PANDORABOTS

Getting started with chatbot development

If you’ve found this tutorial, you hopefully know what a chatbot is and what Pandorabots can do to help you build and deploy one.

For those of you who don’t know, a chatbot is an application that can have a conversation in natural language. You can say something to the bot, and the bot will parse your input and provide the most relevent response it can. This type of application is known as the stimulus response model.

What is Pandorabots?

Pandorabots offers a cloud-based chatbot compiler and runtime. Our platform is the most popular implementation of the AIML 2.0 language, which is used to actually create the bot. Once you’ve deployed your code to our servers, your chatbot will be accessible via our Talk API. This allows you to integrate your bot into any application that can make an HTTP request.

This tutorial will focus on how to actually develop the bot. We will be talking primarily about AIML, and how to organize your code into organized, reusable modules for maximum efficiency.

What is AIML?

AIML is an acronym for “artificial intelligence markup language,” and is the primary language used currently by the Pandorabots platform. It is an extension of XML.

Hello world!

Let’s begin with everyone’s favorite lesson when learning a new programming language. Create a file called `main.aiml` and paste in the following text:

```
<?xml version="1.0" encoding="utf-8" ?>
<aiml version="2.0">

<category>
  <pattern>HI</pattern>
  <template>Hello world!</template>
</category>

</aiml>
```

Save this file to your bot, and now try talking to it:

> ***Human****: Hi*
> ***Bot****: Hello world!*

Let’s break down the file, line by line, to better understand the structure of an AIML file:

```
<?xml version="1.0" encoding="utf-8" ?>
```

This line declares your file as an XML document. While this is unecessary from the point of view of the Pandorabots platform, it actually enables features in your text editor that can be of great help during AIML development. You don’t have to take advantage of this, but we generally suggest that you include the declaration.

```
<aiml version="2.0">
```

This line declares the file as an AIML document. This line is required by the AIML compiler on our platform. All of your AIML code will appear between this and the final line, which marks the end of the AIML document.

```
<category>
```

This line marks the beginning of our first category, which is the base unit of code in an AIML based chatbot. Each category defines an input (pattern) and an output (template).

```
<pattern>HI</pattern>
```

This is the category’s pattern element. The pattern defines some input text. When you say something to the bot, it will evaluate all of its categories until it finds one whose pattern matches the input. In this example, our category will be matched when the user’s input is “HI”.

```
<template>Hello world!</template>
```

The template defines an action that the bot should take when a category has been matched. In this example, the action to return the text “Hello world!” to the person speaking with the bot.

```
</category>
```

This marks the end of our category. You can insert new categories below this, as long as they appear before the closing AIML tag.

```
</aiml>
```

This marks the end of the AIML document.

Moving forward

Your next stop should be Bot Building 101 where you can master both AIML and Platform basics.

Building Bots on the Pandorabots Platform

A tutorial for writing bots using the AIML 2.0 scripting language and the Pandorabots User Interface

Core Concepts

Bot Development consists of a relationship between the following three entities:

- ***chatbot(n.)*** or ***bot(n.)*** - a computer program designed to simulate conversation with a human end-user via voice or textual inputs
- ***botmaster(n.)*** - a person who creates, develops, and maintains the code and content that make up a chatbot
- ***client(n.)*** - a person chatting with a bot

The Pandorabots Platform is a feature-rich user interface designed to help you, the botmaster, create AIML-based chatbots.

- ***AIML(n.)*** - Artificial Intelligence Markup Language: the simple open standard language in which Pandorabots are written

Experienced AIML writers may choose to use their favorite text editor instead of this UI, but many find the features in this UI indispensable for bot creation, deployment, and maintenance.

Platform Components

Log In

The Pandorabots platform supports OAuth (single sign-on) via Google, Facebook, Twitter, Yahoo, and Github. You may also create an account using an email address and password. Upon initial log in, you will be assigned a username (taking the form of "unXXXXXXXX") and prompted to create your first bot. Bot names must be 3-64 characters in length and lowercase alphanumeric. Together, your username and botname form a unique identifier for your bot.

Bot Creation

To create a new bot, use the "+" button next to **My Bots** on the navigation panel. Note that AIML can be written in almost any natural language, and you can select the language for your bot from a drop-down upon creation. (For Asian and other languages that require segmentation, Pandorabots offers a machine learning-based segmenter as a service; email us to learn more).

Content Libraries (open source AIML files) may be available depending on the language selected. For example, adding the "Small Talk" Library (also previously known as "Rosie") to your bot will enable basic chitchat right out of the gate. Adding Content Libraries can save you the hassle of reinventing the wheel by writing AIML categories to handle common things people say in conversation. This base bot files can be updated as needed for your own custom bot responses. However, you can always select "Blank Bot" if you prefer to build from scratch.

If you have written a library and would like to open source it and see it featured for other developers, contact us!

Bot Statistics

Clicking on **My Bots** on the navigation panel will display some key global statistics for your production bots for the past 30 days. These include total *Interactions*, *Clients*, *Sessions*, and the average number of *Interactions/Session*. In the broader ecosystem, "Interactions/Session" is also sometimes known as *Turns per Conversation _("_TPC*"), and is a stat botmasters measure and reference to highlight client engagement with their chatbot.

Clicking on the name of each individual bot will display the same individual statistics for that specific bot.

Platform users who require more in-depth analytics can plug in additional third-party analytics engines of their choosing.

Bot Editing

Selecting the name of an individual bot will open a drop-down menu consisting of several components, including: *Edit*, *Deploy*, *Logs*, etc. The **Edit** is your gateway to creating and maintaining your bot. There are several options such as AIML code editor, Intents Tree, and Chat Design to make updates to your bot.

Using the Code Editor, from the Files option you can accomplish the following:

- Create new files, including AIML files (the primary type of bot files), Sets, Maps, Substitutions, and Pdefaults
- Save Files
- Download a Zip File of your bot (which we recommend doing frequently as a backup!)
- Upload bot files (if, for example, you have been working from your local editor or have AIML files from another platform)

Access and open your files from the menu on the left. Selecting a file will open it in the text editor, which is an ACE editor optimized for XML editing (since AIML is an XML-based language). Formatting AIML with the incorrect syntax (for example, forgetting a character in an opening/closing tag) will cause an error alert to appear in the form of a red X next to problematic line of code, which will include an additional description of the error type on hover.

Beneath the text editor, you can find a status bar full of helpful information such as whether and when the file was last saved/modified, and the file load order.

Staging versus Production

A common best practice in computer programming is to have two versions of software when one of those versions is live for the public: a *Production* version that is live, and a *Staging* version that is in development where you can test (and back out of!) any changes. This way, if something goes wrong in the staging version of the software you are editing, your changes do not impact the production version and disrupt end-users. One great feature of bots is that you can instantly update them after making changes, but it is important to first verify that none of your changes are breaking changes by testing thoroughly.

So, if you have a bot deployed on the platform that is live for the public, any changes you make to your bot using the Edit features will only impact the staging or *Sandbox* version of your bot. Once you have tested your bot, clicking *Publish* will push your changes live to production.

Bot Training via the Chat Widget

In the lower right hand corner of the interface, you may have noticed a circular chat icon. Clicking on this icon will allow you to chat with your bot as if you were the client from almost everywhere in the interface. This **Chat Widget** has several useful features that are vital for debugging and improving your bot.

- *Edit Icon* -
  Clicking the edit icon within the bot output chat bubble allows you to quickly specify a new bot response for a given input and save it to a specified aiml file.

- *Advanced Alter* -
  Clicking the drop down next to “New Response” when the Alter Response modal is open will allow you to also define `that` and `topic` values for your new pattern-template pair.

- *Show Metadata* -
  Clicking Show Metadata beneath the bot response bubble will display some important information about the interaction, including the Pattern that matched, the values (if any) for `that` and `topic`, and a link to the file containing the pattern that was matched, which you can click on to go edit directly.

- *Show Trace* -
  Beneath the displayed Metadata, you will also find an option to run a *Trace*. This will open a modal showing the series of steps the input underwent during processing to find a match, which is incredibly helpful for debugging.

- *Show Predicates* -
  At the top of the Chat Widget next to your bot name, a menu icon allows you to manage predicates as well as to reset your bot’s memory (which will clear any predicate values that have been set and allow you to start the conversation anew as if you were a brand new client).

Bot Log Review

Reviewing chat logs and updating your bot frequently is a critical aspect of bot development. You can review logs by clicking “Logs” associated with your bot. Chat logs are displayed dating back 30 days and available for download.

Unread conversations appear in **bold**. Orange conversations contain at least one input that triggered the UDC. Orange highlighted input-output pairs that triggered the UDC are the highest priority log items to correct.

Clicking *Show Metadata* will reveal the same information about the interaction as doing so within the Chat Widget. You can also edit responses from here or by opening the linked file in the editor.

When you are ready to start collecting logs from actual clients, you can publish your bot via the Deploy page.

Bot Deployment

The **Deploy Page**, accessible via the *Deploy* link under each individual bot name displayed on the left nav, lists a variety of *Integrations*. Integrations provide an easy method for deploying your bot to popular third party platforms and channels like Facebook Messenger and the Web. Supported Integrations are determined by a variety of factors including, but not limited to, technical feasibility, ecosystem demand, and platform partner relationships. If you wish to see support added for a channel that is not currently available, please email us at support@pandorabots.com and we'd be happy to consider your request.

Pandorabots also provides a RESTful API, meaning developer users can integrate their chatbot into any application. Please refer to the API REFERENCE section for a list of API Endpoints, SDKs, and more. You may locate your *Application ID* (*"App ID"*), which is the same as your username, your *User Key*, and your *Botkey* by clicking on the "API Keys" button next to the "Custom Application" integration.

The Internal Bot Directory

The Deploy Page provides an option to publish your bot to **The Internal Bot Directory**, which will make your bot available to chat with other platform users. This is the only channel available to Sandbox users (Free tier). In the Directory, accessible via the left nav, you can also chat with other bots whose botmasters have chosen to make their works-in-progress visible. This is a great way to collect chatlogs from other platform users to leverage for improving your bot prior to making it public, and to pay back the favor by chatting with their bots as well.

Basic AIML Training

Input Pre-Processing

Normalization

Corrects some spelling errors and colloquialisms (e.g. "wanna" –>"want to")

- Substitutes words for common emoticons (e.g. ":-)" –>"SMILE")
- Expands contractions (e.g. "isn't" –>"is not")
- Removes intra-sentence punctuation (e.g. "Dr. Wallace lives on St. John St. –>"Dr Wallace lives on St John St.")

We refer to the above substitution steps as Normalization.

These substitutions are contained in a substitution file called normal.substitution.

Sentence Splitting

Normalization also splits sentences based on predefined punctuation characters ".", "!", and "?". It also removes these punctuation marks once the sentences have been split, leaving the inputs punctuation-free.

The bot responds to each sentence individually and forms a result by combining the responses together.

You can customize which punctuation marks determine the end of a sentence by modifying the value of the "sentence-splitting" property in your bot's properties file.

Categories

The AIML File

Create a new AIML files under the Files drop-down within the Editor. When selecting your new files, you'll notice that the Editor comes pre-populated with some code:

```
<?xml version="1.0" encoding="UTF-8"?>
<aiml version="2.0">
<!-- insert your AIML categories here -->
</aiml>
```

XML Primer

AIML is an extension of a standard called XML.

It is written using "tags" (code) and text, which should be familiar to anyone with HTML experience.

Some tags comes in pairs, with some context (text and/or other tags) appearing between, for example:

```
<template>Some string goes here</template>
```

Other tags are "self-closing" and do not require a partner or an inner string, for example:

```
<get name="age" />
```

The AIML File

A category is the basic unit of knowledge in AIML.

A category always contains an input *pattern* and a response *template*. Categories are sometimes also described as *rules*, which you as the botmaster specify to describe how the chatbot should respond to client inputs.

Generally, the more categories you have, the more robust your chatbot will be.

Hello World Example

Let's take a closer look at the fundamental components of a category: the pattern and template.

- **<pattern>** - Matches what the user says.
- **<template>** - What the bot replies.

Code example:

```
<category>
<pattern>HI</pattern>
<template>Hello world!</template>
</category>
```

The above code would result in the following exchange between your bot and the client:

> ***Human:*** *Hi*
> ***Bot:*** *Hello world!*

Explaining the "tags"

<category>
Delineates the beginning and end of a category, which is a unit of knowledge for your bot.

<pattern>HI</pattern>
Defines a pattern that matches a certain input from the client.
_Note: AIML matching is case insensitive, meaning it does not differentiate between capital and lowercase letters. So, if the client said either "hi" or "HI," the bot would still match this category. Using all caps to write the pattern is a convention adopted to make the code more readable. _

<template>Hello world!"</template>
Defines the bot's response to the matched pattern. Case does matter in the template! Your bots response will be returned to the client exactly as written between the template tags.

</category>
Marks the end of the category.

Pattern Matching

The bot will search through all of its categories to form a match with the user input.

IMPORTANT! Remember that the *input pre-processor strips the input of all punctuation*. Therefore, you must not include punctuation marks in your patterns!

WRONG:

```
<pattern>What is your name?</pattern>
```

** CORRECT:**

```
<pattern>WHAT IS YOUR NAME</pattern>
```

HTML Markup

HTML is also an extension of XML.

You can use some HTML to markup your bot's responses (templates) with the appropriate formatting.

You can also use HTML to include images and hyperlinks that lead to other websites. See this page for some HTML basics supported by our platform.

For example, the <br/> HTML tag allows you to insert line breaks in the text.

```
<category>
<pattern>WHO ARE YOU</pattern>
<template>I am a bot.<br/> I live in a computer.
</template>
</category>
```

> ***Human:*** *Who are you?*
> ***Bot:*** *I am a bot.*
> *I live in a computer.*

The Ultimate Default Category (UDC)

What if the user input does not match any of the patterns you have defined?

The *Ultimate Default Category (UDC)* is used by the bot to provide an answer if no other suitable category can be matched. The * in this case will match anything undefined.

```
<category>
<pattern>*</pattern>
<template>I have no answer for that.</template>
</category>
```

Randomized Responses

You can use the **<random>** tag to provide many different responses for the same input pattern. This is especially useful in the UDC because it can provide some variation to the default answer.

```
<category>
<pattern>*</pattern>
<template>
<random>
<li>What was that?</li>
<li>I don’t understand</li>
<li>Can you say that more clearly?</li>
</random>
</template>
</category>
```

In the above example, each time a category is matched, the bot will pick one of the list elements (<li>) at random as it’s response.

Wildcards

Wildcard Basics

In the UDC, we used an asterisk (*) in the pattern to capture the user’s input. This symbol, in AIML, is known as a *wildcard*.

Wildcards are used to capture many inputs using only a single category.

The `*` Wildcard

The `*` symbol is able to capture **1 or more words** in the user input.

```
<pattern>HELLO *</pattern>
```

This pattern would match all of the following inputs:

- Hello there!
- Hello Daniel.
- Hello my good friend.

This pattern would **NOT** match the word “Hello” by itself, because there must be at least one word captured by the `*` to form a match.

The `^` Wildcard

The `^` symbol is also a wildcard, however, it can capture 0 or more words.

```
<pattern>HELLO ^</pattern>
```

** ** This pattern would match all of the following inputs:

- Hello.
- Hello there!
- Hello Daniel.
- Hello my good friend.

Matching Priority

What if both `HELLO *` **and** `HELLO ^` exist? Which one will form a match?

Wildcards are ranked in order of priority, so that certain patterns will take precedence over others.

The ^ has a higher priority, so if the input is “Hello there”, then `HELLO ^` would be matched first.

Exact Matches

If a pattern forms an exact match with the input, the exact match category will take precedence over any containing the `^` or `*` wildcards that it could potentially match.

So, if the input is “Hello there”, and the pattern `HELLO THERE` exists, it will match before the other wildcard patterns `HELLO ^` and `HELLO *`.

The `_` and `#` Wildcards

There are two other wildcards, `_` and `#`. These wildcards take the highest priority when matching.

Even if the input forms an exact match with a pattern, the match can be overridden by a pattern containing one of these wildcards.

```
<pattern>HELLO _</pattern>
```

If the input is “Hello there”, the pattern above will form a match even if `<pattern>HELLO THERE</pattern>` has been defined.

The `_` wildcard is a “1 or more” wildcard, like `*`

The `#` wildcard is a “0 or more” wildcard, like `^`

Wildcard Matching Priority

The graphic below shows the matching priority for all four wildcards, along with a pattern that contains an exact match:

`HELLO #` > `HELLO _` > `HELLO THERE` > `HELLO ^` > `HELLO *`

IMPORTANT! Be very careful when using `#` and `_`, because they will override all other patterns you may wish to match!

Highest Priority Matching

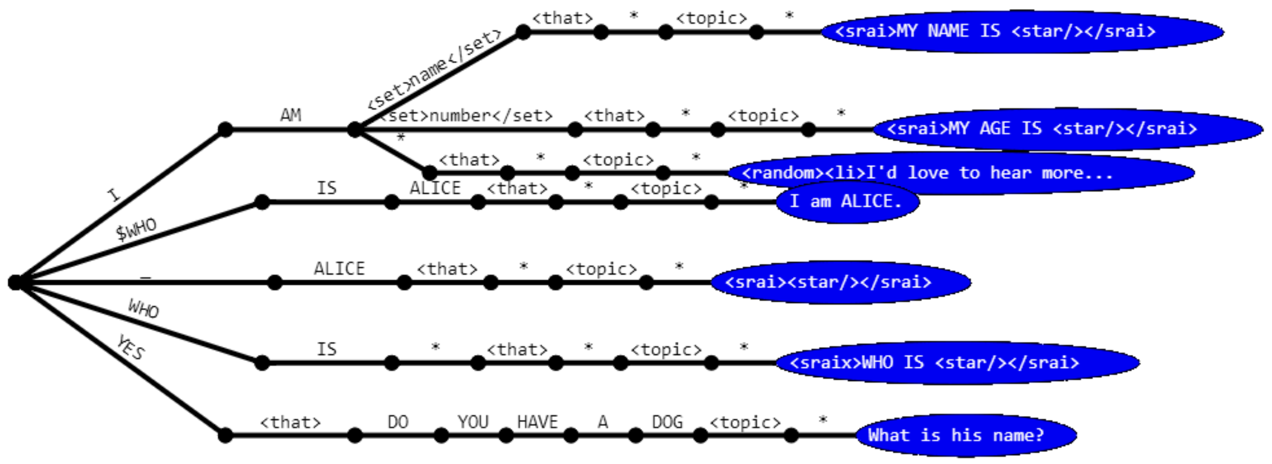
Sometimes, there is an exact match that we would like to take highest priority, overriding the `_` or `#` wildcards. we can use the `$` sign to signify that a pattern will be matched first given a particular word.

For example, `<pattern>$WHO IS MIKE</pattern>` matches “Who is Mike?”, and `<pattern>_ MIKE</pattern>` matches all other inputs ending with “Mike”.

Note that `$` is *not* a wildcard. It is a marker that says “for this particular word(s) - “WHO” in the example above - override the category that would have been otherwise matched.

Visualizing Matching Priority

All of the bot’s AIML categories are loaded into a structure called the *Graphmaster.* The order in which patterns take matching priority can be visualized in the Graphmaster:



“Echoing” Wildcards

You can “echo” the words captured by the wildcard from within the template using the <star/> tag. For example:

```
<category>
<pattern>MY NAME IS *</pattern>
<template>Hello, <star/>.</template>
</category>
```

> ***Human:*** *My name is Daniel.*
> ***Bot:*** *Hello, Daniel.*

Multiple Wildcards

You can have more than one wildcard per pattern. You can also echo multiple wildcards in your pattern by using `<star index="x"/>`, where **x** corresponds to the index number (position in the sentence) of the wildcard:

```
<category>
<pattern>MY NAME IS * AND I AM * YEARS OLD</pattern>
<template>Hi <star/>. I am also <star index=“2”/> years old!</template>
</category>
```

Variables

What are Variables?

In computer programming, a variable is a symbol whose value can be changed.

AIML has variables as well. These can be used to store information about your bot, user, or anything else you would like. There are three types of variables:

- **Properties:** global constants for the bot. These can only by changed by the botmaster.
- **Predicates**: global variables for the bot. These are usually set by the client during conversation when a template is activated.
- **Local Variables:** the same as predicates except their scope is limited to one category.

Using Properties

You can use a property to store your bot’s age. Create a new property with the name “age” and the value “8”. Then, insert this category:

```
<category>
<pattern>HOW OLD ARE YOU</pattern>
<template>I am <bot name=“age”/> years old.</template>
</category>
```

> ***Human:*** *How old are you?*
> ***Bot:*** *I am 8 years old.*

Setting Predicates

Using a predicate variable, you can write a category that will store the name of the client. This category will store the client's name under a predicate called "name":

```
<category>
<pattern>MY NAME IS *</pattern>
<template>Nice to meet you, <set name="name"><star/></set></template>
</category>
```

Note how the user of the `*` wildcard and `<star/>` allows you to write a single category that will capture any name!

Recalling Predicates

Once you have set a predicate, it can be recalled elsewhere in your AIML.

```
<category>
<pattern>WHAT IS MY NAME</pattern>
<template>Your name is <get name="name"/>.</template>
</category>
```

If you have set the predicate using the category in the previous example above, this will now recall the value of the predicate called "name".

In combination, the previous two examples would enable the following conversation:

> ***Human:*** *My name is Daniel.*
> ***Bot:*** *Nice to meet you, Daniel.*
> ***Human:*** *What is my name?*
> ***Bot:*** *Your name is Daniel.*

Using `var`

Local variables work almost exactly like predicates, but their scope is limited to a single category. These are different than predicates, which can be recalled at any time during the conversation. For example:

```
<category>
<pattern>THE APPLE IS *</pattern>
<template>
<think><set var="color"><star/></set></think>
I like <get var= "color"> apples.
</template>
</category>
```

We will revisit local variables in more detail in the section on "Context."

Recursion and Reduction

What is Recursion?

In AIML, you can define a template that calls another category.

This has a wide range of uses:

- Simplifying an input using fewer words
- Linking many synonymous inputs to the same template
- Correcting spelling errors made by the client
- Replacing colloquial expression with proper English
- Removing unnecessary words from the input (*Reduction*)

About half of the categories in a bot use recursion in some way.

The `<srai>` Tag

The `<srai>` tag tells the bot to look for another category:

```
<category>
<pattern>HELLO</pattern>
```

```
<template><srai>HI</srai></template>
</category>
```

If this category is matched (i.e., the input is “Hello”), the bot will recurse. Before returning some text, it will first look for a different category that matches HI.

Using `<srai>`

The <srai> tag effectively translates the input that matches the categories below to “Hi”, contained in the terminal category.

```
<category>
<pattern>HELLO</pattern>
<template><srai>HI</srai></template>
</category>

<category>
<pattern>HI THERE</pattern>
<template><srai>HI</srai></template>
</category>

<category>
<pattern>HOWDY</pattern>
<template><srai>HI</srai></template>
</category>

:

<category>
<pattern>HI</pattern>
<template>Hi, how are you?</template>
</category>
```

The pattern in the terminal category is also commonly known as the ***Intent*****, which is the canonical form of what the words being used fundamentally mean**. In this case, `<srai>` is the method for specifying that Hello, Hi There, and Howdy, all mean the same thing as “Hi” (which we humans generally intend as a Greeting). Much of the work in bot development involves capturing all the different way we humans say the same thing insofar as it relates to your domain, which is why it is so important to regularly review your chatlogs and continously improve your bot over time.

Common Misspellings & Colloquial Expressions

People are bad at spelling and typing, which may cause your bot to fail when trying to find a match. You can use <srai> to account for common spelling mistakes:

```
<category>
<pattern>HOW R U</pattern>
<template><srai>HOW ARE YOU</srai></template>
</category>
```

Synonyms

You can also use `<srai>` in conjunction with wildcards to define synonymous words or phrases:

```
<category>
<pattern>_ DAD *</pattern>
<template><srai><star/> FATHER <star index=“2”/></srai></template>
</category>
```

Anytime the user input contains the word “dad”, the bot will replace it with “father” and recuse using the same input.

Why Synonyms?

Thesaurus.com lists 52 synonyms for the word “good”. To account for this, you would need 52 additional categories for every one that contains the word “good”.

If your bot has 100 patterns that contain the word “good”, that’s 5200 additional categories you would have to write. Using the synonyms technique, you can reduce that number to just 52.

NOTE: once a word has been defined as a synonym, you cannot use it in patterns. The leading underscore `` ensures that the bot translates the synonym before doing anything else._

Reduction

We can also use `<srai>` to remove unnecessary words from the input.

```
<category>
<pattern>I SEE NOW THAT YOU ARE *</pattern>
<template><srai>YOU ARE <star/></srai></template>
</category>
```

Reductions make writing AIML and adding to your bot a far more enjoyable process. The more reductions you have, the better your bot will be at providing relevant matches.

Returning Text and Recursing

The previous examples of `<srai>` have directly returned no text of their own. You template, however, can return both text and `<srai>` tags.

```
<category>
<pattern>HOWDY</pattern>
<template>
<srai>HELLO</srai>
Are you a cowboy?
</template>
</category>
```

Sets and Maps

Sets

An AIML Set is a list of unique text strings. You can create a set using the File drop-down in the Editor.

A set called “colors” might contain the following:

- red
- orange
- yellow
- green
- blue
- purple
- …

Sets are used to dramatically reduce your bot’s overall categories. Consider the following conversation:

> ***Human:*** *Is green a color?*
> ***Bot:*** *Yes, green is a color.*
> ***Human:*** *Is blue a color?*
> ***Bot:*** *Yes, blue is a color.*
> ***Human:*** *Is peanut butter a color?*
> ***Bot:*** *No, peanut butter is not a color.*

Imagine how many categories would be needed to cover every color in the spectrum!

Instead of giving each color its own category, we can create a set that contains a number of colors, and write a single category that checks to see if the user’s input contained a color in the set.

This category will only be matched *if* the user’s input does, in fact, contain one of the colors in the set. If the user’s input does not contain one of the colors listed in the set, the category will not be matched.

How Sets Work

The set functions like a wildcard. It captures one or more words found in the user’s input.

The name of the relevant set file is placed between the `<set>` tags.

```
<category>
<pattern>IS <set>colors</set> A COLOR</pattern>
```

```
<template>Yes, <star/> is a color.</template>
</category>
```

If the input contained a string not found in the set, then that pattern would not form a match. To account for this possibility, we can provide a default answer using a `*` wildcard.

```
<category>
<pattern>IS <set>colors</set> A COLOR</pattern>
<template>Yes, <star/> is a color.</template>
</category>

<category>
<pattern>IS * A COLOR</pattern>
<template>No, <star/> is not a color.</template>
</category>
```

The first category will match if the input contains a string found in the set.

The second category will match if the input contains a string NOT found in the set.

More on Sets

Sets take precedence over * and ^, but can be overridden by `_`, `#`, and an exact word match.

The input captured by the `<set>` tags can be echoed using the `<star/>` tag, just like a wildcard can be echoed.

```
<category>
<pattern>IS <set>colors</set> A COLOR</pattern>
<template>Yes, <star/> is my favorite color!</template>
</category>
```

Set files are a simple string array and are written in the following format:

```
[[ "Austin" ], ["Baltimore"], [ "Chicago" ], [ "Dallas"]]
```

Maps

A map is a list of key-value pairs used to form associations between words. You can create a map using the Files drop-down in the editor.

A map called “statecapitals” might look like this:

- California: Sacramento
- New York: Albany
- Texas: Austin
- …

Maps are accessed from within the template. They are used in conjunction with a set that contains a list of its keys (the words on the left of the colon above). For example, a set that corresponds to the “statecapitals” map would be called “states”:

- California
- New York
- Texas
- …

Maps: Test Case

Consider the following conversation:

> ***Human:*** *What is the capital of California?*
> ***Bot:*** *Sacramento is the capital of California.*
> ***Human:*** *What is the capital of New York?*
> ***Bot:*** *Albany is the capital of New York.*
> ***Human:*** *What is the capital of Texas?*
> ***Bot:*** *Austin is the capital of Texas.*

We can enable this conversation with a single category, one set, and one map:

```
<category>
<pattern>WHAT IS THE CAPITAL OF <set>states</set></pattern>
<template>
<map name=“statecapitals”><star/></map> is the capital of <star/>
</template>
</category>
```

Feel free to try creating the set and map files we discussed, and then add this category to your AIML. Your bot will now be able to relay the capital of the states listed in your set and map.

More on Maps

Like sets, we can include a “default” category when the wildcard contents do not match an item in the map.

```
<category>
<pattern>WHAT IS THE CAPITAL OF *</pattern>
<template>
I don’t know what the capital of <star/> is.
</template>
</category>
```

> ***Human:*** *What is the capital of Pennsyltucky?*
> ***Bot:*** *I don’t know what the capital of Pennsyltucky is.*

Map files are simple key-value pairs in an array, and are written in the following format:

```
[["Texas" , "Austin"], ["California" , "Sacramento"]]
```

Built-in Sets and Maps

Pandorabots has some pre-built sets and maps that are not visible from the editor.

###Is this still true in uUI?

`<set>number</set>` - matches any natural number

`<map name=”successor”>` - maps any number n to n+1

`<map name=”predecessor”>` - maps any number to n-1

`<map name=”plural”>` - (attempts to) find the plural form of a singular noun (English only)

`<map name=”singular”>` - (attempts to) find the singular form of a plural noun (English only)

Context

What is Context?

When humans have a conversation, we are able to remember the things that have been previously said.

> ***Human 1:*** *Do you like coffee?*
> ***Human 2:*** *Yes.*

Human 1 knows that “Yes” is a response to their question, because they have said it in particular context.

There are several features in AIML that allow you to provide context within your category.

The `<that>` Tag

The <that> tag, which sits between the pattern and template, enables the bot to remember that last sentence it uttered.

A category containing a `<that>` statement will ONLY be matched if the contents between the tags match the last sentence given by the bot.

This allows you to have many duplicate patterns which, depending on the previous response, will trigger different templates.

Using `<that>`

In the example below, the second category is bound to a particular context. It will only be matched *if* the last sentence the bot said was “Do you like coffee?”.

```
<category>
<pattern>^ COFFEE ^</pattern>
<template>Do you like coffee?</template>
</category>

<category>
<pattern>YES</pattern>
<that>DO YOU LIKE COFFEE</that>
<template>Do you prefer dark or medium roast?</template>
</category>
```

> ***Human:*** *I should drink some coffee.*
> ***Bot:*** *Do you like coffee?*
> ***Human:*** *Yes.*
> ***Bot:*** *Do you prefer dark or medium roast?*

More on `<that>`

The previous sentence stated by your bot will be stripped of punctuation when read by a category referencing if with `<that>` tags.

The new category will also ignore differences in capitalization / non-capitalized letters, and will expect any normalization you have specified in the normal.substituion file.

For these reasons, you must write the contents of your `<that>` tags like you write patterns: no punctuation, all capital letters, and normalized. If you `<that>` tags are failing, it is usually because one of these three items is amiss.

Note: `<that>` tags can also contain wildcards! Values for `<that>` variables are only valid within the scope of the active conversation.

Topic

There is a built-in predicate variable called `topic`. Categories can be grouped together based on different values for `topic`. These categories can only be matched if the topic predicate has been set to a certain value.

`Topic` allows your bot to keep context for longer than one interaction (the function of the `<that>`tag). This can also be used to write duplicate patterns whose templates vary depending on the context of the conversation.

Using Topic

Step 1:

First, you need a category that sets the topic predicate. The following example would set the topic to “coffee”:

```
<category>
<pattern>CAN WE TALK ABOUT COFFEE</pattern>
<template>Do you like <set name="topic">coffee</set>?</template>
</category>
```

Step 2:

Now, you can group together categories within the coffee topic. Note that `<topic>` tags appear outside the categories. _In AIML 2.0, `<topic>` can appear inside the category, like `<that>`.

```
<topic name= "coffee">

<category>
<pattern>I LIKE THE TASTE</pattern>
<template>I love how coffee tastes.</template>
</category>

<category>
<pattern>IT IS TOO BITTER</pattern>
<template>Maybe you should drink tea.</template>
</category>
```

```
</topic>
```

The patterns above will *only match if* the topic has been set to “coffee.” If no category within the topic tags forms a match with the input, the input will match a category with no topic specified.

More on `<topic>`

The default value of topic is `*`. If you set a topic with no value (i.e., `<set name="topic"></set>`) the value will be “unknown.”

Values for topic variables are only within the scope of an active conversation.

Conditionals

The values of predicates and local variables provide a third type of context in AIML. Using the `<condition>` tag, a bot can respond differently to the same input depending on the value of a predicate or local variable.

The concept is the same as an IF - THEN - ELSE statement found in most programming languages.

Conditionals: Test Case I

Consider the following:

> ***Human:*** *Today is Monday.*
> ***Bot:*** *The start of the work week!*
> ***Human:*** *Today is Tuesday.*
> ***Bot:*** *Tuesday already?*
> ***Human:*** *Today is Wednesday.*
> ***Bot:*** *Humpday, we’re halfway to the weekend!*

The bot response is *conditioned* on the day of the week. Using the <condition> tag within the template enables this with a single category:

```
<condition name=“today”>
<li value=“Monday”>...</li>
<li value=“Tuesday”>...</li>
<li value =“Wednesday”>...</li>
</condition>
```

The opening tag specifies the name of a predicate to check for; *if* the value of the predicate matches the value of any list element (`<li>`), _then _the text of that element will be returned.

Altogether, the category for this test case would look like this:

```
<category>
<pattern>TODAY IS *</pattern>
<template>
<think><set name=“today”> <star/></set></think>
<condition name=“today”>
<li value=“Monday”>Ah. The start of a new week.</li>
<li value=“Tuesday”>Tuesday already?</li>
<li value=”Wednesday>Humpday, halfway to the weekend!</li>
...
<li>That isn’t the name of a day!</li>
</condition>
</template>
</category>
```

The final list element (the one without a value attribute) will be returned if none of the other conditions are met.

Conditionals: Test Case II

You can also use conditionals to check the status of a predicate, i.e., whether or not it has been set.

```
<category>
<pattern>WHAT IS MY NAME</pattern>
<template>
<condition name=“name”>
```

```
<li value="*">Your name is <get name="name"></li>
<li>You haven't told me your name yet!</li>
</condition>
</template>
</category>
```

If the "name" predicate has been set to anything (denoted by the asterisk), the first list element will return; if it has not been set, then the second list element will return.

Attributes v. Tags

In AIML 2.0, any given value given by an XML attribute may also be expressed using a subtag of the same name. For example, `<li value="X">` may also be written as `<li><value>X</value>`. This makes it possible to vary the values of attributes using XML expressions, for example:

```
<category>
<pattern>IS * EQUAL TO *</pattern>
<template>
<think><set var="star"><star/></set></think>
<condition var="star">
<li><value><star index="2"/></value>Yes</li>
<li>No.</li>
</condition>
</template>
</category>
```

Loops

Loops are used in programming to iterate an action or function over a series of values, until a particular state has been reached. In AIML, we can loop over list elements in a condition until a certain value has been reached, at which point the loop will terminate and the bot will give a response.

Let's say we want to recreate the following interaction:

Human: Count to 8
Bot: 1 2 3 4 5 6 7 8

We will be using the "number" set and the "successor" map to establish the relationships between numbers.

First, set up the pattern:

```
<pattern>COUNT TO <set>number</set></pattern>
```

Next, you'll want to set a local variable in your template. The variable starts your count at 0. As we loop over our list elements, this count will change:

```
<think><set var="count">0</set></think>
```

Next, we need to set up a condition whose list elements are dependent on the value of the variable "count":

```
<condition var="count">
```

Then, we create an empty list element whose value is `<star/>`. This element will be returned when the value of "count" reaches the number specified in the input:

```
<li><value><star/></value></li>
```

Our second list element will contain the loop. Using the "successor" map, we can simultaneously change the value of count, and add it to the bot's response:

```
<li>
<set name="count">
<map><name>successor</name><get var="count"/></map>
</set>
<loop/>
</li>
```

This resets the value of "count" to the corresponding number of its current value, found in the "successor" map. It will then add the successive value to the bot's response. Finally, `<loop/>` tells it to return the list element again, however this time, with the new value of “count”.

Altogether, the above components form the following code:

```
<category>
<pattern>COUNT TO <set>number</set></pattern>
<template><think><set var="count">0</set>
</think>
<condition var="count">
<li><value><star/></value></li>
<li><set var="count"><map><name>successor</name><get name="count"/></map></set> <loop/></li>
</condition></template>
</category>
```

When the second `<li>` has looped enough time for “count” to equal 8, the first list item will be returned and the loop will terminate. The bot will then respond to all of the text returned by the second list item (notice that when the “count” variable is reset, there is no `<think>` tag).

Learning

What is Learning?

AIML 2.0 has features that allow your clients to teach the bot new information.

Using the `<learn>` and `<learnf>` tags, clients can actually generate new categories from within their conversation.

Categories learned using `<learn>` will only be accessible to that particular conversation, and will eventually be cleared after some idle time of no interaction with the bot.

Categories learned using `<learnf>` will be written to a new AIML file and can be accessed by anyone talking with your bot. (Careful! Your clients may teach the bot naughty things, so use `<learnf>` with extreme caution.)

How Learning Works

When the category below is matched, the bot will now have access to the category found within the <learn> tags:

```
<category>
<pattern>THE SKY IS BLUE</pattern>
<template>I will remember that the sky is blue
<learn>
<category>
<pattern>WHAT COLOR IS THE SKY</pattern>
<template>The sky is blue</template>
</category>
</learn>
</template>
</category>
```

Learning involves *nested* categories. To access variables defined in the outer category from the inner category, we use the `<eval>` tags. Anything found within the <eval> tags will be evaluated FIRST, before the new category is actually created. This allows us to use `<star/>`within the `<learn>` tags to access parts of the input matched by the outer category.

```
<category>
<pattern>THE * IS BLUE</pattern>
<template>I will remember that the <star/> is blue
<learn>
<category>
<pattern>WHAT COLOR IS THE <eval><star/></eval></pattern>
<template>The <eval><star/></eval> is blue</template>
</category>
</learn>
</template>
</category>
```

We can now reference the words captured by the wildcard in our new category.

Revisiting Key Platform Components

The Chat Widget

In addition to allowing you to converse with your bot as if you were the end-user, the Chat Widget also displays some vital information about each interaction and allows you to train your chatbot by making changes to the AIML within the interface.

Beneath each message bubble containing the bot's response, there is an option to *Show Metadata*. Displaying the metadata will show you the following information:

- **Pattern:** The pattern that matched your input and triggered the output in the message bubble
- **That:** Displays the previous response from the bot (blank if you have just started the dialog)
- **Topic:** Displays the topic if one has been set
- **File:** A link to the file containing the relevant file, which you may click on to edit that file directly in the Editor

Beneath the above metadata, you also have the option to run a Trace. Running a Trace will show you the exact series of steps that led from the input to your bot's output, which can be vital for debugging purposes.

You may also edit the bot response directly from within the chat widget by selecting the edit icon within the bot message bubble. This will open the *Alter Response* modal and prompt you to define a new response (template) and save it to a given `.aiml` file.

Clicking on the arrow icon next to the *New Response* input field will open additional fields allowing you to directly edit the Pattern, Template, That, and Topic values. ###Should these be typed with/without tags? Not clear in the interface!

Retyping your original input after altering the bot response should return the new template you defined.

To reset the bot's memory, which will clear any predicates stored about the current conversation, click on the refresh icon next to your bot's name on the top bar of the chat widget.

The Directory and `<sraix>`

The `<sraix>` tag (first introduced in the AIML 2.0 spec) allows your bot to access the categories of another bot. The tag is very similar to `<srai>`, however, instead of searching your own bot for another category to match, it will search the bot you have specified in the attribute.

You can use `<sraix>` to leverage the knowledge bases' of other bots in the Directory, specifying them based on their botid (username/botname). If you have published your bot to the Directory, and chosen to make your botid available, other platform users may use `<sraix>` to link up with your bot.

Why is this useful? Imagine another platform user has created a bot that is an expert about anything related to coffee botid: unXXXXXXXX/coffeebot).

If this bot has been published to the Directory, your bot can access its expertise (contained in its AIML categories) by sending it any input that relates to coffee:

```
<category>
<pattern>I LIKE COFFEE</pattern>
<template><sraix bot="johndoe/coffeebot">COFFEE</sraix></template>
</category>
```

If coffeebot is able to form a match, your bot will return coffeebot's output as if it were its own output to the client.

Bot Libraries

Bot Libraries are opensource AIML files that provide some base content you can build on as an alternative to building a Blank Bot from scratch. Currently, you can add the Small Talk library to English language bots. Please visit the Rosie Tutorial to learn more about how to leverage this Library.

Additional AIML Features

List Processing

Use the `<first>` and `<rest>` tags to access certain words in the input:

`<first>A B C</first>` = A
`<rest>A B C</rest>` = B C

Note that a "word" in AIML is separated from other words using a space. These tags will have no effect if the content is (1) empty or (2) a single word.

`<rest></rest>` = NIL

Substitutions

Substitutions are used to “normalize” interactions that involve pronounces, punctuation, gender, etc. You can call a substitution by using the file’s name within a tag, for example: `person.substitution` = `<person>`.

```
<category>
<pattern>I AM *</pattern>
<template>
YOU ARE <person><star/></person>
</template>
</category>
```

> ***Human:*** *I am waiting for you.*
> ***Bot:*** *You are waiting for me.*

In this category above, the substitution file “person” has found a 2nd-person pronoun in the wildcard contents and converted it to a 1st-person pronoun when echoed in the template.

This works in reverse as well.

Pronoun-based Substitutions

The example above made use of the file `person.substitution`, which is used to transform pronouns between first and second person.

`person2.substitution` is used to transform between first and third person pronouns.

`gender.substitution` is used to transform between male and female gender pronouns.

Denormalization

Earlier in the tutorial, we introduced the concept of input pre-processing or “normalization”. This involved the correction of common misspellings, contractions, and most importantly, the removal/replacement of punctuation found in the input.

This means that punctuation is not “preserved” by simply echoing part of the user’s input, for example:

```
<category>
<pattern>URL *</pattern>
<template><star/></template>
</category>
```

> ***Human:*** *URL google.com*
> ***Bot:*** *google dot com*

To preserve punctuation found in the input we use `denormalize.substitution`. This file contains all of the words that punctuation marks are normalized to. From the previous example, we know that the string “.com” is normalized to “dot com”. To reverse this substitution, we echo the user’s input within the denormalize tags:

```
<category>
<pattern>URL *</pattern>
<template>
<denormalize><star/></denormalize>
</template>
</category>
```

> ***Human:*** *URL google.com*
> ***Bot:*** *google.com*

###More relevant example using emoji?

The Default Substitutions Files

Substitutions are available to you via the substitution files provided by default. We strongly suggest that you ***do not remove*** any of these substitutions. The contents of these files will only become applicable if you reference them in your template, so you should keep them in tact in case you wish to use any of the operations. You may also add to or update your substitution files.

Date

You can return the current date in the format of your choice, specified like the arguments to the LINUX strftime function. For example, `<date format="%B %d, %Y />` will return today's date.

Interval

You can use `<interval>`, `<from>` and `<to>`, and `<style>` to calculate the internal between two dates (in this case, the bot's birthdate property and today's date):

```
<category>
<pattern>AGE IN YEARS</pattern>
<template><interval>
<style>years</style>
<from><bot name="birthdate"/></from>
<to><date format="%B %d, %Y" /></to>
</interval></template>
</category>
```

Next Steps: Deploying your Bot

Congratulations! If you've made it this far, we hope you are well on your way to becoming a proficient botmaster.

After a period of initial development, you may find yourself wanting to share your bot with the world by unleashing it on popular messaging or voice channels. There are a number of different ways to make bots built on the Pandorabots Platform public.

Downloading your Bot Files

You may download your bot files at any time via the Files tab in the Editor, and we recommend making frequent, regular backups. Since AIML is an open-standard, there are a number of opensource packages, including some interpreters, you can use. Pandorabots is one of the few platforms that is an open system (not a black box) and therefore actually exposes your code to you and allows you to download it and make full use of it elsewhere.

Of course, writing your own interpreter or attempting to make use of opensource projects with little or deprecated support can be incredibly challenging and time consuming, which is why Pandorabots offers its state-of-the-art interpreter as a service and provides API access to its hosting platform and SDKs

Using the Pandorabots API

If you are a developer or have one on staff, you may use the Pandorabots API to integrate your chatbot into any application. Visit the API Reference section to learn more.

Using the Turnkey Platform Integrations

The easiest way to make your bots available to the public (in a few simple steps!) is via the Deploy Page, accessible from the menu that appears whenever an individual bot is selected. Simply navigate to the interface and follow the instructions listed for each available Integration you wish to use.

Updating your Bot

Regularly reviewing chatlogs and continuously updating your chatbot based on what clients are actually saying is a critical component of bot development. In fact, we often say that the *real work* starts after your initial launch, because you cannot design a conversational experience in a vacuum. Even if you have data like call center or live agent chat logs, you still cannot predict everything that your end-users are going to say.

Visit the Logs page for your bot to review conversation logs, and make updates based on high priority logs (which appear at the top) where your bot didn't have a sufficient answer.

Frequently updating your bot to account for all the possible things a human might say in conversation may seem like a daunting task. However, you may rest assured that in reality - and especially with regard to a particular domain - people actually tend to say the same thing in conversation most of the time. This phenomenon is known as Zipf's Law, which explains a curve that represents the frequency at which words appear in human language.

As you can see, there is a word that occurs most frequently, followed by another word that occurs next most frequently, followed by a third word that occurs less frequently and so on. For a given body of text, it doesn't matter whether you break the text into letters, words, phrases, or sentences: the same distribution will always appear for natural languages.

In the course of bot development, you will find that many people say the same thing or ask the same types of questions, so you only need to define a finite number of rules to create a decent bot.

Happy bot-making!

Specific Use Cases

Example AIML and guidance for building bots for specific use cases.

Developing a simple FAQ chatbot in AIML

A Frequently Asked Questions (FAQ) chatbot connected to a messaging platform or to your website, is a great use case for a Pandorabot. We've come up with an approach to build a quick FAQ chatbot. The purpose of this article is to step you through the thought process of this approach for a limited set of FAQs.

This process assumes you understand basic AIML terminology such as categories, symbolic reductions and wildcards.

Step 1: Gather your list of questions and answers

If you have chat or inquiry history with your customers, you might have a list of frequently asked questions, as well as the most common ways your customer asks a question that results in the same answer. For example, these questions may all result in the same answer as shown below:

- How do I sign up?
- How do I sign up for your services?
- I want to sign up now
- Where do I signup?
- Where do I sign up for your service?
- I don't know how to sign up
- I'm having problems signing up

Example answer:

- Go to fabcompany.com/services and click the SIGN UP button in the middle of the page.

Make sure that all the questions you've listed actually map to the same answer. For instance, you might want to provide a different answer to "I'm having problems signing up", such as "Please contact our support staff at XXX-XXX-XXXX".

Note: using a chatbot to solicit more information from your customer to identify a problem is a different chatbot use case. These approaches can be combined into a single Pandorabots though!

Step 2: Develop Canonical Forms

In the example above, you have one answer and at least 6 different ways to ask the same basic question about "sign up". We call the base input, the canonical form (also known as "intents").

You can pick one question as your canonical form, and build the base category:

```
<category>
  <pattern>HOW DO I SIGN UP</pattern>
  <template>Go to fabcompany.com/services and click the SIGN UP button in the middle of the page.</template>
</category>
```

Alternatively, we recommend using a naming convention with business name/department and type of question, such as FABSERVICESSIGNUP.

```
<category>
  <pattern>FABSERVICESSIGNUP</pattern>
  <template>Go to fabcompany.com/services and click the SIGN UP button in the middle of the page.</template>
</category>
```

Step 3: Develop Symbolic Reductions

There are several ways to develop Symbolic Reductions (the "many patterns, one reply" concept) as shown below.

Note on pattern and normalization:
Patterns must be stripped of punctuation and any other normalization that you have specified in your bot (i.e. normal.substitution file). For example, do not include "." (period) or "?" (question mark) and also if you have the substitution in your normal file, "don't" should be "DO NOT" in your pattern tags.

Exact Match

If you have a list from chat history, you can just keep adding each question as a symbolic reduction (using srai AIML tag).

Example AIML would look like:

```
<category>
  <pattern>HOW DO I SIGN UP FOR YOUR SERVICES</pattern>
  <template><srai>FABSERVICESSIGNUP</srai></template>
</category>
<category>
  <pattern>I WANT TO SIGN UP NOW</pattern>
  <template><srai>FABSERVICESSIGNUP</srai></template>
</category>
<category>
```

```
  <pattern>WHERE DO I SIGNUP</pattern>
  <template><srai>FABSERVICESSIGNUP</srai></template>
</category>
     :
     :
```

This is simple but is not as flexible. This solution would not take into consideration typos or unconventional phrasing.

Wildcards & Keywords

Using wildcards in AIML pattern matching for your symbolic reductions can be more flexible. Start by identifying common words (i.e. keywords) in all your questions with the same answers. For example:

```
<category>
  <pattern>_ SIGN UP _</pattern>
  <template><srai>FABSERVICESSIGNUP</srai></template>
</category>

<category>
  <pattern>_ SIGNUP _</pattern>
  <template><srai>FABSERVICESSIGNUP</srai></template>
</category>
```

These reductions would apply to all 6 of the original questions. You could even add reductions that addresses a very common misspelling of your keyword(s). For instance, the word “sign” is commonly misspelled as “sing” and “sigh”

```
<category>
  <pattern>_ SING UP _</pattern>
  <template><srai>FABSERVICESSIGNUP</srai></template>
</category>
<category>
  <pattern>_ SIGH UP _</pattern>
  <template><srai>FABSERVICESSIGNUP</srai></template>
</category>
```

The limitations of this approach is that it requires some human analysis, and too few words in the reduction can be less accurate. For instance, if someone inputs “I did sign up and hate it!”, the chatbot’s response would not work well if using the FABSERVICESIGNUP canonical form.

The great thing about AIML is that you can mix up these approaches building categories for your Pandorabot.

Step 4: Putting it all together

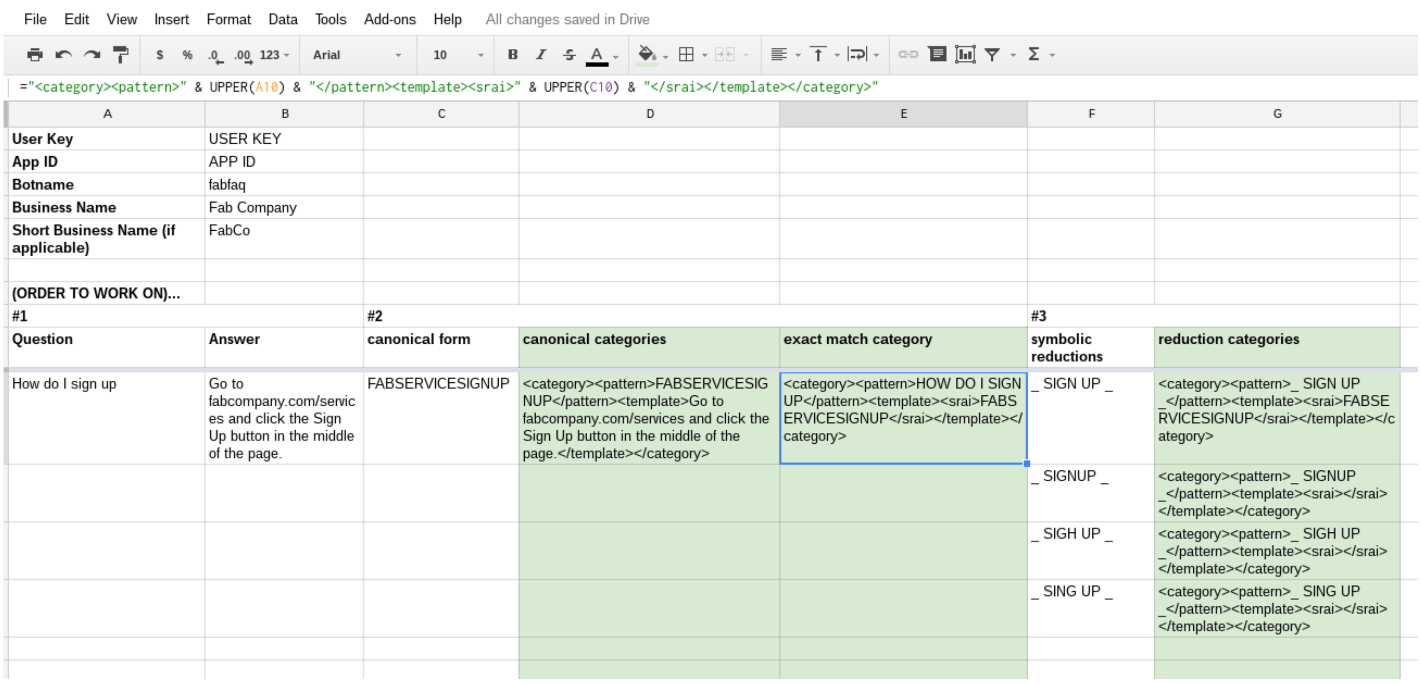
Follow this process for each question and answer pair. For example, try these additional questions on your own.

- How do I get paid?
- How do I cancel your services?
- I need to speak to a real person.
- Who is your CEO?

Add all your categories to your Pandorabot, compile and start training your bot to see if you can break it!

TIPS TO STREAMLINE THIS PROCESS

In the past, using a spreadsheet to automate building categories with AIML tags, and creating a tab-separated value file to upload into our bot on the dashboard made the process a bit easier. We now offer bot editting tools such as Intents Tree and Beta feature Chat Design for a more visual experience.


File Edit View Insert Format Data Tools Add-ons Help All changes saved in Drive
Arial
10

```
="<category><pattern>" & UPPER(A10) & "</pattern><template><srai>" & UPPER(C10) & "</srai></template></category>"
```

A	B	C	D	E	F	G
User Key	USER KEY					
App ID	APP ID					
Botname	fabfaq					
Business Name	Fab Company					
Short Business Name (if applicable)	FabCo					
(ORDER TO WORK ON)...						
#1		**#2**			**#3**	
Question	**Answer**	**canonical form**	**canonical categories**	**exact match category**	**symbolic reductions**	**reduction categories**
How do I sign up	Go to fabcompany.com/services and click the Sign Up button in the middle of the page.	FABSERVICESIGNUP	<category><pattern>FABSERVICESIGNUP</pattern><template>Go to fabcompany.com/services and click the Sign Up button in the middle of the page.</template></category>	<category><pattern>HOW DO I SIGN UP</pattern><template><srai>FABSERVICESIGNUP</srai></template></category>	_ SIGN UP _	<category><pattern>_ SIGN UP _</pattern><template><srai>FABSERVICESIGNUP</srai></template></category>
					_ SIGNUP _	<category><pattern>_ SIGNUP _</pattern><template><srai></srai></template></category>
					_ SIGH UP _	<category><pattern>_ SIGH UP _</pattern><template><srai></srai></template></category>
					_ SING UP _	<category><pattern>_ SING UP _</pattern><template><srai></srai></template></category>

Building a Concierge-style bot

The “Concierge” chatbot is actually a network of bots designed for the purpose of making recommendations about a service or product to the customer. We’ve built a music concierge that can recommend songs and artists to you based on your mood or current activity.

The network is made of up three distinct bot types, each of which takes on a different role:

- Personal
- Comparison engine
- Artist “expert”

We’re going to take a look at how each of these bots work, as well as some of the AIML required for them to do so! Grab the code here to follow along.

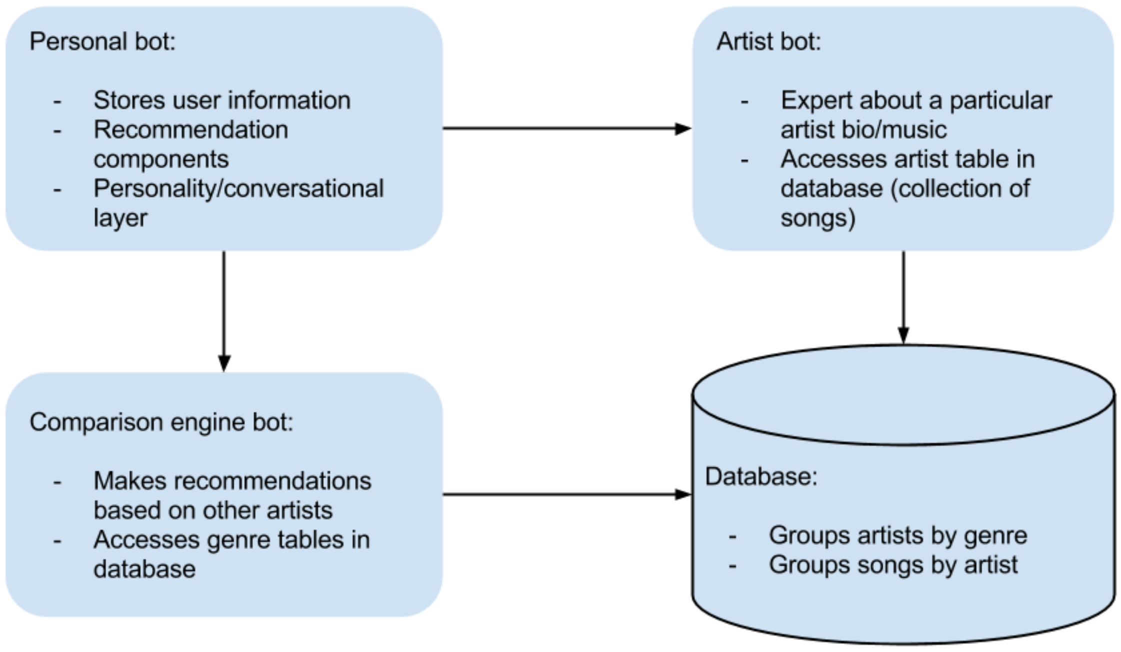
The following AIML elements are featured heavily in this tutorial:

- `<srai>`
- `<sraix>`
- `<condition>`
- `<that>`
- `<formal>`

Make sure you are familiar with the high level concepts via the AIML Reference before you begin.

> ***Note:*** *this music-oriented bot is just an example, but you can take these design patterns with you when building your own Concierge-style bot.*

Architecture


Personal bot:
- Stores user information
- Recommendation components
- Personality/conversational layer
Artist bot:
- Expert about a particular artist bio/music
- Accesses artist table in database (collection of songs)
Comparison engine bot:
- Makes recommendations based on other artists
- Accesses genre tables in database
Database:
- Groups artists by genre
- Groups songs by artist

The concierge bot network has a simple architecture that separates concerns by dividing tasks between the bots.

The_Personal bot_is the front-end for the network. This is the bot that your users will be talking to, and has the responsibility of storing information that your users share with it. This information can be reused later when making a recommendation.

The_Comparison engine_is a bot that mimics a database which organizes musicians by genre. This allows the network to make a recommendation based on the user's preference for a particular artist. We've given you a "pure AIML" solution in this example, but in reality you will probably want the power of a query language like SQL to fetch data from your database. It is not designed to be "talked" to - rather, it can be used as a utility by the personal bot in an `<sraix>` block.

Finally, the_Artist expert_is a bot that is knowledgeable about a particular artist or musician. Again, this bot is not really designed to be talked to directly - it is just a bot that can return information that the Personal bot will share with the user.

Comparison engine

Let's first take a look at the comparison engine so that we can have a little context for how the personal bot is used. The comparison engine is a bot that exposes two patterns to other bots.

The first pattern is XSIMILAR *, which accepts the name of an artist in the wildcard. It will return the name of another artist that makes music in the same genre as the artist who was provided. For example:

```
Input: XSIMILAR Kendrick Lamar
Output: Kanye West
```

> ***Note:*** *the X prefix in some patterns is used to indicate that a pattern is intended to be a "private" category - in other words, these categories are used by the bot and it is not expected the user will ever type something like XSIMILAR. You may create your own private categories in this vein, and may use anything in the place of X (say, your initials), so long as the text is highly unlikely to appear in a user input.*

The second pattern is XRANDOM * BLOCK ^, which accepts the name of a genre, and returns a random artist from that genre. The BLOCK ^ portion is designed so that you may optionally block an artist from being selected.

```
Input: XRANDOM hiphop BLOCK Tyga
Output: Iggy Azalea
```

You'll also notice the use of the `<set>` tags in the pattern of these categories. Each set file (classical.set, edm.set, etc.) is just a grouping of artists by genre. To add more artists to the comparison engine, just find the correct set and add their name to the end!

This bot is really a "utility" - in other words, it's not meant to be contacted directly by your user. Instead, it is to be used via `<sraix>` in another bot. Careful placement in the template of your calling bot can yield seamless insertion of the output of the utility bot:

```
<category>
  <pattern>I LIKE TO LISTEN TO DRAKE</pattern>
  <template>
    If you like Drake, you might also like <sraix bot="djf/musiccat">XSIMILAR DRAKE</sraix>
  </template>
</category>
```

Artist expert

As an example of an “expert” bot, we’ve also included a bot that has some information specific to an artist. Drakebot only has 2 categories:

```
<category>
  <pattern>XBIO</pattern>
  <template>
    Aubrey Drake Graham (born October 24, 1986) is a Canadian rapper,
    singer, songwriter and actor. He was born in Toronto, Ontario. He first
    garnered recognition for his role as Jimmy Brooks on the television series
    Degrassi: The Next Generation. He later rose to prominence as a rapper,
    releasing several mixtapes before signing to Lil Wayne's Young Money
    Entertainment in June 2009.
  </template>
</category>

<category>
  <pattern>XRANDOMSONG</pattern>
  <template>
    <random>
      <li>0 To 100</li>
      <li>The Motto</li>
      <li>Marvin's Room</li>
      <li>Hold On, We're Going Home</li>
      <li>Worst Behavior</li>
      <li>6 God</li>
    </random>
  </template>
</category>
```

The first category is an example of how you might return some biographical information about the artist in question. The second category is an example of how you might store additional information about an artist like their playlist - here, the category is designed to return the name of a random song. You could build on this by including a link to the song on a streaming service, which might actually play the song when you click.

This bot is also a utility - rather than talking to it directly, we can use `<sraix>` to call its private categories and insert its responses into another bot:

```
<category>
  <pattern>TELL ME ABOUT DRAKE</pattern>
  <template><sraix bot="djf/drakebot">XBIO</sraix></template>
</category>
```

Personal bot

Now that we’ve explained some of the bots in our network, let’s take a look at the centerpiece: the_Personal bot_. The personal bot is perhaps the most important piece in the Concierge model - this bot is the user’s gateway to using other components like the comparison engine and the artist experts. The personal bot has access to information about the user, and can make decisions based on this information as to which bot may have a response to the user’s input.

This bot also contains many_reductions_, which are categories that attempt to “reduce” or parse what the user has said into something the bot might understand.

```
<category>
  <pattern>HELLO THERE</pattern>
<template><srai>HI</srai></template>
</category>
```

In this reduction, the pattern HELLO THERE does not return any text of its own, but instead uses `<srai>` to insert the return value of a different category (HI). We won’t be sharing all of these in the tutorial because that would make it far too long, but you can check them out in the code on Github.

The personal bot is made up of a number of AIML files:

- activity.aiml - make recommendations based on what the user is doing
- artists.aiml - talk about musicians
- main.aiml - ultimate default category, prompts
- mood.aiml - make recommendations based on the user’s mood
- recommend.aiml - make recommendations based on the user’s favorite artist

main.aiml

```
<!-- UDC - catches inputs that don't match elsewhere -->
<category>
  <pattern>*</pattern>
  <template>I don't understand. Let's try something else. <srai>XPROMPT</srai></template>
```

```
</category>

<!-- HI - greetings redirect to XPROMPT -->
<category>
  <pattern>HI</pattern>
  <template>Hello there! <srai>XPROMPT</srai></template>
</category>

<!-- XPROMPT - prompts the user to give some information -->
<category>
  <pattern>XPROMPT</pattern>
  <template>
    <random>
      <li>How are you feeling?</li>
      <li>What are you up to right now?</li>
    </random>
  </template>
</category>
```

The goal of the Personal bot is to get some information from the user so that it can make a recommendation about what they should listen to. You'll notice that the first two categories both end with `<srai>XPROMPT</srai>` , which is a design pattern we use to try and direct the conversation. You can use this all over your AIML codebase to "tack on" the bot's question to the end of an output.

recommend.aiml

The personal bot also features a file recommend.aiml, which makes a recommendation to the user based on their favorite artist.

```
<category>
  <pattern>WHAT SHOULD I LISTEN TO</pattern>
  <template>
    <condition name="favoriteartist">
      <li value="unknown">Tell me one of your favorite artists.</li>
      <li><srai>XREC</srai></li>
    </condition>
  </template>
</category>
```

The bot relies on a predicate `favoriteartist` to make the recommendation. In the first category, the `<condition>` block runs a check on the existence of this predicate - if doesn't exist, the bot will ask the user for that info. If it does exist, then the bot can move on the the `XREC` category to make the recommendation.

```
<category>
  <pattern>XREC</pattern>
  <template>
    If you like <formal><get name="favoriteartist"/></formal>, you might also like to listen to
    <sraix><bot><bot name="musiccat"/></bot>XSIMILAR <get name="favoriteartist"/></sraix>.
    Want to hear a track?
  </template>
</category>
```

It is at this point that the personal bot reaches out to another bot in its network. In this case, it is sending the user's `favoriteartist` to the comparison engine bot, and inserting that bot's response into its own. From the user's perspective, there is only one bot, but in reality the two work together to form the network's response.

```
User: What should I listen to?
Bot: Tell me one of your favorite artists.
User: Ed Sheeran.
Bot: If you like Ed Sheeran, you might also Sam Smith. Want to hear a track?
```

mood.aiml

The categories in this file are designed to give the user a recommendation based on their current mood.

```
 <category>
  <pattern>HAPPY</pattern>
  <that>HOW ARE YOU FEELING</that>
  <template>Glad to hear it! You should listen to some <sraix bot="djf/musiccat">XRANDOM POP BLOCK</sraix> to
</category>
```

All of these categories follow a similar model where the pattern is a different mood, and the response calls out to the comparison engine bot to get a random artist from a specific genre. If you remember our XPROMPT category, it will either return "What are you up to right now" or "How are you feeling?" - we use the `<that>` tag to ensure this category will only match if the latter was the last response given by the bot.

activity.aiml

This file is similar to the mood component, except it attempts to make a recommendation based on the user's current activity. This could be working, exercising, sleeping, etc. Depending on what the user is doing, the personal bot will again reach out to the comparison engine to get a random artist from a specified genre:

```
<category>
  <pattern>WORKING</pattern>
  <that>WHAT ARE YOU UP TO RIGHT NOW</that>
  <template>I find <sraix bot="djf/musiccat">XRANDOM CLASSICAL BLOCK</sraix> to be great background music when
</category>
```

artists.aiml

The categories in this file are designed for interfacing with the artist expert.

```
<category>
    <pattern>TELL ME ABOUT *</pattern>
    <template><srai>XBIO <star/></srai></template>
  </category>

<category>
  <pattern>XBIO *</pattern>
  <template>
    <think><set var="artist"><map name="artists"><star/></map></set></think>
    <condition var="artist">
      <li value="unknown">I don't know much about that artist.</li>
      <li><sraix><bot><bot name="app-id"/>/<map name="artists"><star/></map></bot>BIO</sraix> Want to hear a <s
    </condition>
  </template>
</category>

<category>
  <pattern>YES</pattern>
  <that>WANT TO HEAR A * SONG</that>
  <template>
    <sraix><bot><bot name="app-id"/>/<map name="artists"><thatstar/></map></bot>REC</sraix> - <formal><thatstar
  </template>
</category>
```

Take a close look at the `<think>` tag in XBIO *:

```
<think><set var="artist"><map name="artists"><star/></map></set></think>
```

This block of AIML sets a local variable whose value comes from the artists.map file, which is a sort of registry for all the artists expert bots in the network. This registry associates the artist's name with the name of the artist expert bot.

```
[
  ["Drake", "djf/drakebot"]
]
```

Right now we only have one expert bot in the network, but the idea is that the bot might know about some artists, and might not know about others. In the case that the bot does not have access to an expert bot, the first `<li>` will be returned:

```
User: Tell me about Taylor Swift
Bot: I don't know much about that artist.
```

But if we have registered the artist bot in artists.map, then the second `<li>` will be returned and the bot will contact the artist bot for an answer.

```
User: Tell me about Drake
Bot: Aubrey Drake Graham (born October 24, 1986) is a Canadian rapper ...
```

Final notes

Hopefully you've already gotten your hands on the code. You can deploy this network yourself, just make sure to follow the directions in the Readme file to properly route your `<sraix>` calls!