Dialogue Framework

*RSVP Flavored AIML Specifications*

*Comprehensive documentation of how to write and extend AIML files.*

Version: v1.0

Last revised: 2014-12-11

Table of Contents

[1. AIML Basics 3](#_Toc415058983)

[1.1 What is AIML 3](#_Toc415058984)

[1.2 Elements of AIML 3](#_Toc415058985)

[1.3 Patterns 3](#_Toc415058986)

[1.4 Templates 4](#_Toc415058987)

[2. Supported Tags 5](#_Toc415058988)

[2.1 <bot> 5](#_Toc415058989)

[2.2 <condition> 5](#_Toc415058990)

[2.3 <date> 6](#_Toc415058991)

[2.4 <get> 6](#_Toc415058992)

[2.5 <set> 6](#_Toc415058993)

1. AIML Basics
   1. What is AIML

AIML, or Artificial Intelligence Markup Language, is and XML dialect for creating natural language software agents. In another word, it let you define the logic for a chatterbot to generate responses based on certain user inputs. AIML is used to write the world famous chatterbot ‘A.L.I.C.E.’ and also adopted in many of today’s popular chatterbots.

* 1. Elements of AIML

AIML contains several elements, which are XML nodes. The unit in which the logic of generating response to a certain input is contained inside a <category>. And inside each <category> there are two required nodes <pattern> and <template>, as well as one optional nodes <that>. They are described in further detail below.

<category>

<pattern>Hello</pattern>

<that>Hi</that>

<template>How are you</template>

</category>

When this category is loaded, the bot will respond ‘How are you’ when the received user input is ‘Hello’ and the bot’s last response was ‘Hi’.

* 1. Patterns

A pattern is a string of characters intended to match one or more user inputs. A literal pattern can be something like ‘Hello’ in the above case that only the input ‘Hello’ will be matched by this category. Standard AIML also supports wildcard matching. For example, one can use ‘\*’ to match anything. However, remember that AIML is targeting English so words are expected to be separated by spaces, which doesn’t fit Chinese well.

As a result, we extended standard AIML patterns and allows the usage of grammar patterns. To use grammar patterns, one need to solely use a <grammar> tag inside the <pattern> tag and place the grammar term that he/she is trying to match inside the tag. For example:

<category>

<pattern>

<grammar>basic.greeting</grammar>

</pattern>

<that>Hi</that>

<template>How are you</template>

</category>

Instead of matching the literal version of user input, now this category tries to match a public grammar term ‘basic.greeting’ which was already defined in the chatterbot’s grammar files, where ‘basic’ is the namespace and ‘greeting’ is the term. For instance, the grammar file could look like this:

Namespace basic

hi := strings{Hello}

public greeting : hi

Thus, if the user input was ‘Hello’, the grammar parser will parse the input beforehand and output the term ‘basic.greeting’, therefore it will be captured by the category listed above.

* 1. Templates

A template specifies the response to a matched pattern. A template may be as simple as some literal text, like

<template>My name is Lisa.</template>

A template may use variables, such as

<template>My name is <bot name=”name”/></template>

This template will substitute the bot’s name into the sentence, or

<template>You told me you are <gval term=”ask.name” key=”name /></template>

This will substitute the user’s name into the sentence. Note that <gval> is a custom tag which is not in the standard AIML. It renders value defined and captured in your grammar file when a public term is matched to access certain part of a user input.

Most of a chatterbot’s logic is performed inside <template> tags, one can use, combine, and nest tags to do tasks and generate complex response, while updating user associated information such as session and profile. In the following section we are going to demonstrate all supported tags for writing templates.

1. Supported Tags

In dialogue framework, every AIML and our custom tag is evaluated by a handler extends abstract class ‘TagHandler’ inside the ‘core.bot.ab.handlers’ package. If you want to add a new custom tag, feel free to create your own handler and link it to the ‘TagHandlerCollection’. Here we list and explain all the built-in tags and their handlers.

* 1. <bot>

The <bot> tag retrieves predefined bot properties. Bot properties are key-value pairs stored inside each bot’s configuration file ‘bot\_folder/config/properties.txt’, for example, a bot’s name can be defined in properties.txt as

Then we can refer to it in template like

<template>Hello my name is <bot name=”name” /></template>

name: Megabot

And it will print ‘Hello my name is Megabot’, where the <bot> tag is replaced by the predefined name property.

* 1. <condition>

The condition tag will render content within when the specified condition is met, otherwise it will simply be ignored. A template may contain multiple <condition> tags, and tags can be nested inside a <condition> tag. For example

<template>

<condition name=”gender” value=”male”>Hello, Sir.</condition>

<condition name=”gender” value=”female”>Hello, Madam.</condition>

</template>

The name property inside a condition (e.g. ‘gender’), is evaluated towards the current user session’s predicates table. In this case, if the name predicate was previously set to ‘female’, this template will render ‘Hello, Madam’ as the output. How to set a session predicate will be covered in the section of ‘<set>’ tag. The <condition> tag also has a list format, where multiple clauses can be placed inside one <condition> tag, for instance

<template>

<condition>

<li name=”gender” value=”male”>Hello, Sir.</li>

<li name=”gender” value=”female”>Hello, Madam.</li>

</condition>

</template>

This gives the identical behaviour. Additionally, one can place a default item which will be rendered if no condition can be met.

<template>

<condition>

<li name=”gender” value=”male”>Hello, Sir.</li>

<li name=”gender” value=”female”>Hello, Madam.</li>

<li>Hmm.</li>

</condition>

</template>

In this example, if ‘gender’ hasn’t been set (or anything but ‘male’ or ‘female’), the robot will reply back ‘Hmm.’.

* 1. <date>

The <date> tag will be render the server machine’s current date-time based on user provided java formatting string. If not present, the format is set to ‘EEE MMM dd HH:mm:ss zzz yyyy’ by default. An example is shown below

<template>

Current time is <date jformat=”HH:mm:ss MMM dd, yyyy” />.

</template>

This will return something like ‘Current time is 13:43:53 Mar 25, 2015’.

* 1. <get>

The <get> tag will retrieve two types of information for current user. If paired with ‘name’, it will return the matching value from current user session. On the other hand, when paired with ‘var’, it returns the temporary in memory matching value which shares the life cycle of only the current request. For example

<template>

OK, so you went to <get name=”place” />?

</template>

This will look up for the value matches key ‘place’ in current user sessions’ predicates table. If we do <get var=”place” /> instead, it will look for ‘place’ which is set as a variable within the current template, as variables for is parsing state are re-initialized every time a template evaluation starts.

* 1. <set>

As the counterpart of the <get> tag, the <set> tag let the dialogue developer to store a value either into the current user session or temporary in-memory template life cycle.

<template>

<set name=”place”>Paris</set>

</template>

If the above example was executed before the one for <get> tag, the <get> tag example will be evaluated to ‘OK, so you went to Paris’. Similarly, if we do <set var=”place”>Paris</set> then the value of place will be stored into memory instead and only available to <get> tag within the same <template> but written under.

* 1. <gval>

The tag <gval> stands for grammar value, it is used to retrieve captured information by grammar matching. It is handy when we are doing wildcard matching with grammars and can let dialogue developers access information provided by user that is caught by wildcards. For example if we have the following grammar file and AIML file

<category>

<pattern>

<grammar>ask.username</grammar>

</pattern>

<template>

Hello <gval term=”ask.username” key=”name” />!

</template>

</category>

namespace ask

wc := wildcard(1, 100)

sayname : “My name is “ wc

public username : sayname { name = wc.key; }