Dialogue Framework

*Developer Document*

*System Design, Configuration Format and User Cases*

Version: v1.0

Last revised: 2014-12-11

Table of Contents

[1. System Overview 3](#_Toc414435862)

[1.1 Introduction 3](#_Toc414435863)

[1.2 Contact Information 3](#_Toc414435864)

[2. System Architecture 3](#_Toc414435865)

[2.1 System Workflow 4](#_Toc414435866)

[2.2 Bot Life Cycle 5](#_Toc414435867)

[2.3 Play and Akka 5](#_Toc414435868)

[3. Configurations 6](#_Toc414435869)

[3.1 Properties 6](#_Toc414435870)

[3.2 Grammar Files 6](#_Toc414435871)

[3.3 Dialogue Configuration (AIML Files) 6](#_Toc414435872)

[3.4 An Example 6](#_Toc414435873)

[4. File Structure 7](#_Toc414435874)

[5. Data Storage 7](#_Toc414435875)

[5.1 Session Storage 8](#_Toc414435876)

[5.2 Profile Storage 8](#_Toc414435877)

[6. Deployment 8](#_Toc414435878)

[6.1 Prerequisites 8](#_Toc414435879)

[6.2 Installation 9](#_Toc414435880)

[7. Expandability 10](#_Toc414435881)

[7.1 Parser 10](#_Toc414435882)

[7.2 Evaluator 10](#_Toc414435883)

[8. Confidentiality and Access 10](#_Toc414435884)

1. System Overview
   1. Introduction

This document is written for system developers who are going to implement the dialogue framework. Also, it is useful for application developers who will use the dialogue framework to develop specific dialogue applications. The RSVP Dialogue Framework depends on a grammar compiler/parser for understanding natural language input. It is designed to enable application developers easily build up different dialogue systems with specific interactive settings. The implementation details should be included in another functionality specification document (FSD).

* 1. Contact Information

In case you encounter issues, please contact:

**RSVP Technologies**

[kunxiong@internetanswer.com](mailto:kunxiong@internetanswer.com)

1. System Architecture

Bot Farm

Developers

Users

Chat

Dialogue Framework

Provide

Grammar

Compiler

Dialogue

Config

Grammar

File

Profile

Storage

Session

Storage

Dialogue framework acts as a central hub between application developers and end users. It wraps underlying low level implementation and provides application developers APIs to only focus on designing chat bots’ behavioral logic. By submitting dialogue configurations and grammar definitions, application developers can specify how their chat bots accept user input and generate responding answer. Moreover, the dialogue framework provides APIs to empower application developers to build more complex logic such as randomization and recursion. And it communicates with session storage server and profile storage server under the hood so that application developers can also access users’ native attributes, dialogue history, and sessional information.

At run time, dialogue framework’s built in grammar compiler will compile provided grammar files and initialize chatterbot instances with dialogue configurations. It hosts a bot farm system built on top of Akka’s Actors and can not only keep multiple independent chat bots concurrently to server users with different needs, but also boost performance and throughput. Meanwhile, dialogue framework will route requests from users using different applications to appropriate chat bots accordingly.

* 1. System Workflow

Based on dialogue configurations, to provide a dialogue service, the dialogue framework coordinates with session storage, profile storage, and grammar parser for each user request to generate corresponding response. Its workflow is shown as the following.

Grammar Parser

Dialogue Framework

query

Request

Parse

query, session id, user id

parsed query

Session/Profile Server

User

session id, user id

Fetch

Chat Bot Instance

user session, profile

Execute Logic

Respond

updated user session, profile

answer

Update

status

When end user submits a query, the dialogue framework will first call the grammar parser to process it and the result can then be referred to later. The chat bot instance handling given request will try to match parsed query and direct it to specific category accordingly. During logic execution, it can then access the session/profile storage at request and perform procedures defined in dialogue configuration to generate answer to send back to user. When the request/response cycle completes, dialogue framework will update session/profile storage to record necessary information and keep track of user session and history.

* 1. Bot Life Cycle

In our system, a bot class extends Akka Actor. And the bot farm class extends Akka Round-Robin Router. As handling of each user request takes roughly the same amount of computing power, round-robin routing is a simple but good enough fit. The number of instances for each bot can be specified inside the configuration file. Thanks to Akka’s supervision and recycling system, we only need to implement the logic to generate a response after each user request. And Akka will take care of exception handling and restarting each bot (actor) instance if it died unexpectedly.

When dialogue framework starts, it will load all bots defined under the bot folder, each folder must contain a valid bot file structure where the name of the folder is the name of the bot. The bot actor farm will then instantiate specified number of instances for each bot and load all properties and aiml categories into memory so aiml pattern matching and template evaluation will all be in-memory process afterwards. Then the life cycle of each bot instance is taken care by Akka’s Actor system, unless a specific ‘reload’ command is issued towards a bot then it will reset all of its instances and reload all of its properties and aiml categories.

* 1. Play and Akka

The dialogue framework is built on top of state of the art frameworks on the JVM - Play and Akka. Play is a web application framework follows the MVC architectural pattern and entirely written with asynchronous paradigm. We use play as the host of the dialogue framework and implement all our RESTful APIs for remote access. It also acts as the hub to access our data storage and logging facility. Play is entirely written with asynchronous programing style and we try to implement everything in the dialogue framework as asynchronously as possible to guarantee best performance.

On the other hand, Akka is a toolkit and runtime to construct concurrent and distributed applications. Akka is the core of our bot farm and each bot instance running within the platform is an Akka actor to implement actor-based concurrency. It not only simplify the development and maintenance of our system but also boost its robustness by taking care of supervision and exception handling gracefully. Moreover, as Akka actors can be deployed remotely, it makes the entire system more expandable thus we are able to run bot instances in distributed fashion for better scalability and performance.

1. Configurations

To define a dialogue system (chatterbot)’s behavior, an application developer need to provide two types of configuration files.

* 1. Properties

Each bot has its own internal properties, such as its name. Properties are stored as a text file under each bot’s folder and defined as key value pairs and will be loaded when a bot instance starts running. Properties can be accessed in aiml templates when bot responses are generated.

* 1. Grammar Files

Grammar files are used to determine what input can be accepted. Detailed explanation on how grammar files work can be found in our grammar file specifications document.

* 1. Dialogue Configuration (AIML Files)

Dialogue configurations are written in xml format based on the Artificial Intelligence Markup Language (AIML). We extends AIML by allowing using grammar files to accept user input instead of its native pattern matching mechanism. We support majority of tags defined in AIML standard and also add a few of our custom tags. Detailed description and explanation of our customized AIML specifications can be found in its dedicated document.

* 1. An Example

Here is a simple example demonstrates how a grammar file and aiml file can work side by side to generate response to a user’s request.

namespace basic

hello := strings{Hello,Hi,Hey}

public greeting : hello

basic.gram

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

<aiml>

<category>

<pattern>

<grammar>

basic.greeting

</grammar>

</pattern>

<template>

Hello, how are you?

</template>

</category>

</aiml>

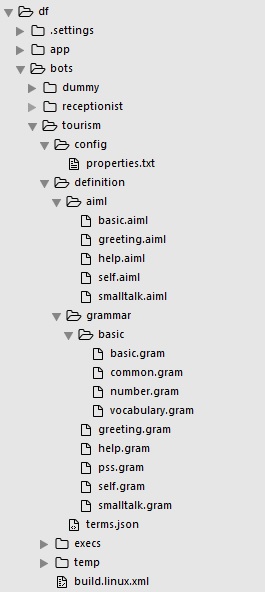
basic.aiml

The grammar file will catch ‘Hello’, ‘Hi’, ‘Hey’ and produce the public term ‘basic.greeting’. In the aiml file, it will look for user inputs that matches grammar term ‘basic.greeting’, and generate response based on its content inside template node. In this case, it will reply back ‘Hello, how are you’ when user says ‘Hello’, ‘Hi’, or ‘Hey’.

Of course, our customized aiml format supports way more than just plain text, and please refer to our aiml specs for further reading on how templates are evaluated.

1. File Structure

Each bot and its associated files must be placed under the bots folder. The typical file structure is shown in the following diagram.

In this example, we have three separate bots ‘dummy’, ‘receptionist’ and ‘tourism’, all of a bot’s definition and configuration files are under its named folder. ‘temp’ and ‘execs’ are placeholders which are only used for grammar compilation, and ‘build.linux.xml’ is a predefined ant build file. Under the config folder, there is the ‘properties.txt’ file defining key value pairs that will be loaded into each bot instance for future access in aiml template evaluation. Persistent properties of a bot should be defined here such as the bot’s name, version, etc.

All files that define the conversational logic of the bot reside in the definition subfolder. The ‘terms.json’ is an automatically generated file by the grammar compiler and should not be changed by dialogue developers. All grammar files are inside the grammar folder while all aiml files are inside the aiml folder. We support nested folders for these two types of files so one can create subfolders to better categorize ones definition files.

To add a new bot, we only need to create a new folder under bots and with this predefined structure. We also have an online editor for easy editing of the grammar and aiml files.

1. Data Storage

To store persistent data and enable context-based features, the dialogue framework’s functionality depends on both a persistent data storage and a session data storage. The persistent data storage will take care of long lasting user information (called profile in our system) such as a user’s name, gender, and properties alike. On the other hand, the session storage keep track of the conversation so we can always refer to any of the previous requests and responses the bot received and replied back.

* 1. Session Storage

The dialogue framework uses Redis as its session storage, the connection parameters to the Redis instance can be configured in ‘application.conf’ under the ‘conf’ folder. We use Play Framework’s Redis plugin and Jedis to talk to Redis in our system.

One built in feature of our session storage is that it automatically keeps track of the conversation between user and bot. Whenever a request is received and a response is generated, the dialogue is stored in our session storage. And such information can be accessed via some helper methods.

The session exposes to the bot instance a map of key value pairs, which can be set and get by using tags inside an aiml category. A session is associated with a certain user and identified by its own session id. A session will be released once it has passed its expiration time (no conversation between user and bot for a certain amount of time).

In our dialogue framework, everything is interfaced and injected with Google Juice for better separation of concerns. If you want a different implementation for session storage, you can implement your own class that extends ‘SessionStorage’ and configure it for Juice in Play Framework’s ‘Global.java’ file.

* 1. Profile Storage

The persistent data for user profiles are stored in a MySQL database in our system. Like session storage, it is interfaced and injected, too. A user profile is identified by user id, and contains a map of key value pairs just like the session storage. Similarly, these pairs can be accessed by using tags in the aiml categories.

Whenever a user send a request in, the request contains both its associated user id and session id. If these ids are skipped then random ones will be created instead, thus no profile or session information will be persisted across requests/responses.

The MySQL database do not have to be installed locally, its location can be configured in the ‘application.conf’ file, too. As we use the ORM tool Ebean for communication with the MySQL database, if you want custom table names or field names, feel free to do so, but you have to update the model classed defined inside package platform.models.

1. Deployment
   1. Prerequisites

First of all, Java must be present, and version 1.7.\* is recommended.

As the dialogue framework is built on top of Play Framework, one must have Play installed beforehand and run the application via the activator command line tool.

Moreover, the dialogue framework depends on Redis and MySQL by default for session and profile storage support. Please have both database installed and configured, either locally or remotely.

* 1. Installation

It is recommended to install dialogue framework on a Linux box. Although it can work on a Windows machine, certain modification has to be made to achieve so. Here we assume you are installing dialogue framework on a clean state (Linux) box. If you already have some of tools required installed, please skip steps accordingly.

1. Install Java 1.7. The Oracle Java is recommended (we haven’t tested anything with OpenJDK). And we only did testing on 1.7 too so we cannot guarantee everything would work the same on Java 1.8. Java can be downloaded from https://www.java.com
2. Install Redis. By default, the Redis plugin assumes Redis is installed locally without password. If your Redis instance is somewhere else or with a different setting, you can change the config in ‘conf/application.conf’ by specifying ‘redis.host’, ‘redis.port’, ‘redis.password’ and ‘redis.database’. More information about the Redis plugin can be found at <https://github.com/typesafehub/play-plugins/tree/master/redis>
3. Install MySQL. The dialogue framework uses jdbc driver to connect to MySQL, the location of the MySQL server can be configured in ‘conf/application.conf’ by setting ‘db.default.\*’ accordingly. We are using Ebean as the ORM tool and do no perform evolution. So the tables must be setup manually in your designated MySQL schema. To link table names and field names accordingly, you should change the annotation in ‘platform.models’ package. If you want to use the default setup, we have an ‘install.sql’ file in the root folder of the dialogue framework which you can execute within your designated schema to create tables that will work with our default model classes. MySQL can be downloaded from http://www.mysql.com
4. Install Play Framework. Play’s latest package can be downloaded from its website at <https://www.playframework.com/>. You need to put the folder in your executable path and run activator from command line to download everything required.
5. Run dialogue framework from command line. Head into the root folder of dialogue framework and use activator to run the Play application. E.g. ‘activator run’. It will execute the sbt build file and load up all dependencies for you and start the application afterwards. Dialogue framework is a standalone system and do not need to be run under any external web server applications. Additionally, Play supports continuous compilation and will automatically compile source files whenever you make a change on the go. So there is no need to restart the application when you are making changes.
6. Expandability

Many parts of the dialogue framework can be expanded easily. One can add a new class that implements a certain interface to empower the framework to do more things. And two of the most customizable and expandable parts are the parser and the evaluator. The parser allows you to match a certain pattern in a user’s request while the evaluator generates the response from the bot based on your predefined logic in an aiml category.

* 1. Parser

In our aiml files, pattern nodes are used to match users’ inputs. Inside our input parser, we have a bunch of predefined matchers (inside the core.bot.ab.matchers package) to perform the matching tasks. In general, if we are only using grammar files, then the ‘WordMatcher’ is the only one will be called in the matching process which performs exact text matching of the combination of grammar, that (previous bot response), and topic. The priority of all the matchers is defined in ‘DefaultMatcher’ and most of the other matchers are used when no grammar pattern is present thus everything falls back to default pattern matching for a standard ALICE bot. If you want to add your custom matchers, you only need to extend the abstract ‘PatternMatcher’ and add your new matchers into ‘DefaultMatcher’s match chain.

* 1. Evaluator

Evaluator is used to evaluate the template node from an aiml category once it matches a user input. It is used to generate the response which the user eventually receives. In dialogue framework, the evaluator has a collection of tag handlers that each handler takes care of one xml tag and the evaluation is done recursively. For instance, the ‘RandomHandler’ handles the ‘<random>’ tag that return one of the items listed inside the tag randomly. If you want to add a new custom tag, you only need to add a new handler that extends the abstract ‘TagHandler’, implement the ‘handle’ method and set up the ‘TagHandlerCollection’ accordingly. All tag handlers and are residing in the ‘core.bot.ab.handlers’ package. More details about the tag handlers can be found in the documentation of our aiml specs.

1. Confidentiality and Access

The readers should be aware that you must not disclosure this document itself or any information related with it to others, and that any load test or malicious access is not allowed.