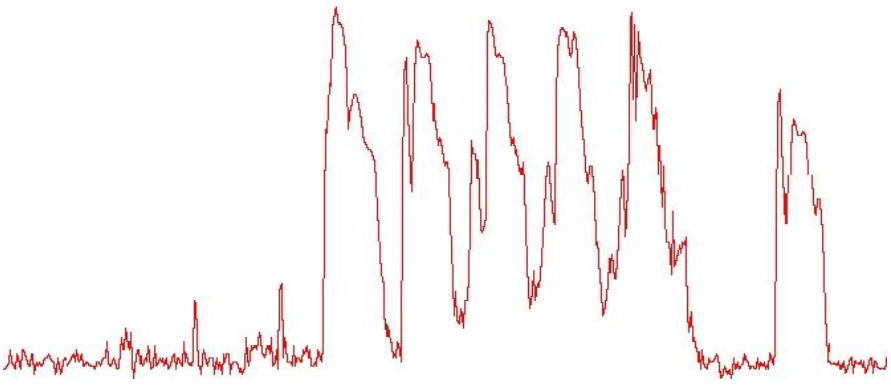
D R A F T

AT&T Speech Mashups

Application Developer's Guide



AT&T Speech Mashups, Application Developer's Guide, v. 0209
Printed in USA.
February 2009

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 1, "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 is manages account and connection information and is where you upload and manage grammars needed for applications requiring automatic speech recognition (ASR).

Chapter 2, "Creating a Rule-Based Grammar," describes the syntax and notations used to create the two rule-based grammars supported by WATSON ASR: XML and WNBF. It also describes how to parse the semantic interpretation so that a specific string is returned, based on the recognized result.

Chapter 3, "Creating 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 4, "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 5, "Modifying Output Speech for TTS," describes how to modify the output speech using SSML tags or to change word pronunciations using phonemic transcriptions.

Chapter 6, "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.

Appendix A, "Administration and Troubleshooting," describes how to view log files, change portal account information, and address problems that may occur.

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**.

Changes in this release

This version of the developer's guide (v.0209) documents the following new features and other changes that were incorporated since v.1208.

- 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.

Contents

Overview

1

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.

Speech mashup architecture

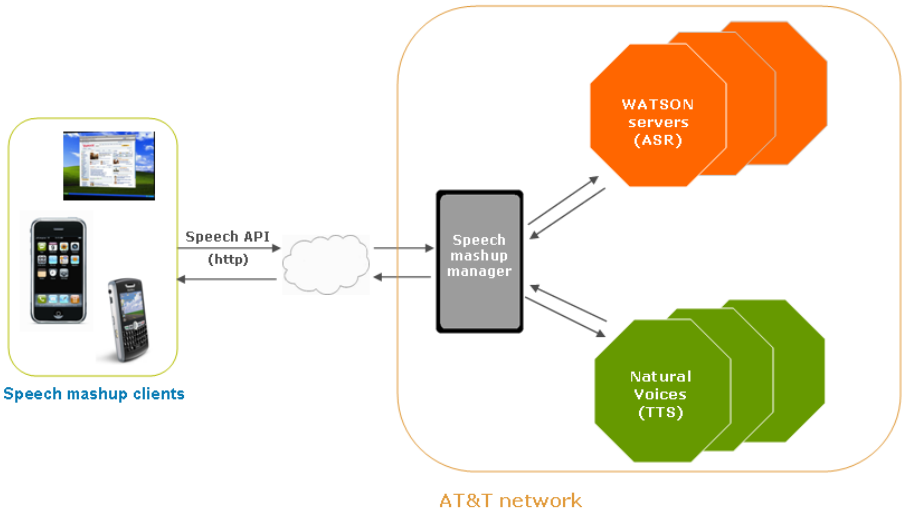


FIGURE 1.1 *In a speech mashup, clients access AT&T speech services through the speech mashup manager*

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:

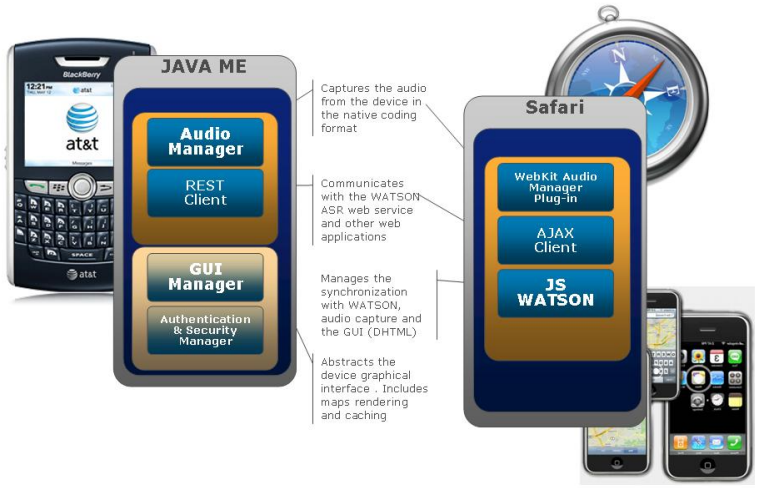


FIGURE 1.2 *Speech mashup clients (available for Java ME, the iPhone, and the Safari Mac browser) interface between the web application and WATSON ASR.*

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.

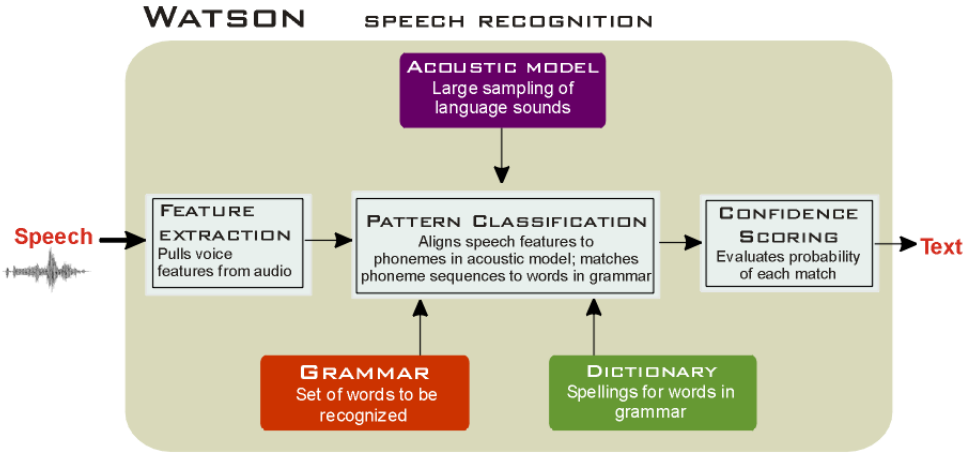


FIGURE 1.3 *WATSON ASR converts speech to text based on words, phrases, and sentences in the grammar.*

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.

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.

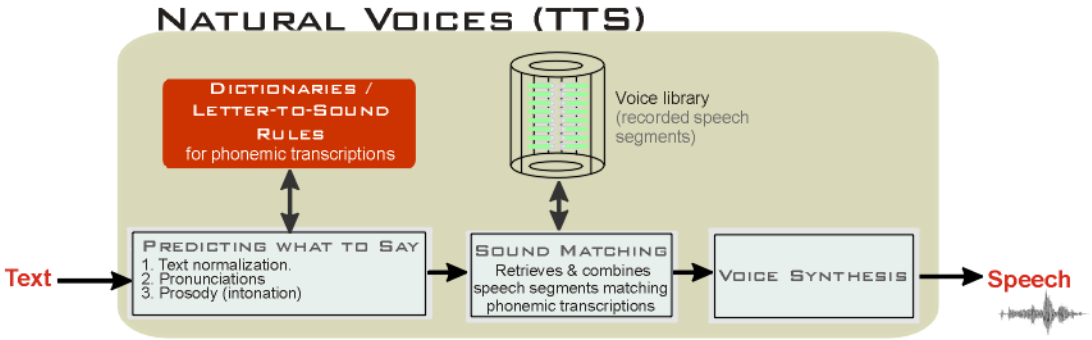
- A statistical language model (SLM) does not use explicit rules, but instead uses the statistical properties of thousands of transcribed utterances to help infer the language.

  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.

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.



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.

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.

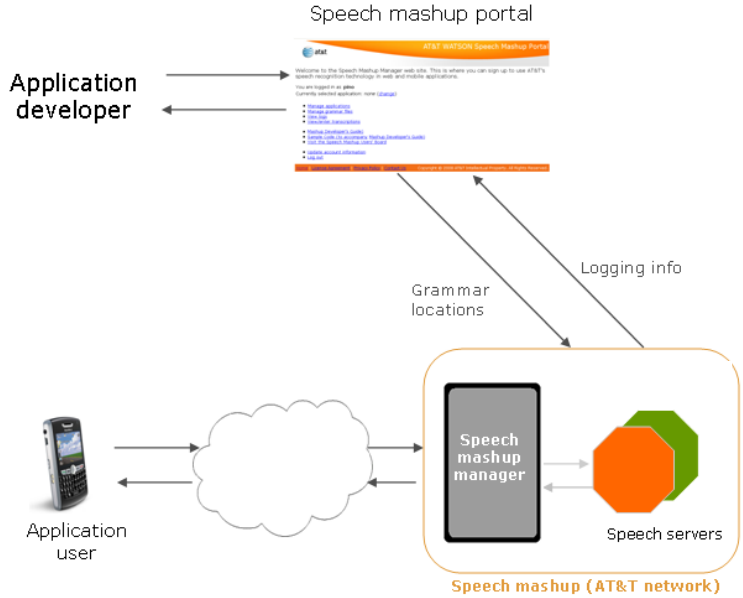


Figure 1.4 *Files and information related to speech mashups are accessible through the speech mashup portal*

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 1.5 *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.

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.

What you need to know

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

Table 1.1 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 42.)
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.

Creating a Rule-Based Grammar

2

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 27. 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.

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.

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.

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>
```

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 13) or "Adding semantic tags to a WBNF grammar" (page 18) as appropriate.

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 2.1 Define statement options	
name	**value**
pron	Applies specified pron tag to entire grammar. Can be **alpha**, **digits**, **name**, **quant**, **confirm**. See page 8 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>`

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 14):

```
<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 2.2 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 *n* times.
repeat="*n-m*"	Repeat the expression between *n* and *m* times.
repeat="*n-*"	Repeat the expression *n* 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.

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>
```

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*.

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>
```

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.

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
```

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 2.3 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)

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 }_ ;
```

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 19 shows the use of semantic tags.

The portal contains a set of builtin WNBF grammars that can be used, after being tested and modified for the intended purpose, as standalone grammars or incorporated with other grammars. See page 27 for a listing.

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?").

Setting up a Statistical Language Model

3

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 23.

Uploading, Managing, and Testing Grammars

4

Grammars belong to three contexts:

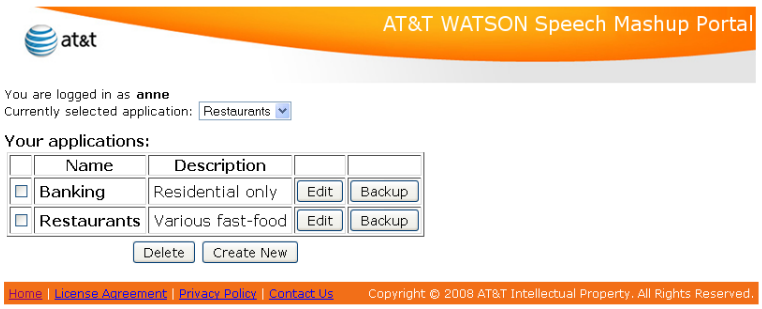
- **My grammars**. These are your personal grammars that you create and control. They are not accessible to anyone else.
- **Shared grammars**. Shared grammars are grammars created by others and then made available to all speech mashup users. You can view (and copy) these files but cannot rename or delete 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`.

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:

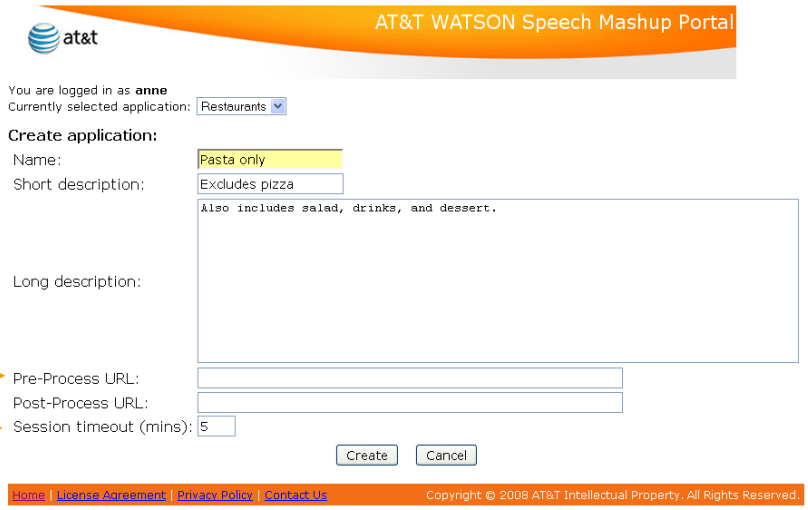


From the Manage Applications screen, you edit, back up, or delete current application directories, or create new ones.

Use Backup option 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 55), 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.

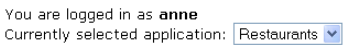
Warning: Deleting an application directory deletes all grammars, log files, and audio stored in it.

Uploading grammars

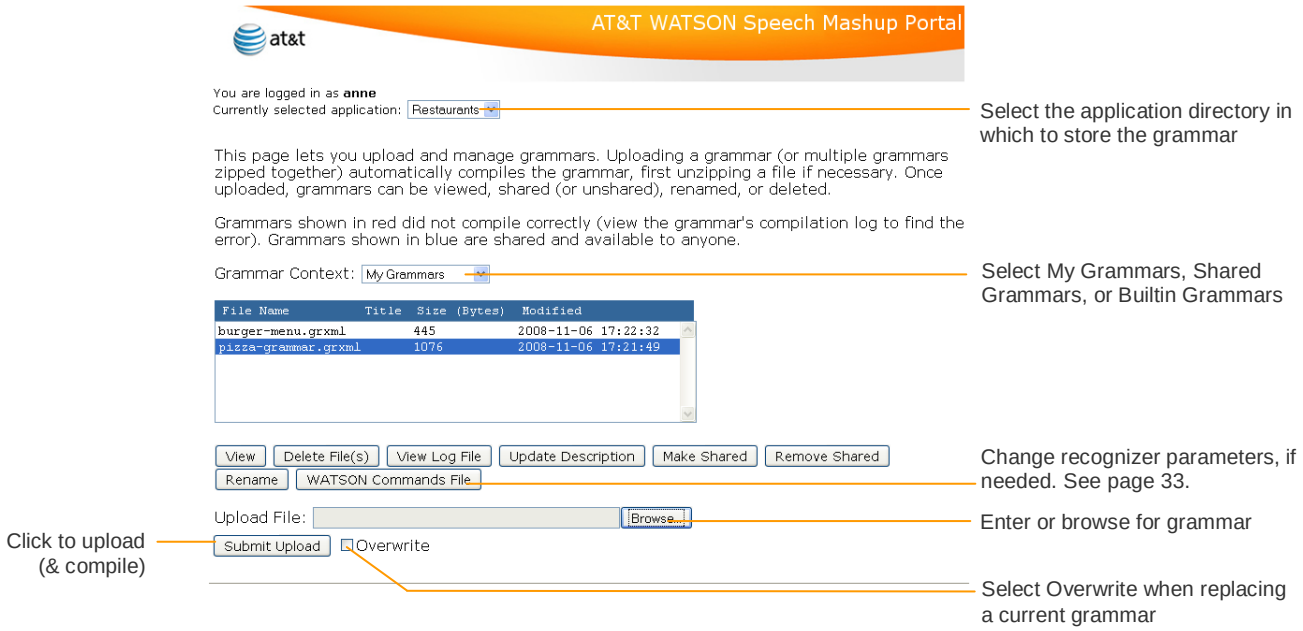
You upload grammars in two ways: either from the portal or by using `wget`. In both cases, uploading the grammar automatically compiles it.

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

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



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.



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**.

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.

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.

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, than shared grammars, and then prebuilt ones), it's probably good practice to uniquely name each grammar.

Table 4.1 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 *hhmmx*, where *hh* is the hour, *mm* is the minutes, and *x* 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 *mmm.nn* where *mmm* is the number of dollars (3 digits or less). *nn* 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 *mmm.nn*, *mmm* is the number of dollars and may be 3 digits or fewer. *nn* 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, yyyymmdd with ? for missing fields Example: july 4th: ????0704, november: the first: january 1st 1970: 19700101
es-us-helpexit.wbnf	Returns words **help**, **cancel**, and **exit**.

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.

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&amp;UUID=[your_own_UUID]"/>
<emma:model id="model1"
  ref="smm:file=en-us-date.xsd&amp;UUID=[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:UUID=[you_own_UUID]&amp;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&amp;device_id=null"
  emma:process="smm:type=asr&amp;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`.

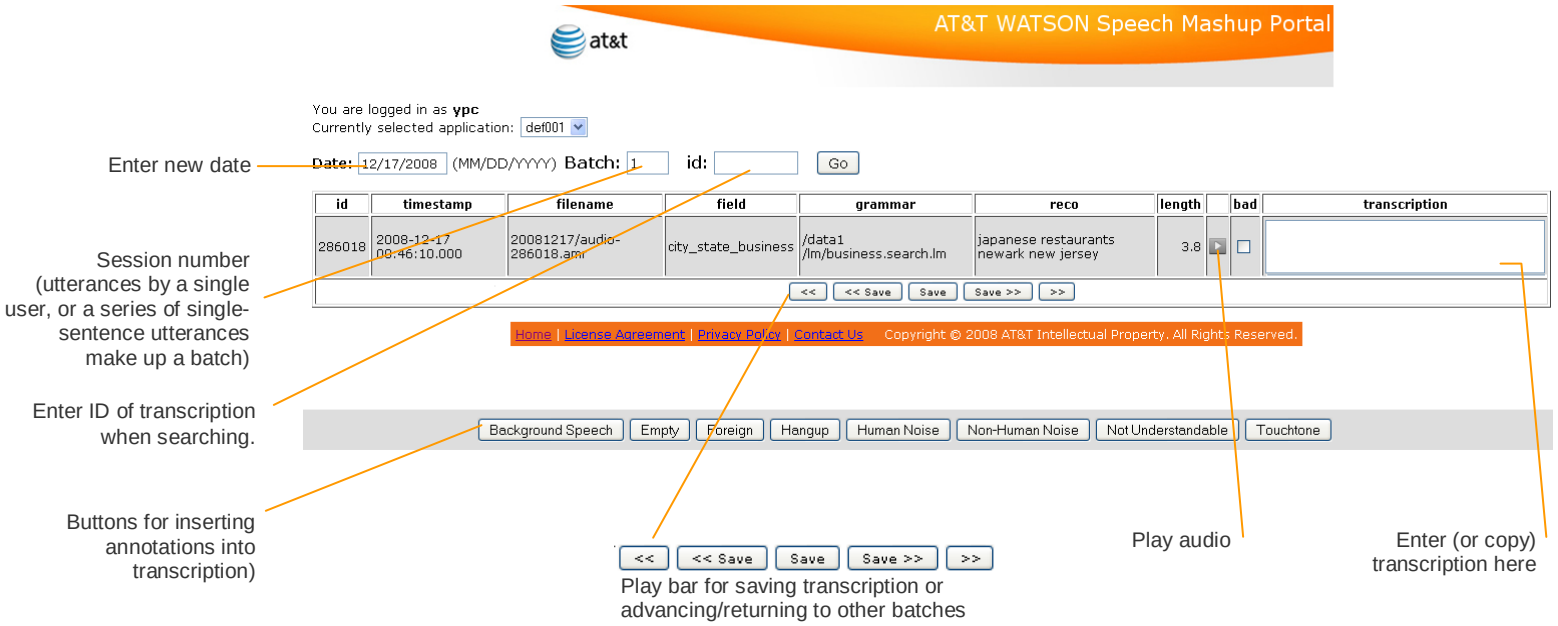
The reason is that that specific content type is handled specially by web servers; they assume that the request body contains data from an HTML form being submitted, and attempt to parse it accordingly, and the servlet or CGI program that processes the request never gets to see the raw request body. 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.

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.



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.

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 10. 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 13) or "Adding pronunciation tags to a WBNF grammar" (page 18) 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 affect.

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.

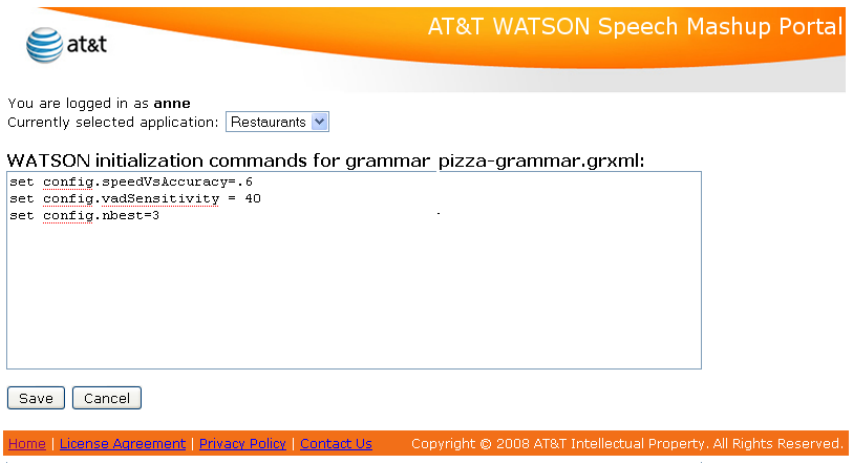
You set recognizer parameters by creating a WATSON commands file from the **Manage Grammars** dialog (select WATSON commands file from the dialog).

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.

`config.speedVsAccuracy`	Value range: 0.0 – 1.0 (default is .5)
`config.vadSensitivity`	Value range: 1-100 (default is 50)
`config.nbest`	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 44). Note that the REST `control` parameters override the settings contained within the WATSON commands file.

Modifying Output Speech for Natural Voices

5

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.

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 "leed"?
- 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).

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 5.1 SSML tags	
<say-as> tags *value* can be:	<say as> tags provide hints on which text expansions are more likely given the context. Syntax is: `<say-as interpret-as="value"> text </say-as>`
`"acronym"`	Pronounces each letter individually. Useful for acronyms embedded in all uppercase text. Same as `spell` tag.
`"address"`	Identifies text as an address. Pauses are inserted between address components, and building numbers are spoken in pairs, e.g., *1250 Elm* is read *twelve fifty elm*.
`"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`". `<say-as interpret-as="date" format="dmy"> 1/2/2008 </say-as>`
`"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 `<s>` to force sentence breaks.
`"literal"`	Passes the string through literally (as is), exactly as typed and without expansion (*Main St.* would be pronounced *Main* s t.).
`"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). `<say-as interpret-as="number" format="fraction | digits"> text </say-as>`
`"spell"`	Reads characters & digits separately (*Hello*, becomes H E L L O). Same as `acronym`.
`"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: `<tag-name> text </tag-name >`
`<speak>`	Identifies the language to use. `<speak xml:lang="en us">`, 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. `<voice name="mike"> text </voice> <voice name="cyrstal"> text </voice>` Can also identify a language: `<voice xml:lang="es_us"> 1 2 3 </voice>`
`<paragraph>`	(or `<p>`). 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: `<p xml:lang="en_us">`
`<sentence>`	(or `<s>`). Sets off text that is to be read in sentence structure. Can also change language: `<s xml:lang="es_us">`
Prosody tags	Prosody tags (except for `<break/>`) consist of an opening and closing tag.
`<break/>`	Inserts a pause of the specified length. Use with the **strength** or **time** attribute. `<break strength="level"/>` where *level* can be **none**, **x-weak**, **weak**, **medium**, **strong**, or **x-strong**

Table 5.1 SSML tags	
	<break time="*n*s | *n*ms"/> where *n* is the number of seconds or milliseconds For slight pauses with no intonation change, use the silence phoneme (<phoneme alphabet="darpa"″ ph="pau"/>).
<prosody volume=*level*>	Adjusts loudness of the audio output (<prosody volume="*level* | *n* | *n*%"> text</prosody>): <prosody volume=*level*> where *level* can be **silent**, **x-soft**, **soft**, **medium**, **loud**, **x-loud**, **default** (return to normal) <prosody volume=*n*> or *n* can be a number between 1 and 100 (or above) for a relative change from current volume. <prosody volume=*n*%> A minus or plus makes the change in volume a variation from the default, not current, volume
<prosody rate=value>	Varies the speaking rate to be faster or slower (<prosody rate="*value*"> *text*</prosody>): where *value* can be **x-fast**, **fast**, **medium**, **slow**, **x-slow**, or **default** (to return to a normal rate). *value* 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.
<emphasis level="*level*">	Applies more or less emphasis (<emphasis level="*level*"> *text* </emphasis>) where *level* can be **strong**, **moderate**, **none**, or **reduced**.
Phoneme tag	Specifies a particular pronunciation for a single instance of a word. No part-of-speech rules are applied. <phoneme alphabet="darpa" ph="f aa 1 dh er 0"/>. For a listing of the phonemes, see table 5.2.

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 5.2 DARPAbet phoneme set

Phoneme	Example	Transcription	More Examples
aa	Bob	/b aa b 1/	knot
ae	bat	/b ae t 1/	bad, gnat
ah	but	/b ah t 1/	cub, tuck, bud
ao	bought	/b ao t 1/	saw, caught
aw	down	/d aw n 1/	about, how
ax	about	/ax 0 b aw t 1/	onion, upon
ay	bite	/b ay t 1/	high, psycho
b	bet	/b eh t 1/	
ch	church	/ch er ch 1/	
d	dig	/d ih g 1/	
dh	that	/dh ae t 1/	
eh	bet	/b eh t 1/	mess, led
er	bird	/b er d 1/	curb
ey	bait	/b ey t 1/	laid, eight
f	fog	/f ao g 1/	phone
g	got	/g aa t 1/	
hh	hot	/hh aa t 1/	
ih	bit	/b ih t 1/	hip, in
iy	beat	/b iy t 1/	heap, teal
jh	jump	/jh ah m p 1/	
k	cat	/k ae t 1/	kick
l	lot	/l aa t 1/	
m	mom	/m aa m 1/	
n	nod	/n aa d 1/	noun
ng	sing	/s ih ng 1/	
ow	boat	/b ow t 1/	sew, coal
oy	boy	/b oy 1/	foil
p	pot	/p aa t 1/	
r	rat	/r ae t 1/	
s	sit	/s ih t 1/	
sh	shut	/sh ah t 1/	
t	top	/t aa p 1/	
th	thick	/th ih k 1/	
uh	book	/b uh k 1/	full
uw	boot	/b uw t 1/	fool, crude, do
v	vat	/v ae t 1/	
w	won	/w ah n 1/	
y	you	/y uw 1/	
z	zoo	/z uw 1/	xylophone
zh	measure	/m eh zh 1 er 0/	

Testing the TTS conversion

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

`text`	Text to speak. Note: Text may also be supplied as the body of a POST request (see below).
`audioFormat`	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**.
`sampleRate`	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 `sampleRate` will produce odd-sounding results.
`voice`	Voice to use. Currently there are two options: **Mike** (male, English) and **Crystal** (female, English). The default is **Crystal**.
`ssml`	Specifies whether or not the text to speak contains SSML mark-up; this may be **true** or **false**. The default is **false**.

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".

Building a Speech Mashup Client

6

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.

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 6.1 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	**Required.** One of the following command strings: oneshot, rawoneshot Starts ASR in stateless mode; the request body will contain the entire audio stream. start — Starts ASR in stateful mode; the audio stream will be sent using one or more *audio* or *rawaudio* requests. stop — Stops stateful ASR and returns the result. audio, rawaudio Sends a chunk of audio for stateful ASR The "raw" versions of *oneshot* and *audio* include the audio unencoded; the non-"raw" versions include the audio hex-encoded, for the benefit of JavaScript clients that cannot manipulate raw bytes.
audioFormat	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 au — Sun AU, µ-law or 16-bit linear caf — Apple Core Audio Format, µ-law or linear wav — Microsoft/IBM Wave, µ-law or linear mulaw — Raw µ-law linsbe — Raw 16-bit linear, signed, big-endian linsle — Raw 16-bit linear, signed, little-endian linube — Raw 16-bit linear, unsigned, big-endian linule — Raw 16-bit linear, unsigned, little-endian 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. If unspecified, this option defaults to **auto**.
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 6.1 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.

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 6.2 Recognizer parameters		
Parameter	**Value range**	**Description**
`config.speedVsAccuracy`	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**.
`config.vadSensitivity`	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**.
`config.nbest`	*n*	Number of best results forwarded to the client. Default is **1**.

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 6.3 Request API parameters for TTS		
Parameter	**Value**	**Description**
uuid	string	**Required.** Unique user ID assigned at registration.
text	string	**Optional.** Text may also be supplied in the body of a POST request.
audioFormat	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.
voice	string	**Optional.** Crystal (default) or Mike.
sampleRate	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.
ssml	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).

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.

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.mxp1.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.l
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 MXP1, 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.

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 receiveResponse 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 SMMDemoViewController.xib
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 SMMDemo.

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 41 for details.

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.

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&amp;UUID=[your own UUID]"/>
<emma:model id="model1"
  ref="smm:file=en-us-date.xsd&amp;UUID=[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:UUID=[your_own_UUID]&amp;file=/l/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&amp;device_id=null"
  emma:process="smm:type=asr&amp;version=watson-6.1.3655"
  emma:duration="3600"
  emma:model-ref="model1"
  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)

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.

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.

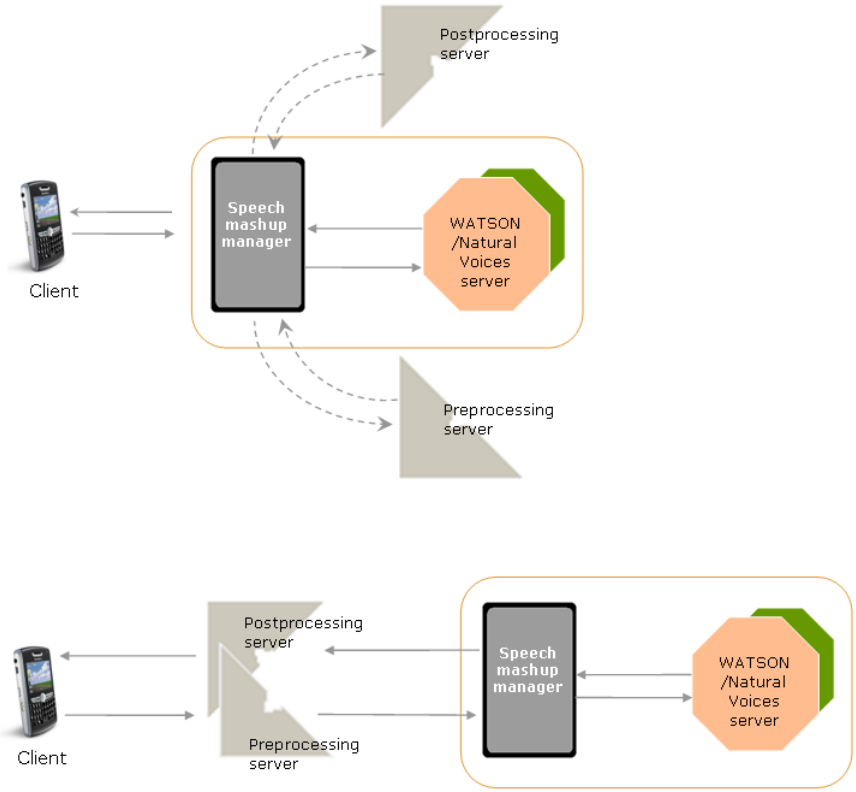


FIGURE 6.1 *Pre- or post-processing performed by another server can be applied before or after the speech task. The connection can be opened by the SMM (top flow) or by the other server (bottom)*

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 23) 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.

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.


CLIENT
REQUEST
PARAMETERS
uuid =
grammar =
...
HEADERS
X-Latitude =
X-Longitude =
...
BODY
audio data
(optional if no speech recognition needed)
Pre-processor not defined
Pre-processor defined
Pre-processor failed
RESPONSE
5xx Error Message*
HEADERS
X-Error = Error Message
PRE-PROCESSOR
REQUEST
PARAMETERS
uuid =
grammar =
...
HEADERS
X-Latitude =
X-Longitude =
...
Pre-processor Succeeded
RESPONSE
200 OK
HEADERS
X-WhatNext =
X-Param-foo =
X-Param-bar =
X-FromPreProc1 =
X-FromPreProc2 =
...
BODY
pre-processor-generated data (optional)
X-WhatNext = 1 or 2
X-WhatNext = 0
RESPONSE
200 OK
HEADERS
X-FromPreProc1 =
X-FromPreProc2 =
...
BODY
pre-processor-generated data (optional)
WATSON SERVLET
REQUEST
PARAMETERS
uuid =
grammar =
...
BODY
audio data
WATSON succeeded; Post-processor defined
WATSON succeeded; Post-processor not defined
WATSON failed
RESPONSE
200 OK
HEADERS
X-Watson-Host =
X-Watson-Session =
X-FromPrePreProc1 =
X-FromPrePreProc2 =
...
BODY
WATSON response
REQUEST
PARAMETERS
uuid =
grammar =
foo =
bar =
...
BODY
audio data
WATSON succeeded; Post-processor not defined or X-WhatNext = 1
WATSON succeeded; Post-processor defined and X-WhatNext = 2
WATSON failed
RESPONSE
500 Internal Server Error
HEADERS
X-Watson-Host =
X-Watson-Session =
X-Watson-Error =
X-FromPrePreProc1 =
X-FromPrePreProc2 =
...
POST-PROCESSOR
REQUEST
PARAMETERS
uuid =
grammar =
...
HEADERS
X-Latitude =
X-Longitude =
...
BODY
WATSON response
Post-processor failed
Post-processor succeeded
RESPONSE
5xx Error Message*
HEADERS
X-Watson-Host =
X-Watson-Session =
X-Error = Error Message
X-FromPreProc1 =
X-FromPreProc2 =
...
RESPONSE
200 OK
HEADERS
X-Watson-Host =
X-Watson-Session =
X-FromPostProc1 =
X-FromPostProc2 =
X-FromPrePreProc1 =
X-FromPrePreProc2 =
...
BODY
post-processor generated data (optional)
REQUEST
PARAMETERS
uuid =
grammar =
foo =
bar =
...
HEADERS
X-Latitude =
X-Longitude =
...
BODY
WATSON response
Post-processor failed
Post-processor succeeded
Notes:
*Actual error number is one returned by preprocessor.
Italicized X-FromPreProc* is present only when preprocessor has actually run.

Administration & Troubleshooting

A

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.

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.

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 33 to set from the portal or page 43 to set from the REST API.

Glossary

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 mobiles 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.

Index