response header inserted by the preprocessing server determines the next step. There are three options:

X-WhatNext=0 – copy the response body received from the pre-processor to the client. This ends the transaction. X-WhatNext=0 is useful when an error occurs in preprocessing.

X-WhatNext=1 - perform speech recognition but no post-processing.

X-WhatNext=2 - perform speech recognition and then post-processing.

If no X-WhatNext header is supplied, X-WhatNext=2 is assumed.

8.5.1 Processing transaction steps

If there's a pre- or post-processing URL (or both), the sequence of steps is as follows:

1. The SMM opens a connection to the preprocessing server. Request parameters are passed to the preprocessor as request parameters; any X-Param-* response headers will be used to add or override parameters to be passed to the /smm/watson servlet proper and to the postprocessor. The SMM returns a response code of 200 when the call is successful; any other code denotes failure.

If the preprocessing call fails, no recognition occurs, and the SMM passes the response code and message back to the client. This ends the transaction.

2. If the preprocessing call is successful, (a) any X-Param- response headers override parameters from the original request, (b) the step sequence required by X-WhatNext is performed, and (c) any response header starting with X- is copied to the response to the client.
3. The SMM connects to the appropriate AT&T server and sends it the audio or text in the client's request body.

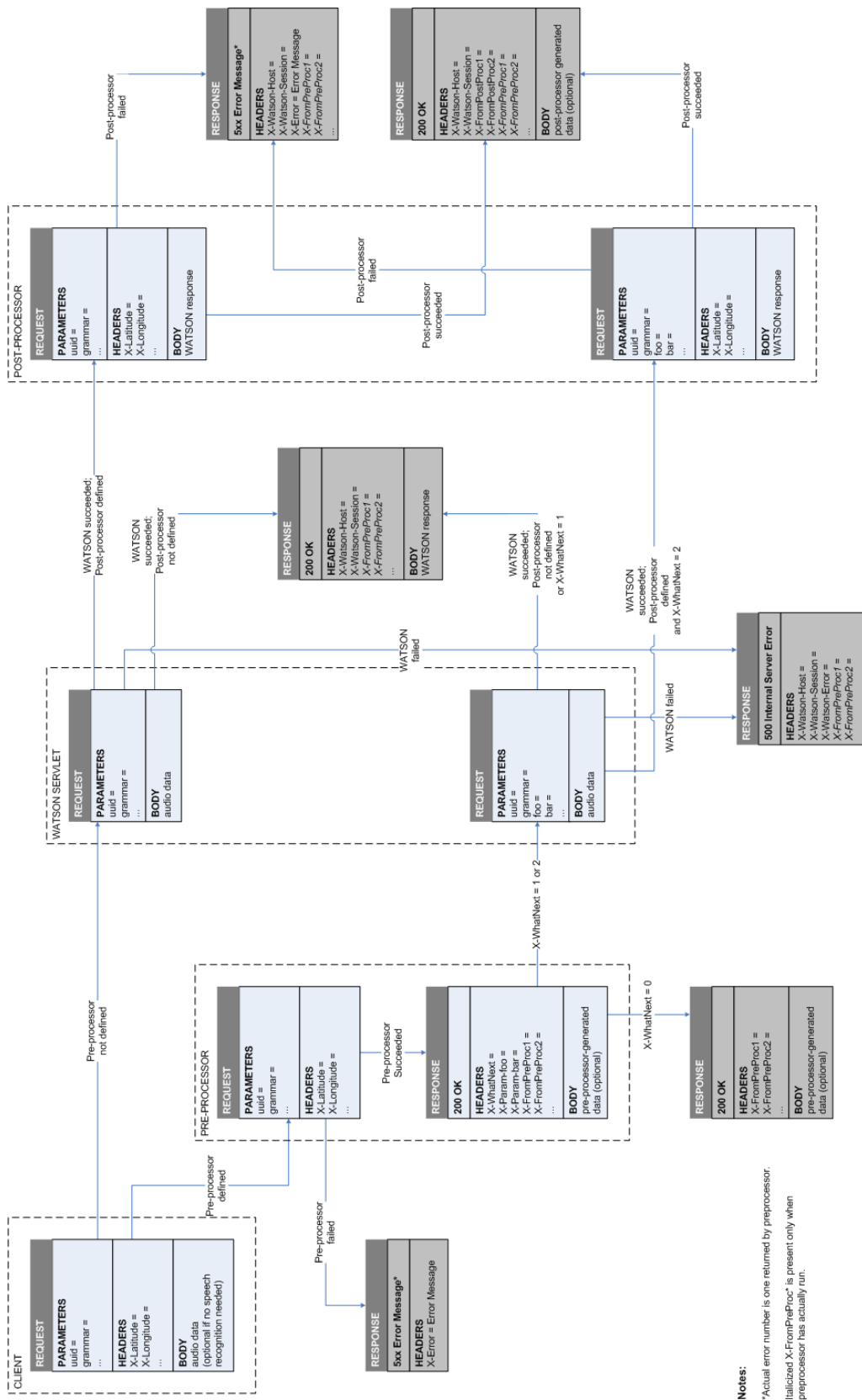
If the recognition fails, a response code of 500 is relayed to the client, and an exception message as the HTTP response message (also in the response body). This ends the transaction.

If the recognition or text conversion is successful (and there is no post-processing), the recognition result along with results from any preprocessing are returned to the client.

4. If there is a post-processing URL and the pre-processor (if there was one) returned X-WhatNext=2, the SMM connects to the post-processing server, passing it the same set of request parameters sent to the AT&T server, along with all request headers from the original request whose names start with X-.

If the post-processing fails (returns an HTTP response code other than 200), the SMM passes the response code and response message back to the client along with the speech task result.

If the post-processing succeeds, the SMM passes the X- response headers and the response body back to the client, along with the speech task result.



9 Administration & Troubleshooting

9.1 Viewing log files

Log files are maintained for each day's activities. To view a log file, log into the portal, select an application and then click **View Log Files**; select a date or, for the current day's log file, click **Log** at the top of the list.

Clicking the **Tail this Log** option allows you to view the logging entries in real time as they occur.

9.2 Updating passwords and other account information

To change the password used on the portal account or other information such as the name on the account and email: at the portal's Home page, select **Update account information** and enter the information you want to change.

9.3 Troubleshooting

This section gives solutions to the most common problems. To get help with problems not listed here, send email to watsonadm@research.att.com. Also include log files if they contain helpful information.

Problem: No recognition result was returned (the reco or token field was empty).

Solution: The recognizer was not able to match the audio to anything contained in the grammar.

- > The audio file may be noise only with no speech.
- > Words may have been truncated.

Problem: Results are not matching the actual utterances.

Solution: This problem can be due to any one of the following:

- > The threshold may be set too high.
- > Utterances may not be included in the grammar. Check whether the grammar should include utterances that it doesn't currently contain. (If you have transcriptions, verify that they can be matched by the grammar.)
- > Some voices are simply hard to recognize, particularly those that are heavily accented or that are mumbled.

Problem: Results are not being returned fast enough.

Solution: Speech mashups are in an early stage of development and efforts are continually being made to increase the speed of recognition. However, also check the following.

- > If you're allowing more than one result to be returned, reducing the number of results will increase speed a bit.
- > Adjust the recognition parameter `config.speedVsAccuracy` to be lower than 5. Since there is a speed-vs-accuracy tradeoff, increasing the speed will lower the accuracy. See page 37 to set from the portal or page 53 to set from the REST API.

10Glossary

acoustic model. The set of phoneme models (HMMs) that the decoder matches to voice features. WATSON ASR provides a general English acoustic model optimized for telephony.

automated speech recognition (ASR). A type of program that derives text from spoken language.

dictionary. In WATSON ASR, dictionaries specify how each word is pronounced in terms of the phonemes in the acoustic model. WATSON ASR supplies two dictionaries: a general dictionary with a large set of standard English words, and a TTS (text-to-speech) dictionary that generates spellings for words not included in other dictionaries. Custom dictionaries are also supported for words not found in the general dictionary.

grammar. Set of sentences that the recognizer is able to recognize. Sentences not included in the grammar cannot be recognized. There are two types of grammars: *rule-based* and *statistical*. Rule-based grammars (BNF) explicitly define a set of sentences to be recognized. In a statistical language model (SLM), the probable set of sentences is determined statistically based on a large set of training utterances. The application developer is responsible for building the grammar.

mashup. See **speech mashup** or **web mashup**.

phone set. Within the acoustic model, separately trained phones for special contexts: digits, names, quantities (whole numbers), confirmations, and alpha. Use of a phone set is specified either by using a digit between 1 and 10 (for the digits phone set) or by using a pronunciation tag.

pron (pronunciation) tag. An instruction contained within a rule-based grammar to specify use of special phone sets within the acoustic model. Pron tags, which can be applied to the entire grammar or to individual words, specify phone sets for digits, spelled alphanumeric characters, proper names, confirmations, and natural numbers (quantities).

rejection threshold. See *threshold*.

semantic tag. An expression inserted in a rule-based grammar so that specified utterances are replaced with a text string expected by the end application. For example, the utterance “I want to buy two round-trip tickets” might be replaced with the string “buy_ticket” and forwarded to the end application.

speech mashup. A web application that incorporates an AT&T speech technology, such as automatic speech recognition (ASR) or text-to-speech (TTS). An ASR speech mashup enables users of mobile devices to use voice commands when making requests from an HTTP application running on a web browser or mobile device such as an iPhone or BlackBerry; a TTS speech mashup takes text and returns speech. Speech mashups consist of a speech mashup client that relays audio or speech to the speech mashup manager, which then forwards.

speech mashup client. A Java client residing on a PC or mobile device that (1) relays audio to the speech mashup manager for forwarding to an AT&T speech server, and (2) accepts the result returned.

speech recognition. The technology that matches spoken language against the set of phrases and sentences defined in the grammar.

speech recognizer. Software driver that converts speech to text by matching an utterance to a recognized phrase or sentence, taking an acoustic signal and translating it to a digital signal.

text-to-speech (TTS). The speech service that converts text to speech.

threshold. Level of confidence the recognizer must have before it returns a result to the end application; any utterance with a score lower than the threshold is rejected. The threshold is currently set from the application.

WATSON ASR. AT&T's speech recognizer. It converts spoken language to text by matching speech sounds to words and sentences contained in the grammar.

WATSON server. A general-purpose engine that can perform a number of speech-related tasks, including ASR, text-to-speech, or TTS, dialog management, and natural language.

web mashup. A web application that combines data from more than one source into a single integrated tool, thereby creating a new and distinct web service that was not originally provided by either source. Google map mashups, for example, combine maps with other data, such as temperatures and crime statistics.

11Index

- abbreviations
 - handling in grammars, 10
- account. *See* speech mashup account
- accuracy (of recognition), 30
 - improving, 34
 - measuring, 30
- acoustic model, 2, 75
 - changing, 13
- application directories, 25
 - backing up, 26
 - creating, 26
 - selecting, 25
- ASR, 1, 2, 75
 - audio formats, 7
 - output formats, 7
 - supported languages, 7
- audio
 - sending with wget, 30
 - supported formats, 7
- audio input formats (ASR), 7
- define statements, 13
- dictionaries, 2
 - TTS, 37
- disfluencies, 34
- endpointing, 36
- grammars, 2, 3
 - assembling from other grammars, 9
 - backing up, 27
 - compiling, 27
 - contexts, 25
 - creating in WBNF, 12–21
 - creating in XML, 13–18
 - definition, 75
 - prebuilt, 29
 - rule-based, 3
 - sharing, 29
 - SLMs, 3, 23
 - testing, 31
 - unsharing, **29**
 - uploading, 27
 - uploading interactively, 28
- help, getting, iv
- include statements (WBNF), 9
- iPhone, client for, 47
 - downloading, 47
 - overview, 56–58
- Java Applets, **60**
 - ASR, 60
 - TTS, 62
- Java ME client, 47
 - downloading, 47
 - overview, 51–55
 - sample code, 51
- languages
 - supported for ASR, 7
 - supported for TTS, 7
- log files
 - WATSON ASR, 73
- Mac plugin. *See* Safari plugin
- Manage applications**, 25
- my grammars, 25
- Natural Voices, 4
- nbest, 49
- number of results
 - setting from REST API, 49
 - setting from the portal, 35
- numbers
 - expanding in TTS, 37
 - handling in grammars, 11
 - in a rule-based grammar, 9
- output formats
 - ASR, 7
 - TTS, 7
- passwords, changing, 73
- phone sets, 10, 75
- phoneme set, 40
- portal. *See* speech mashup portal
- POST
 - using to send text (TTS), 43
- postprocessing, 70
- prebuilt grammars, 25
- preprocessing, 70

pron tags. *See* pronunciation tags

pronunciation tags, 10, 34

- in a WBNF grammar, 20
- in an XML grammar, 15

prosody, 38

- SSML tags for, 39

recognizer, 3, 76

- setting parameters from portal, 35
- setting parameters in REST API, 49

rejection threshold. *See* threshold

Request API parameters, 48

REST API, 48

rule-based grammars, 3

- guidelines for creating, 9

Safari client (Mac)

- overview, 58–60

Safari plugin (Mac), 58

semantic tags, 12, 75

- in a WBNF grammar, 20
- in an XML grammar, 15

sensitivity, 50

- setting from the portal, 35
- setting from the REST API, 49

session timeout, 26

shared grammars, 25

silence penalty, 13, 34

SLMs, 3

- building, 23

speech mashup, 1, 75

- changes since previous release, iv
- problems with, 73
- summary of specifications, 7
- what you need to do, 7

speech mashup account

- changing passwords, 73
- updating information, 73

speech mashup clients, 2

- API parameters, 48, 50
- building, 47

speed vs accuracy

- setting from the portal, 35
- setting from the REST API, 49

speedvsaccuracy, 50

SSML tags, 4, 38

Tail this Log, 73

text

- converting to speech, 37
- normalizing (TTS), 37
- sending with wget, 43

text-to-speech. *See* TTS

threshold, 74

- setting, 69

transcriptions

- comparing to actual speech, 33
- creating, 33

troubleshooting, 73

TTS

- bookmarked stream formats, 43
- multipart/mixed, 46
- ogg, 45
- simple, 44
- default parameters, 42
- output formats, 7
- setting parameters, 42
- specifying a word pronunciation, 40
- supported languages, 7

UUID, 6, 13, 31

voice

- choosing in TTS, 50

WATSON ASR, 2, 3, 76

- architecture, 3

WATSON commands file, 35

WATSON servers, 1, 76

WBNF grammars, 3, 12–21

- pronunciation tags, 20
- sample grammar, **21**
- semantic tags, 20

wget

- using to send audio files, 30
- using to send text for TTS, 42
- using to upload grammars, 28

word penalty, 34

word weighting, 16

XML grammars, 3

- creating, 13–18
- pronunciation tags, **15**
- sample grammar, 16
- semantic tags, **15**
- word weighting, **16**