Eva Documentation

Release 0.1

Edouard Poitras

February 09, 2017

Contents

Eva is an open source alternative to Amazon Echo or Google Home. The core of Eva is a simple service that provides hooks for plugins during interactions with users. On it's own, Eva does very little. It's potential comes from the plugins the user chooses to enable (or create).

CHAPTER 1

Installation

If you wish to use docker-compose to run Eva, ensure you have installed Docker and the docker-compose utility.

If you wish to run Eva outside of Docker, install the required Python dependencies:

```
pip3 install -r requirements.txt
```

Todo

Allow for a proper pip install eva command.

CHAPTER 2

First Steps

To run Eva with docker-compose, you simply have to run the following command:

```
docker-compose up
```

This may take a while on the first run as the Eva container is built and all the dependencies are installed.

To run Eva outside of Docker, you simply have to execute the `eva.directory.serve()` function. This is exactly what the `serve.py` script does for you:

```
python3 serve.py
```

The default setting for Eva is to install the Web UI Plugins and Web UI Updater plugins (and their dependencies) on first startup. This behaviour can be changed by modifying the core Eva *configuration* file (typically found at `~/eva/eva.conf`).

Alternatively, once Eva is started, you may navigate to the Web UI (https://localhost:8080) and enable/disable plugins as needed.

Note: A self-signed certificate warning from your browser is normal at this point.

While you're in the Web UI, you may as well download and enable the Web UI Interact plugin which will allow you to test out Eva from the Web UI.

We discuss Eva clients more in depth on the *Clients* page.

Eva's capabilities are entirely controlled by the plugins enabled by the user. Try enabling the echo plugin and see if Eva echos back the commands you send it. Also try the weather plugin and ask Eva for the current forecast (you'll need to setup DarkSky API keys).

Note: You will need to have enabled a voice recognition plugin if you wish to speak to Eva instead of using text commands.

Note: You will need to enable a Text-to-Speech plugin if you wish to receive spoken words from Eva as a response.

2.1 Clients

Eva uses a client-server model for it's distributed application structure. Generally speaking, an effort is being made to keep the clients as simple as possible in order to increase portability. This is the reasoning behind the Audio Server plugin - the clients only need to send audio data to Eva and let the plugins take care of voice recognition.

2.1.1 Local CLI

Primary used for development and testing purposes.

The Local CLI is a text-only stand-alone client that does not require the Eva server to be running in the background. Eva will be bootstrapped in the process of running the Local CLI.

Ensure you have all the required dependencies installed and a local instance of MongoDB running. You can also use the MongoDB instance available through the docker-compose configuration provided:

```
pip3 install -r requirements.txt --user
docker-compose up mongo
python3 clients/local_cli.py
```

2.1.2 Remote CLI

This client is similar to the Local CLI except that it does not bootstrap Eva and so requires a running Eva server to connect to. The only dependency is the anypubsub Python module.

The default settings assumes that the Eva MongoDB instance can be accessed locally:

```
pip3 install anypubsub --user
python3 clients/remote_cli.py
```

If this is not the case, the following code can be used to connect to the remotely accessible MongoDB instance that Eva is using:

```
from clients.remote_cli import RemoteCLI
cli = RemoteCLI(host='remote.host', port=27017, username='', password='')
cli.start_consumer('eva_messages')
cli.start_consumer('eva_responses')
cli.interact()
```

2.1.3 Headless (Experimental)

A simple voice-enabled client that has no user interface. It currently supports keyword-based activation through either pocketsphinx or snowboy models.

> **Warning:** This client is experimental and is currently under active development.

Requirements

- Requires a working pyaudio installation (with portaudio)

```
apt-get install portaudio19-dev
```

```
apt-get install python-pyaudio
```

Or

```
pip3 install pyaudio --user
```

- Requires pocketsphinx, webrtcvad, respeaker

```
apt-get install pocketsphinx
```

```
pip3 install pocketsphinx webrtcvad
```

```
pip3 install git+https://github.com/respeaker/respeaker_python_library.git
```

- May also need PyUSB

```
pip3 install pyusb
```

- Requires pydub for converting mp3 and ogg to wav for playback

```
pip3 install pydub
```

See https://github.com/jiaaro/pydub for system dependencies.

```
apt-get install ffmpeg libavcodec-ffmpeg-extra56
```

Or

```
brew install ffmpeg --with-libvorbis --with-ffplay --with-theora
```

- Requires anypubsub

```
pip3 install anypubsub --user
```

- Requires pymongo

```
apt-get install python3-pymongo
```

Or

```
pip3 install pymongo --user
```

- Requires that Eva have the Audio Server plugin enabled

Optional

You may optionally use snowboy for keyword detection. To do so, you need to get the `_snowboydetect.so` binary for your platform (the one found at `clients/snowboy/_snowboydetect.so` in this repo is only for Python3 on Ubuntu).

You can get precompiled binaries and information on how to compile here.

If you end up compiling, ensure you use swig >= 3.0.10 and use your platform's Python3 command in the Makefile (default is just `python`).

Once you've compiled snowboy (or downloaded the dependencies), put the `_snowboydetect.so` and `snowboydetect.py` files in the `clients/snowboy/` folder.

You can either get a keyword detection model on the snowboy website or use the provided alexa one in this repository.

Usage

`python3 clients/headless.py`

Or with a snowboy model:

`python3 clients/headless.py --snowboy-model=clients/snowboy/alexa.umdl`

2.1.4 Desktop (Incomplete)

A desktop client with a proper UI and taskbar icon is currently in the works. The progress can be followed in the dev/desktop_client branch (help appreciated).

2.1.5 Developers

The main way to communication with Eva is through the *communications* collection of Eva's main MongoDB instance.

There are three types of *channels* in the collection:

`eva_commands`

This channel is used to send commands or queries to Eva. Eva will continuously listen on that channel for queries/commands from clients.

An entry looks something like this:

```
{
"message" : {
"input_text" : "What is the current wind speed and humidity?",
"input_audio" : {
        "audio" : BinData(0, <data-here>),
        "content_type" : "audio/mpeg"
}
},
"type" : "message",
"channel" : "eva_commands",
"when" : ISODate("2017-01-25T03:00:00.000Z")
}
```

The 'input_audio' key is not needed if 'input_text' is provided.

`eva_responses`

This is the channel the clients should be listening on. All responses from Eva will be inserted into the MongoDB *communications* collection on this channel.

An entry looks something like this:

```
{
        "message" : {
                "output_text" : "The current wind speed is 6.3 kilometers per hour. The current humi
                "output_audio" : {
                        "audio" : BinData(0, <data-here>),
                        "content_type" : "audio/mpeg"
                }
        },
        "type" : "message",
        "channel" : "eva_responses",
```

```
        "when" : ISODate("2017-01-25T03:00:01.000Z")
}
```

`eva_messages`

This channel is used by Eva for notifications and to broadcast messages to all clients.

An entry looks something like this:

```
{
        "channel" : "eva_messages",
        "when" : ISODate("2017-01-25T03:00:05.000Z"),
        "message" : "There is a severe thunderstorm warning in effect in your area",
        "type" : "message"
}
```

In Python, the simplest way to send messages to Eva is to use the anypubsub Python module:

```
from pymongo import MongoClient
from anypubsub import create_pubsub_from_settings
client = MongoClient(URI_OF_EVA_DB)
pubsub = create_pubsub_from_settings({'backend': 'mongodb', 'client': client, 'database': 'eva', 'co
pubsub.publish('eva_commands', {'input_text': 'command or query here'})
```

You can also use the anypubsub module to receive responses or notification/messages from Eva:

```
from pymongo import MongoClient
from anypubsub import create_pubsub_from_settings
import time
client = MongoClient(URI_OF_EVA_DB)
pubsub = create_pubsub_from_settings({'backend': 'mongodb', 'client': client, 'database': 'eva', 'co

subscriber = pubsub.subscribe('eva_responses')
# Subscriber will continuously tail the mongodb collection channel.
for message in subscriber:
                if message is not None:
                                print(message['output_text'])
                time.sleep(0.1)
```

You would typically have a couple consumers (one for `eva_responses` and one for `eva_messages`) running in a separate thread. See clients/remote_cli.py for a working example.

Don't forget to check out clients/headless.py for a working example with audio and keyword activation.

2.2 Configuration

Eva is very flexible and easy to configure.

If you have Web UI Plugins enabled, you can configure each individual plugin from the Web UI instead of managing config files on disk.

2.2.1 Core

The core of Eva can be configured through the `eva.conf` file.

You can find this configuration file in any of the following locations:

- ~/eva.conf

- ~/.eva.conf
- ~/eva/eva.conf
- /etc/eva.conf
- /etc/eva/eva.conf

If you can't find the configuration file, Eva is most likely using all default options for your installation. You can simply create the file in any of the locations mentioned above and Eva will pick up your settings on next restart.

Here is the contents of the Eva specification file (`eva/eva.conf.spec`) which outlines the different configuration options available:

```
[eva]
# The git-accessible repo that holds all the available Eva plugins for download.
plugin_repository = string(default='https://github.com/edouardpoitras/eva-plugin-repository.git')

# The local path where the plugin_repository will be stored on disk.
plugin_repo_path = string(default='/tmp/eva-plugin-repository')

# The local directory holding all existing (and downloaded) plugins.
plugin_directory = string(default='~/eva/plugins')

# The local directory holding all plugin configurations.
config_directory = string(default='~/eva/configs')

# The list of enabled plugins - dependencies will be handled by Eva on boot.
enabled_plugins = force_list(default=list('web_ui_plugins', 'web_ui_updater'))

[logging]
# The namespace used for logging.
log_name = string(default='eva')

# The logging level.
log_level = option('CRITICAL', 'ERROR', 'WARNING', 'INFO', 'DEBUG', default='INFO')

[mongodb]
# The MongoDB username.
username = string(default='')

# The MongoDB password.
password = string(default='')

# The MongoDB host where Eva's database will be held.
host = string(default='localhost')

# The MongoDB port used to access Eva's database.
port = integer(default=27017)

# The MongoDB database name for Eva.
database = string(default='eva')
```

Note: See the ConfigObj documentation for more details on the specification file syntax.

An example configuration file could look like this:

```
[eva]
plugin_directory = /etc/eva/plugins
```

```
config_directory = /etc/eva/configs
enabled_plugins = web_ui_plugins, web_ui_updater, weather

[logging]
log_level = DEBUG

[mongodb]
host = my.eva.com
username = myusername
password = mypassword
database = my_eva
```

2.2.2 Plugins

All Eva plugins have the option of supplying a specification file to allow users to configure different behaviour for their installation.

Note: The plugin specification file must be named after the plugin name. If a plugin is named `my_plugin`, the specification file must be in the root of the plugin directory and named `my_plugin.conf.spec`.

When a plugin is enabled, Eva will scan the `config_directory` for a matching config file for that plugin. If one is found, the configuration values are validated through the plugin's specification file, loaded, and made available to all the plugins. The values can be accessed through the `conf` singleton.

For example, the Weather plugin has the following entries in it's specification file (`weather.conf.spec`):

```
darksky_api_key = string(default='')
location = string(default='')
latitude = float(-90.0, 90.0)
longitude = float(-180.0, 180.0)
metric = boolean(default=True)
```

Note: See the ConfigObj documentation for more details on the specification file syntax.

This means that if the weather plugin is enabled, all plugins (include weather) can access those configuration options like so:

```
from eva import conf
location = conf['plugins']['weather']['config']['location']
```

Not all that exciting as the location is set to an empty string by default. However, if a file named `weather.conf` is in the Eva config directory (default is `~/eva/configs`), Eva will pull in those values when loading the plugin:

```
# Making sure the weather plugin knows where I am.
location = 'Ottawa, Ontario, Canada'
```

Now the location variable from above will contain the value 'Ottawa, Ontario, Canada'.

2.3 Plugin Development

A typical Eva plugin will consist of a folder with the following structure:

- *plugin_name/*
- *plugin_name/plugin_name*.py
- *plugin_name/plugin_name*.info (optional)
- *plugin_name/plugin_name*.conf.spec (optional)
- *plugin_name/*requirements.txt (optional)

2.3.1 Folder Structure

The plugin folder must be located in the *configured* `plugin_directory` for Eva to pick it up as an available plugin (unless it's already in the public plugin repository).

Warning: The name of the info file, python file, and spec file must match the folder name in order to be picked up by Eva.

The specification file and the requirements.txt file are optional.

2.3.2 Python File

This is where your plugin code resides. You can do anything you want with your plugin, but it's most likely a good idea to make use of some of the helper functions and triggers that Eva exposes to the plugins.

See the *API Reference* documentation for more details.

Triggers

Plugins must register with some triggers in order to be notified of interactions with the user. Here is a simple example of a weather plugin that registers to the `eva.interaction` trigger in order to handle weather queries from clients:

```
import gossip

@gossip.register('eva.interaction')
def interaction(context)
    # Ensure no other plugin has responded to this query yet.
    # Ensure the query contains the word 'weather'.
    if not context.response_ready() and context.contains('weather'):
        weather = get_current_weather()
        # Respond with the weather information.
        context.set_output_text('Here is the current weather: %s' %weather)
```

For more details and a list of available triggers, see the *Triggers* section of this documentation.

Configuration

All plugins have the option to load a `conf` singleton dictionary that holds all the configuration and plugin information that Eva sees. It is also the primary way of accessing the custom configuration that a user may have specified for your plugin:

```
from eva import conf
# You can access all sorts of information on plugins.
location_value = conf['plugins']['weather']['config']['location']
info_file_object = conf['plugins']['weather']['info']
```

```
path_on_disk = conf['plugins']['weather']['path']
is_git_repo = conf['plugins']['weather']['git']
# You can access values from the Eva core configuration file.
plugin_path = conf['eva']['plugin_directory']
config_path = conf['eva']['config_directory']
```

See *Configuration* page for more info on creating your own specification file and allowing users to provide custom configuration for your plugin.

Scheduler

The `scheduler` singleton is an instance of an APScheduler BackgroundScheduler.

It should be used for any long-running or periodic jobs that your plugin need to initiate. The documentation for creating jobs can be found here.

Here is an examples that fires a new job in the background immediately:

```
from eva import scheduler
scheduler.add_job(func_name, id="eva_my_plugin_job")

def func_name():
    # Job stuff here.
    pass
```

The function provided needs to exist and the job ID needs to be unique.

Here is an example using the decorator syntax that fires a job every hour:

```
from eva import scheduler
from eva import log

@scheduler.scheduled_job('interval', hours=1, id='eva_my_plugin_hourly_job')
def hourly_job():
    log.info('Running this job on the hour again!');
```

Here is an example of running a job with parameters on a specific date:

```
from eva import scheduler
from eva import log
scheduler.add_job(birth_day, 'date', run_date=date(2017, 02, 10), args=['Happy Birthday!'])

def birth_day(message):
    log.info(message)
```

Publish

All plugins can import the `publish` function which will allow plugins to easily broadcast messages to all Eva clients:

```
from eva import publish
publish('This is a message to all!')
```

`publish` can take a second parameter, which is the channel to publish the message on. This value is 'eva_messages' by default as that's the channel that Eva plugins should be listening on.

Todo

Does not yet support publishing audio to clients.

Logger

The `log` singleton makes for easy logging:

```
from eva import log
log.debug('This is a debug message')
log.info('This is an info message')
log.warning('This is a warning message')
log.error('This is an error message')
log.critical('This is a critical message')
```

2.3.3 Info File

The plugin info files are pretty simple.

Here is the specification file used to load Eva plugins:

```
# Every Eva plugin should have a name matching it's python module name.
name = string(default='Plugin Unknown')
# Description of this Eva plugin.
description = string(default='No description')
# The current version of the Eva plugin.
version = string(default='0.0.0')
# List of Eva plugin dependencies for this plugin.
dependencies = force_list(default=list())
# Use the requirements.txt for Python module dependencies.
```

As you can see, all fields have a default value, and so it is not necessary to have an info file.

Here is an example plugin info file taken from the Weather plugin:

```
name = Weather
description = Enables the response of weather-related queries from Eva.
version = 0.1.0
dependencies = conversations
```

Warning: The *dependencies* field refers to Eva plugin dependencies, not python module dependencies. Use a *requirements.txt* in your plugin folder to specify python module dependencies.

2.3.4 Specification File

A plugin specification file can be provided if you wish to give the user a way of configuring different aspects of your plugin.

If a specification file is available, Eva will use it to validate a configuration file that the user may have provided in the `config_directory` (see *Core* configuration for more details).

The Weather plugin is a good example. It requires that the user provide an API key in order to access weather information.

See *Plugins* configuration for more details and an example.

2.3.5 requirements.txt File

Every plugin can provide a *requirements.txt* in order to specify python module requirements.

Eva will automatically installs the python modules from this file when the plugin is enabled.

2.3.6 Full Example

We're going to build a simple plugin named `motivate` that has the goal of motivating the user.

Our plugin should be able to send encouraging responses to the user when asked, and send follow-up motivational comments if the user claims it didn't work the first time.

It will also send random encouraging statements to the user every day.

Let's start with our info file (`motivate/motivate.info`):

```
name = Motivate
description = Motivate the user with this amazing plugin!
version = 0.1.0
dependencies = conversations
```

We're adding the conversations plugin as a dependency because we want to be able to handle follow-up query/commands, which is something the conversations plugin offers through it's `eva.conversations.follow_up` trigger.

Let's allow the plugin to capture the user's name so as to make the motivations more personal.

Here our configuration specification file (`motivate/motivate.conf.spec`):

```
user_name = string(default='User')
```

We won't be using any python modules other than the ones required by Eva, so no requirements.txt file is needed.

Now for our actual plugin code (`motivate/motivate.py`):

```
import random
import gossip
from eva import conf
from eva import publish
from eva import scheduler

# User name pulled from the configuration.
USER = conf['plugins']['motivate']['config']['user_name']

# We could also pull motivational phrases from the internet.
# We could also make the motivational phrases configurable in the spec file.
PHRASES = ['Never give up %s!' %USER,
           'You can do it %s!' %USER,
           '%s, you don\'t have to have it all figured out to move forward.' %USER,
           '%s, keep your eyes on the stars, and your feet on the ground.' %USER]

def get_phrase(ask_follow_up=True):
    # Choose a random motivational phrase.
    phrase = random.choice(PHRASES)
    if ask_follow_up:
        # Don't forget to ask if they are sufficiently motivated.
        return '%s Are you sufficiently motivated?' %phrase
    return phrase
```

```
@gossip.register('eva.interaction')
def interaction(context):
    # Ensure no other plugin has already responded and the user's query or
    # command contains the word 'motivate' (as in 'motivate me please').
    if not context.response_ready() and context.contains('motivate'):
        # Get are motivational phrase.
        response = get_phrase()
        # Apply the response so that Eva knows to send it to the client.
        context.set_output_text(response)

@gossip.register('eva.conversations.follow_up')
def follow_up(plugin_id, context):
    # Check if we should be handling the follow-up query/command.
    if plugin_id == 'motivate':
        # If the user's query/command contains the word 'no', we try again.
        if context.contains('no'):
            # Get another motivational phrase.
            response = get_phrase()
            context.set_output_text(response)
        else:
            # Tell other plugins that this interaction has been taken care of.
            context.responded = True
            # Explicitly close the conversation (don't wait for timeout).
            context.conversation.close()

@scheduler.scheduled_job('interval', hours=24, id='eva_motivate_job')
def motivate_job():
    # We don't want to ask the user for a follow-up here.
    phrase = get_phrase(False)
    # Publish the motivational message to clients.
    publish(phrase)
```

There we have it. Sending commands to Eva from the Web UI Interact page or one of the provided *Clients*, you get something like the following:

```
You > Hello Eva, can you motivate me please?
Eva > You can do it User! Are you sufficiently motivated?
You > Um... no.
Eva > User, keep your eyes on the stars, and your feet on the ground. Are you sufficiently motivated?
You > Yeah actually that worked. Thanks.
```

You could now setup your own name by creating a `motivate.conf` file in Eva's *configuration directory* with the following content:

```
user_name = Eddie
```

Next steps would be to add your plugin to a GitHub repository and submit it to the public plugin repository so everyone can be motivated!

2.4 Triggers

Triggers are the way that Eva and plugins can interact with each other and share control of the program flow. Eva uses the gossip python module for this purpose.

To register a function in your plugin to a specific trigger, you simply need to decorate the function like so:

```
import gossip

@gossip.register('trigger_name_here')
def custom_function():
    # Perform actions here when the 'trigger_name_here' trigger is fired.
    pass
```

If the trigger provides variables, your function needs to have those parameters as well:

```
import gossip
from eva import log

@gossip.register('test_trigger')
def custom_function(value):
    log.info('You fired the trigger with the value: %s' %value)

value = 'Hello World!'
gossip.trigger('test_trigger', value=value)
```

It is a good idea to create triggers throughout your plugin to allow other plugins to modify data, or simply be notified of certain events. Ensure you document your triggers so other plugin developers can take advantage of them.

There are many ways of managing trigger priorities and dependencies. See the gossip documentation for more details.

It is not possible to `return` data back to the caller from a registered trigger. The triggering plugin must provide a referenced object (list, dict, object) as a parameter to the trigger in order to receive feedback from the other plugins:

```
import gossip

@gossip.register('test_trigger')
def test(data):
    data.append('One')
    data.append('Two')

data = ['Testing']
gossip.trigger('test_trigger', data=data)
print(', '.join(data))
# Testing, One, Two
```

Note: There are many triggers that are exposed by plugins. If your plugin overlaps in functionality with another, or if you want to integrate your plugin with another, it is certainly worth looking at the other plugin documentation to see if a trigger already exists to fullfill the requirement. If not, I'm sure a pull request would be welcome :)

You can register functions in your plugin with any of the following triggers:

2.4.1 eva.pre_boot

A trigger that gets fired before Eva starts loading plugins. This is not accessible by Eva plugins.

2.4.2 eva.plugins_loaded

A trigger that gets fired immediately after all Eva plugins have been loaded.

2.4.3 eva.post_boot

This trigger is fired once Eva has booted, but before Eva has begun to listen for commands.

2.4.4 eva.voice_recognition

A trigger that gets fired when a new interaction has begun, but only `input_audio` was provided (no `input_text`). This is primarily used by plugins that transcribe audio.

param data The data received from the clients on query/command. See *eva.context.EvaContext.__init__()* for more details.

type data dict

2.4.5 eva.pre_interaction_context

A trigger that gets fired when a new interaction is about to begin. No *eva.context.EvaContext* object is available at this point.

param data The data received from the clients on query/command. See *eva.context.EvaContext.__init__()* for more details.

type data dict

2.4.6 eva.pre_interaction

Same as the `eva.pre_interaction_context` trigger except that the context object has been created and is passed to the registered function.

param context The context object created for this interaction.

type context *eva.context.EvaContext*

2.4.7 eva.interaction

This trigger is where most plugin check if they should be handling the input from the user.

Usually plugins will check if another plugin has not already acted on the user's query/command before acting:

```
@gossip.register('eva.interaction')
def interaction(context)
    if not context.response_ready():
        context.set_output_text('Too late other plugins, I'm responding!')
```

You would typically want to use the context object's *eva.context.EvaContext.contains()* method to see if certain text was part of the query/command from the user:

```
@gossip.register('eva.interaction')
def interaction(context)
    if not context.response_ready() and context.contains('weather'):
        weather = get_current_weather()
        context.set_output_text('Here is the current weather: %s' %weather)
```

param context The context object created for this interaction.

type context *eva.context.EvaContext*

Todo

Need to mention other plugins that offer more powerful tools like follow-up questions and intent parsing.

2.4.8 eva.post_interaction

Triggered immediately after `eva.interaction`.

param context The context object created for this interaction.

type context *eva.context.EvaContext*

2.4.9 eva.text_to_speech

This trigger is called when the interaction is complete and no output_audio is present in the context object. This is primarily used by plugins to convert text to audio data for the clients to play as a response from Eva.

You would usually use the *eva.context.EvaContext.set_output_audio()* if you wanted to add output_audio to the interaction.

param context The context object created for this interaction.

type context *eva.context.EvaContext*

2.4.10 eva.pre_return_data

This is triggered right before returning the response data to the clients. It gives plugins the opportunity to alter the raw response from Eva.

param return_data Same as what is returned from the *eva.director.interact()* function.

type return_data dict

2.4.11 eva.scheduler.job_failed

This is triggered when an APScheduler job sends the `EVENT_JOB_ERROR` event. See APScheduler events documentation for more details.

param event The APScheduler event returned from the failed job.

type event apscheduler.events.JobEvent

2.4.12 eva.scheduler.job_succeeded

This is triggered when an APScheduler job sends the `EVENT_JOB_EXECUTED` event. See APScheduler events documentation for more details.

param event The APScheduler event returned from the successful job.

type event apscheduler.events.JobEvent

2.4.13 eva.logger.debug

A trigger that gets fired every time a debug message is logged.

param message The message that is being logged.

type message string

2.4.14 eva.logger.info

A trigger that gets fired every time a info message is logged.

param message The message that is being logged.

type message string

2.4.15 eva.logger.warning

A trigger that gets fired every time a warning message is logged.

param message The message that is being logged.

type message string

2.4.16 eva.logger.error

A trigger that gets fired every time a error message is logged.

param message The message that is being logged.

type message string

2.4.17 eva.logger.critical

A trigger that gets fired every time a critical message is logged.

param message The message that is being logged.

type message string

2.4.18 eva.pre_publish

A trigger that is fired when a message is getting ready for publishing.

param message The message that will be published.

type message string

2.4.19 eva.publish

A trigger that is fired right before a message will be published to clients.

param message The message that will be published.

type message string

2.4.20 eva.post_publish

A trigger that is fired immediately after a message is published to clients.

param message The message that will be published.

type message string

2.4.21 eva.pre_set_input_text

A trigger that gets fired right before setting a new `input_text` value for the current interaction.

param text The new text that is being set as `input_text`.

type text string

param plugin_id The plugin ID that is setting this new `input_text`.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.4.22 eva.post_set_input_text

A trigger that gets fired right after setting a new `input_text` value for the current interaction.

param text The new text that was set as `input_text`.

type text string

param plugin_id The plugin ID that has set this new `input_text`.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.4.23 eva.pre_set_input_audio

A trigger that gets fired right before setting a new `input_audio` value for the current interaction.

param audio The new audio data that is being set.

type audio binary string

param content_type The content type of this audio data.

type content_type string

param plugin_id The plugin ID that is setting this new audio data.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.4.24 eva.post_set_input_audio

A trigger that gets fired right after setting a new `input_audio` value for the current interaction.

param audio The new audio data that was set.

type audio binary string

param content_type The content type of this audio data.

type content_type string

param plugin_id The plugin ID that has set this new audio data.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.4.25 eva.pre_set_output_text

A trigger that gets fired right before setting a new `output_text` value for the current interaction.

param text The new text that is being set as `output_text`.

type text string

param responding True if this new `output_text` is responding to this interaction's query/command. False if simply altering the response. This flag is used in *eva.context.EvaContext.response_ready()* to determine if a response has already been formulated for a query/command.

type responding boolean

param plugin_id The plugin ID that is setting this new `output_text`.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.4.26 eva.post_set_output_text

A trigger that gets fired right after setting a new `output_text` value for the current interaction.

param text The new text that was set as `output_text`.

type text string

param responding True if this new `output_text` was responding to this interaction's query/command. False if simply altering the response. This flag is used in *eva.context.EvaContext.response_ready()* to determine if a response has already been formulated for a query/command.

type responding boolean

param plugin_id The plugin ID that was setting this new `output_text`.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.4.27 eva.pre_set_output_audio

A trigger that gets fired right before setting a new `output_audio` value for the current interaction.

param audio The new audio data that is being set.

type audio binary string

param content_type The content type of this audio data.

type content_type string

param plugin_id The plugin ID that is setting this new audio data.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.4.28 eva.post_set_output_audio

A trigger that gets fired right after setting a new `output_audio` value for the current interaction.

param audio The new audio data that is being set.

type audio binary string

param content_type The content type of this audio data.

type content_type string

param plugin_id The plugin ID that is setting this new audio data.

type plugin_id string

param context The context object for this interaction.

type context *eva.context.EvaContext*

2.5 API Reference

2.5.1 Config

Holds functions related to Eva and plugins configuration.

`eva.config.`**`get_config`**(*config_file=None*, *spec_file=None*, ***kwargs*)

Function used to fetch Eva core and plugin configurations on startup. If config_file and spec_file are `None`, assumes the we're looking for Eva's config and spec file.

> **Warning:** This function should not be used directly unless you know what you are doing. Use the singleton conf variable to access Eva and plugin configurations:

```
from eva import conf
plugin_directory = conf['eva']['plugin_directory']
fake_plugin_variable = conf['plugins']['fake_plugin']['config']['variable_name']
```

Parameters

- **config_file** (*string*) – The location of the configuration file to parse.
- **spec_file** (*string*) – The location of the configuration specification file.

Returns The loaded configuration object.

Return type ConfigObj

eva.config.**get_config_spec**(*spec_file=None*)
Returns a configuration specification based on the spec file provided. Assumes the <eva_directory>/eva.conf.spec file if no spec file path specified.

Parameters **spec_file** (*string*) – The path of the specification file to load.

Returns The specification file object.

Return type

ConfigObj

eva.config.**get_eva_config_file**()
Function that attempts to determine where Eva's main configuration file resides.

Looks for the following files:

•~/eva.conf

•~/.eva.conf

•~/eva/eva.conf

•/etc/eva.conf

•/etc/eva/eva.conf

Returns Eva's configuration file, if found.

Return type string

eva.config.**get_eva_directory**()
Function used to get the directory of the current file. Effectively determining Eva's source code directory.

Returns Eva's source code directory.

Return type string

eva.config.**get_plugin_config**(*plugin_id*, *config_dir*)
Wrapper around *get_config()* to fetch a plugin's configurations.

> **Warning:** The same *get_config()* warning applies for this function. Stick with the conf singleton in order to retrieve plugin configurations.

Parameters

- **plugin_id** (*string*) – The plugin ID.
- **config_dir** (*string*) – The directory where the plugin configuration file is found.

Returns The loaded configuration object.

Return type

ConfigObj

eva.config.**save_config**(*plugin_id=None*, *section=None*)
Save current active plugin configuration to disk.

If `plugin_id` is not provided, then `section_id` MUST be provided. That is because without `plugin_id`, Eva assumes you're trying to save it's core configuration - which requires a section to save.

If Eva can't find it's core configuration object when saving, it will write out it's current configurations to `~/eva/eva.conf`.

Parameters

- **plugin_id** (*string*) – The plugin ID to have it's configurations preserved. If `None`, assumes Eva's core configurations - in which case `section` is required.
- **section** (*string*) – If `plugin_id` is `None`, section must be provided to determine which section of the Eva configuration file should be saved.

2.5.2 Context

Holds the EvaContext class - an integral part of every Eva interaction.

class eva.context.**EvaContext**(*data=None*)
An EvaContext object is passed along to plugins (via gossip triggers) during an Eva interaction with the user. The object contains all the information required for the plugin to determine whether or not it should act on the current interaction.

Plugin developers should always interact with Eva (and back out to the user) through the context object. This is important as the context object fires various triggers that enable other plugins to hook into ongoing interactions.

__init__(*data=None*)
The data attribute is typically a dict with the following structure:

```
dict {
    'input_text': The text/query provided by the client
    'input_audio': dict {
        'audio': The binary audio data of the query (optional)
        'content_type': The content type of the audio binary data (optional)
    }
    'output_text': The text of the response from Eva
    'output_audio': dict {
        'audio': The binary audio data of the response(optional)
        'content_type': The content type of the audio binary data (optional)
    }
}
```

It may also contain `output_text` and `output_audio` with the same format as it's `input` counterpart, but that is unlikely on initial creation of the context object (unless a plugin is doing something unorthodox on a trigger that's fired before interaction begins).

Input in this case refers to text or audio that a client has sent Eva. Output refers to the resulting response from Eva that gets sent back.

Parameters **data** (*dict*) – The data received from an Eva client.

contains(*keyword*)
Simple helper method to determine if a keyword appears in a client's input text (query or command).

Parameters **keyword** – The keyword to check for in the input text.

Returns True if the keyword is found, False otherwise.

Return type boolean

get_input_audio()
Method used to get the input audio data that was sent by the client for this interaction.

Returns The audio binary data received from an Eva client this interaction.

Return type binary string

get_input_audio_content_type()
Method that returns the content type of the audio binary data received during this interaction.

Typically something like 'audio/mpeg' or 'audio/wave'.

Returns The content type of the audio binary data received this interaction.

Return type string

get_input_text()
Method used by plugins to get the input text (query or command) from the Eva client for this interaction.

Returns The input text from the Eva client this interaction.

Return type string

get_output_audio()
Method that returns the resulting output audio binary data that the Eva client will play to the user.

Returns The output audio binary data that Eva will send back to the client.

Return type binary string

get_output_audio_content_type()
Method that returns the output audio content type that will be sent back to the Eva client.

Returns The content type of the audio binary data returned to the Eva client.

Return type string

get_output_text()
Method used by plugins to get the output text (response) that has been generated so far by plugins in this interaction.

The string returned from this method may not always end up being the string returned to the Eva clients. A plugin may end up modifying the output text at any point in the interaction (even right before sending the response to the client).

Returns The output text for the Eva client so far in this interaction.

Return type string

input_audio = None
The input audio binary data from an Eva client.

input_audio_content_type = None
The content type of the input audio binary data.

input_text = None
The input text (query or command) from an Eva client.

output_audio = None
The output audio binary data from Eva.

output_audio_content_type = None
The content type of the output audio binary data.

output_text = None
The output text (response) from Eva.

responded = None
True if a plugin has already handled the response, False otherwise.

response_ready()
Method used by plugins to determine whether or not they should take part in this current interaction.

A response being ready means that another plugin has already set some output text that should be sent back to the Eva client.

Returns True if a response has already been generated, False otherwise.

Return type boolean

set_input_audio(*audio*, *content_type*)
Same as *set_input_text()* except it works for the input audio and content type. Not very many plugins will end up using this as it involves modifying the audio query or command that was sent by the Eva client.

This method fires the `eva.pre_set_input_audio` and `eva.post_set_input_audio` triggers.

Parameters

- **audio** (*binary string*) – The audio to be set as the input audio for this interaction.
- **content_type** (*string*) – The content type of this binary audio data.

set_input_text(*text*)
Method used to set the input text of the current interaction.

This function is primarily used by voice recognition plugins to convert input audio into input text when clients don't provide any.

This method fires the `eva.pre_set_input_text` and `eva.post_set_input_text` triggers.

Parameters **text** (*string*) – The text that will now become the input text from the client.

set_output_audio(*audio*, *content_type*)
Similar to *set_output_text()* except for the audio data and content type. This method will be used primarily by the text-to-speech plugins.

Parameters

- **audio** (*binary string*) – The audio binary data to send back to the client.
- **content_type** (*string*) – The content type of the binary audio data.

set_output_text(*text*, *responding=True*)
The bread and butter method of the context object.

This method is used by nearly every plugin as it's used to tell Eva what the response to the client should be.

This method fires the `eva.pre_set_output_text` and `eva.post_set_output_text` triggers.

Parameters

- **text** (*string*) – The text to be set as output for the client.
- **responding** (*boolean*) – `True` if the text provided is a response to the client. `False` if you're simply modifying the output text without claiming to be the primary plugin to respond to the query or command.

Leaving this as `True` means other plugins will be able to tell that the client's query has already been answered (by checking the boolean variable returned by *`response_ready()`*).

If a plugin is simply prepending/appending text or making slight modifications to the output, it should use `responding=False` so as to allow follow-up questions to be routed to the appropriate plugin.

2.5.3 Director

This is where the magic begins. The director has functions to fire up Eva, load all the plugins, and begin interactions with the clients.

`eva.director.`**`boot`**`()`
The function that runs the Eva boot sequence and loads all the plugins.

Fires the *eva.pre_boot* and *eva.post_boot* triggers.

`eva.director.`**`get_return_data`**`(`*context*`)`
This function is used to extract appropriate data from the context object before sending it to the Eva clients.

It will check the context object for a text response and an audio response, and return a dict containing this information.

Parameters **`context`** (*`eva.context.EvaContext`*) – The context object used for this interaction.

Returns A dict that may contain the key *output_text*, *output_audio*, or both. It should be identical to the return value of the *`interact()`* function barring any changes during the *eva.pre_return_data* trigger.

Return type dict

`eva.director.`**`handle_data_from_client`**`(`*pubsub*, *data*`)`
Helper function to fire off an interaction with Eva based on client data received, and then send off the response back to the clients.

Parameters

- **`pubsub`** (anypubsub.interfaces.PubSub) – The pubsub object used to publish Eva messages to the clients.
- **`data`** (*`dict`*) – The data received from Eva clients. See *`eva.context.EvaContext.__init__()`* for more details.

`eva.director.`**`interact`**`(`*data*`)`
Eva's bread and butter function. Feeding data from the clients directly to this function will return a response dict, ready to be consumed by the clients as a response. This takes care of firing all the necessary triggers so that the plugins get a say in the responding text and/or audio.

Fires the following triggers:

- *eva.voice_recognition*
- *eva.pre_interaction_context*
- *eva.pre_interaction*
- *eva.interaction*
- *eva.post_interaction*
- *eva.text_to_speech*

- *eva.pre_return_data*

Parameters **data** (*dict*) – The data received from the clients on query/command. See *eva.context.EvaContext.__init__()* for more details.

Returns

A dictionary with all the information necessary for the clients to handle the response appropriately. Typically something like this:

```
dict {
    'output_text': The text of the response from Eva
    'output_audio': dict {
        'audio': The binary audio data of the response (optional)
        'content_type': The content type of the audio binary data (optional)
    }
}
```

Return type dict

eva.director.**serve**()

This is the one function you need to execute to start Eva.

It begins the boot sequence, loads up all plugins, and starts listening for client interactions.

2.5.4 Logger

Contains all of Eva's logging facilities.

class eva.logger.**Logger**

The Logger class is a very light wrapper around Python's standard logging framework. It's primary purpose is to wrap every logging message into a method that fires triggers on messages. This allows for plugins to act on certain log messages.

It should not be necessary to instantiate this class manually as this is already done in the __init__.py file. Use *from eva import log* to use a singleton instance of this class.

__init__()

Initializes the standard Python logging class at the appropriate level set in the Eva configuration file. Will also specify the appropriate logging format.

critical(*message*)

Simple wrapper around the standard Python logger's critical method. Fires the *eva.logger.critical* trigger.

debug(*message*)

Simple wrapper around the standard Python logger's debug method. Fires the *eva.logger.debug* trigger.

error(*message*)

Simple wrapper around the standard Python logger's error method. Fires the *eva.logger.error* trigger.

info(*message*)

Simple wrapper around the standard Python logger's info method. Fires the *eva.logger.info* trigger.

warning(*message*)

Simple wrapper around the standard Python logger's warning method. Fires the *eva.logger.warning* trigger.

2.5.5 Plugin

All necessary helper functions to facilitate plugin management.

eva.plugin.**download_plugin**(*plugin_id*, *destination*)
Will download the specified plugin to the specified destination if it is found in the plugin repository.

Parameters

- **plugin_id** (*string*) – The plugin ID to download.
- **destination** (*string*) – The destination to download the plugin on disk.

eva.plugin.**enable_plugin**(*plugin_id*, *downloadable_plugins=None*)
Enables a single plugin, which entails:

•If already enabled, return

•If plugin not found, search online repository

•Download if found in repository, else log and return

•Recusively enable dependencies if found, else log error and return

•Run a `pip install -r requirements.txt --user` if requirements file found

•Insert plugin directory in Python path and dynamically import module

•Execute the `<plugin>.on_enable()` function if found

Todo Need to clean up, comment, and shorten this function.

Parameters

- **plugin_id** (*string*) – The plugin id to enable.
- **downloadable_plugins** (*dict*) – A dict of plugins that are available for download from Eva's repository. This is typically the return value of the *get_downloadable_plugins()* function.

eva.plugin.**enable_plugins**()
Function that enables all plugins specified in Eva configuration file. Will enable all available plugins if none is specified in the configs.

eva.plugin.**get_downloadable_plugins**(*pull_latest=False*)
Gets a dict of downloadable plugins from the plugin repository. The plugin repository is simply a git repo with a list of available plugins stored in a CSV file. The repository will be cloned locally if not found.

Parameters **pull_latest** (*boolean*) – Whether or not to perform a `git pull` on the repository before parsing the plugins.

Returns A dict of all available plugins for download. The format is:

```
{
    <plugin_id> {
        'id': <id>,
        'name': <name>,
        'description': <description>,
        'url': <url>
    }
    ...
}
```

Return type dict

eva.plugin.**get_plugin_directory**()
Helper function to get Eva's plugin directory specified in the config file.

Will automatically replace ~ with the user's home directory.

eva.plugin.**load_plugin_configs**(*config_dir*)
Function that loops through all available plugins and loads their corresponding plugin configuration if found in the configuration directory provided.

The *load_plugin_directory()* function must be called before calling this function as it relies on the plugin info files having been loaded into the `conf['plugins']` dict.

Parameters **config_dir** (*string*) – The configuration directory that holds all Eva plugin configuration files.

eva.plugin.**load_plugin_directory**(*plugin_dir*)
Will crawl Eva's plugin directory and load the plugin info files for all the valid plugins found. The info file information will eventually be used when enabling plugins and their dependencies.

This function does not return anything. It stores all plugin information in the `conf['plugins']` dict. Every plugin should have the following accessible information once this function is run:

```
conf['plugins'][<plugin_id>] = {
    'info': 'Data from the info file for this plugin'
    'path': 'The path of the plugin on disk'
    'git': 'True if the plugin is a git repo (and can be updated)'
}
```

Use the following statement to access the conf dict: `from eva import conf`

Parameters **plugin_dir** (*string*) – The directory containing available Eva plugins. Typically the return value of *get_plugin_directory()*.

eva.plugin.**load_plugin_info**(*plugin_path*, *plugin_id*)
Given a plugin path and plugin name, this function will attempt to return a loaded plugin info file as a ConfigObj specification instance.

Parameters

- **plugin_path** (*string*) – The path of the plugin in question.
- **plugin_id** (*string*) – The plugin ID.

Returns A ConfigObj specification instance.

Return type

ConfigObj

eva.plugin.**load_plugins**()
The function that is called during Eva's boot sequence. Will fetch the plugin directory , load all of the plugins' info files, load all of the plugin's configurations, and enable all the required plugins and their dependencies specified in Eva's configuration file.

Fires the *eva.plugins_loaded* trigger.

eva.plugin.**num_available_plugins**()
Function used to get the number of available plugins. Will simply check the length of the `conf['plugins']` dict.

Returns The number of available plugins.

Return type integer

eva.plugin.**num_enabled_plugins**()
Function used to determine the number of enabled plugins. Simply checks if the `module` key exists in the `conf['plugins'][<plugin>]` dict.

Returns The number of enabled plugins.

Return type integer

eva.plugin.**plugin_enabled**(*plugin_id*)
Function used to determine whether a plugin is enabled or not. Simply looks in the `conf['plugins']` dict to see if the imported module is present.

Parameters **plugin_id** (*string*) – The plugin name.

Returns True if the plugin seems to be enabled, False otherwise.

Return type boolean

eva.plugin.**plugin_is_git_repo**(*plugin_path*)
Will attempt to determine if the plugin_path specified is a git repo. Simply checks if the `.git` folder exists.

Todo There's probably a better way of doing this.

Parameters **plugin_path** (*string*) – The path of the plugin to check.

eva.plugin.**pull_repo**(*repo_path*)
Helper function to perform the equivalent of a `git pull origin master` on a specified git repository on disk.

Parameters **repo_path** (*string*) – The path of the git repository to pull.

eva.plugin.**refresh_downloadable_plugins**()
Will remove the entire local directory holding the plugin repository and re-clone the repo locally. This can be useful when changing plugin repositories.

2.5.6 Scheduler

Holds functions required to start and manage the Eva scheduler.

eva.scheduler.**get_scheduler**()
Function used to return the APScheduler instance that is used by Eva and plugins.

Warning: This function should only be used by Eva. Plugins should access the scheduler through Eva's singleton object:

```
from eva import scheduler
# This will fire off the job immediately.
scheduler.add_job(func_name, id="eva_<plugin_id>_job")
```

Todo

Need to add listeners for all event types: https://apscheduler.readthedocs.io/en/latest/modules/events.html#event-codes

Note This function most likely needs to be revisited as it may not be thread-safe. Eva and plugins can modify the config singleon simultaneously inside and outside of jobs.

Returns The scheduler object used by plugins to schedule long-running jobs.

Return type apscheduler.schedulers.background.BackgroundScheduler

eva.scheduler.**job_failed**(*event*)
A callback function that gets called when an APScheduler job fails.

Currently will simply fire the *eva.scheduler.job_failed* trigger with the failed event object.

eva.scheduler.**job_succeeded**(*event*)
A callback function that gets called when an APScheduler job succeeds.

Currently will simply fire the *eva.scheduler.job_succeeded* trigger with the success event object.

2.5.7 Util

Any function that did not neatly fit in any other Eva python file.

eva.util.**get_calling_plugin**(*depth=2*)
This method will inspect the `depth` level of the call stack to find which python module is responsible for the current method invocation.

It is used when determining which plugin called the get/set output/input text/audio methods in the context object.

Parameters **depth** (*integer*) – How deep to look down the call stack (2 for calling function).

eva.util.**get_mongo_client**()
A helper function to get a MongoDB client object with the credentials, host, and port specified in the Eva configuration file.

Returns A MongoClient configured for a Eva MongoDB connection.

Return type pymongo.MongoClient

eva.util.**get_pubsub**()
Helper function to get the pubsub client used to send and receive messages.

Returns The pubsub object used to publish Eva messages to the clients.

Return type

anypubsub.interfaces.PubSub

eva.util.**publish**(*message*, *channel='eva_messages'*)
A helper function used to broadcast messages to all available Eva clients.

Todo Needs to be thoroughly tested (especially with audio data).

Parameters

- **message** (*string*) – The message to send to clients.
- **channel** (*string*) – The channel to publish in. The default channel that clients should be listening on is called 'eva_messages'.

eva.util.**restart**(*args=[]*)
Function used to restart Eva.

> **Warning:** This will restart Eva immediately and kill all running scheduler jobs.

Parameters **args** (*list*) – A list of arguments to feed Eva on restart.

CHAPTER 3

Index and Search

- genindex
- search

e

Symbols

B

C

D

E

G

H

I

J

L

N

O

P

R

S

W