Moe Serifu Agent Documentation

Release 0.1

Moe Serifu Circle & Contributors

Aug 11, 2021

Contents

CHAPTER 1

Contents

1.1 Moe Serifu Agent

1.1.1 Overview

Build Status

logo

Project Website Documentation

Moe Serifu Agent (MSA) is an event-driven personal assistant system that presents itself as existing in a particular location (like a house or a smartphone) and performs various tasks as directed by the user.

At a high-level, this system provides an anime-themed character that exists in cyberspace. It runs around the location it's installed in and appears at the end-users' beck and call in order to perform whatever services are needed, including timed reminders, checking and reporting on the state of its location, conversation, and performing in an entertainment role.

As an example, a user might tell the MSA to greet them when they return from work, or to wake them up in a customized way in the mornings. With its plugin API, new sensors and interfaces can be added to allow the MSA to interact with the world in just about any way the user desires.

1.1.2 Anime AI

The MSA project is inspired by various fictional artificial entities, such as the Virtual Intelligences from the Mass Effect Series, the Persocoms from the Chobits series, the Tachikoma from the Ghost in the Shell series, and the

AnthroPCs from the Questionable Content webcomic. The primary goal of the project is to create a system that carries out commands for the user and that gives the appearance of being an independent intelligent entity.

The anime theme was chosen because the author believes that the demograph that consumes anime tends to have a lower barrier to their willing suspension of disbelief in ascribing emotions to fictional characters than that of the general population.

mascot-vsign

1.1.3 Exchangable Personality and Appearance

The MSA system at its core represents itself as an anime-themed character. An intelligent agent system is used to determine how to accomplish goals set by the user, as well as to control the character's state, including the appearance of emotions and how to react to events. The AI is driven partially by a personality module, which can be exchanged in order to make the character act differently. Different personality modules are created with different behaviors in mind; each would fall under a different anime character archetype, such as tsundere, kuudere, yandere, deredere, etc.

An avatar of the character is presented to the end-user for interfacing with the system. This initial project narrows the goal of the avatar system to exist purely in cyberspace; there is no physical device (such as a robotic assembly) that the MSA can manipulate, although this functionality could certainly be added using the plugin system.

This MSA avatar can be interacted with using a variety of methods including voice recognition and via command-line interface, and it is shown to the user as a 3D model or 2D character on whichever devices are included in an instance of the system.

The specific details regarding what the avatar looks like visually, how it sounds, and how it demonstrates emotions are controlled by an avatar module within the MSA. This module can be exchanged with other such modules in order to change the appearance of the avatar.

A personality module and avatar module are intended to be combined into a set and distributed as a complete 'character pack', though there is nothing in the system design that would prevent the personality module of one pack from being used with the avatar module of another.

1.1.4 Physical Representation

In a complete MSA installation, a device (such as a screen/monitor) is set up in each of the rooms that it is to be interacted with. The MSA maintains a 'room' that the character resides in, and the character 'travels' between rooms by its avatar exiting a device and entering another one in an adjacent physical room. In general, the avatar will only travel between adjacent devices, e.g. if the system is set up such that device A is next to device B which is next to device C, then in order to travel from device A to device C, the avatar will move from A to B, then B to C.

Additionally, the user may download an app that allows their mobile device to be used as an output device. In this case, the avatar could travel directly to the user in order to interact with them. The MSA system would use a variety of sensors in order to detect the physical location of the mobile device and track which other output devices it should be considered adjacent to.

mascot-chibi

1.2 Installation

Note: Currently the only way to run MSA is from source see: Running From Source

1.2.1 Windows Installation

[STUB]

1.2.2 Linux/MacOS Installation

[STUB]

1.3 Getting Started

!!! Until our first official release, the only supported way of installing MSA is from source. Please see Running From Source.:w

The goal of this getting started guide is to walk you through the basic usage of MSA.

Note: This guide assumes that you have already installed the `moe-serifu-agent` package, if not, please see the Installation page.

Note: This guide will explain how to run MSA from the `moe-serifu-agent` package. To run from source see Running From Source

1.3.1 Up and running

Starting the daemon

Before we can start interacting with MSA we need to start it. To start MSA, run `moe-serifu-agent daemon` in a terminal. By default the daemon will run on port `8080`.

Connecting to the daemon with a client

The development Command Line Interface (also called the cli)

To start the cli, open a new terminal on the same computer running the daemon. Then run `moe-serifu-agent cli`. This should start the interactive cli. This tool is an interactive python interpreter for writing and uploading scripts. It is the lowest level way of interacting with MSA.

To exit at any time, press `Ctrl+c` or type `quit()`.

1.3.2 Next Steps

Now that we have covered how to get up and running with MSA, here are a few more topics worth reading:

- *Customizing your MSA*: Covers setting up a configuration file that you can use to customize the behavior of MSA.
- *Using the CLI*: A tutorial on using the cli to write scripts for MSA to run, and to build custom automations.
- *Extending MSA with plugins*: Adding additional functionality through MSA plugins.

1.4 Configuration File

The configuration file is the easiest way to begin configuring MSA to your liking. The configuration file os a JSON file. JSON stands for JavaScript Object Notation, and is a common way of storing structured data. As tutorials on how to write JSON are easily found, we will avoid going into specifics with how json works here. The most you need to know is that JSON is a series of key -> value associations.

This guide will refer to various nested configuration values in the config file, in order to easily reference a given JSON value we will use the following naming scheme: `agent.name` to refer to the `"Masa-chan"` value of `{"agent": { "name": "Masa-Chan" }}` easily.

1.4.1 Configuration Values

Agent

The agent section configures the behavior and appearance of the agent.

agent.name

The name MSA will refer to itself as.

Example:

```
{
  "agent": {
    "name": "Your humble servant"
  }
}
```

agent.user_title

The name MSA will refer to the user as.

Example:

```
{
  "agent": {
    "user_title": "Supreme Leader"
  }
}
```

Plugin Modules

The plugin modules section, allows a user to configure which third-party plugins to load when MSA starts. It should be a list of plugin modules to load at startup.

Example:

```
{
  "plugin_modules": [
    "my_demo_plugin"
  ]
}
```

Module Config

The module config section is a mapping of module name to JSON object. The JSON object is configuration values that will be passed to the module to modify its behavior.

Example:

```
{
  "module_config": {
    "my_demo_plugin": {
      "my_demo_message": "hello world"
    }
  }
}
```

Logging

The logging section, allows you to configure how MSA will record information about how well it is running, It will also record any errors that are encountered.

logging.global_log_level

Sets the global log level. Must be one of “error”, “warn”, “info”, or “debug”. The global log level defines how verbose all modules will be with their logging.

Example:

```
{
  "logging": {
    "global_log_level": "info"
  }
}
```

logging.log_file_location

The file that the logging output is written to. Example:

```
{
  "logging": {
    "log_file_location": "my_custom_file.log"
  }
}
```

logging.truncate_log_file

Toggles overwriting or truncating the log file when MSA starts up. If `false` log files will be preserved between runs. Example:

```
{
  "logging": {
    "truncate_log_file": false
  }
}
```

logging.granular_log_levels

A module to log level mapping that overrides the `logging.global_log_level` setting for that module. This can be used to increase logging or suppress a module that is logging too much unneeded information. Log level values must be one of “error”, “warn”, “info”, or “debug”.

Example:

```
{
  "logging": {
    "granular_log_levels": [
      { "namespace": "echo", "log_level": "debug"},
      { "namespace": "command_registry", "log_level": "error"}
    ]
  }
}
```

1.4.2 Example configuration

```
{
   "agent": {
       "name": "Masa-chan",
       "user_title": "Onee-chan"
   },
  "plugin_modules": [

  ],

  "module_config": {

  },

  "logging": {
    "global_log_level": "info",
    "log_file_location": "msa.log",
    "truncate_log_file": false,
    "granular_log_levels": [
      { "namespace": "echo", "log_level": "debug"},
      { "namespace": "command_registry", "log_level": "error"}
    ]
  }
}
```

1.5 Using the CLI

1.5.1 `cli` basics

When you start the cli, you will be greated with something like the following:

```
>>>
```

At it’s core the cli is stripped down python interactive interpreter. Similar to if you just ran `python` on your terminal. There are some notable differences:

- The presence of a `msa_api` object which is used for interacting with the daemon.
- Some "meta-commands" for the cli that allow special functionality within the cli. This includes recording scripts and saving them.

1.5.2 Intro to meta-commands

Meta-commands are executed within the cli and do not directly touch the daemon.

Available meta commands can be listed by typing `# help` in the cli e.g.:

```
>>> # help
MSA Interpreter Help:
 Availiable Commands:
  # help: Show this help text
  # record <file name>: Begin recording commands to a script.
  # record stop: stop recording commands, save the script, and open to review.
```

Note: the result of this `help` meta-command is subject to change as meta-commands are added or removed.

See below for an example using meta-commands to record a script interactively and sending it to the daemon to run periodically.

1.5.3 Intro to the `msa_api` object

When you start the cli a `msa_api` object is inserted into scope to facilitate interaction with the daemon process. The `msa_api` object can be used to interface with the daemon process.

Basic usage

Below you can find a list of methods available on `msa_api`.

The methods marked `async` below need to be prefixed with `await`. For example:

```
>>> await msa_api.talk("hello")
```

Recording and uploading scripts

One of the meta-commands available in the cli is the `# record` command.

As in the following example, it is possible to record an commands and save them as a script which can then be uploaded to the daemon.

```
$ moe-serifu-agent
>>> # record test.py
>>> await msa_api.talk("Hello, how are you?")
I am well thank you. What can I do for you?
>>> await msa_api.talk("Please turn on the livingroom light")
I am afraid I don't know what to say.
>>> # record stop
Opening test.py
```

Now if we `ls` the current directory, we should see a new file `test.py` has been created, containing the following:

```
await msa_api.talk("Hello, how are you?")
await msa_api.talk("Please turn on the livingroom light.")
```

We can also upload the script to the daemon and schedule it to run every so often.

```
>>> await msa_api.upload_script("demo_script", crontab="* * * * *", file_name="test.py
↪")
demo_script was sucessfully uploaded
```

The test script we uploaded should run once every minute. Unfortunately this is not very useful as we cannot see the conversation response from MSA because the daemon doesn't know where to send the response. Future tutorials will walk through some more interesting uses for uploaded scripts such as triggering events.

One final note on uploading scripts is that they are saved to MSA's database and are reloaded at startup.

1.5.4 `msa_api` Reference

class msa_api

check_connection(*self*)
Raises an exception if the cli cannot contact the daemon.

Async

Returns *None*

check_version(*self*, *quiet=False*)
Called automatically at startup, ensures that the cli version and the daemon versions match.

Async

Parameters **quiet** (*bool*) – Print's the response of *True*.

Returns *None*

get_version(*self*)
Fetches the daemon's version.

Async

Returns *None*

ping(*self*, *quiet=False*)
Send's a ping to the daemon. If the ping succeeds, you should see a pong response.

Async

Parameters **quiet** (*bool*) – Print's the response of *True*.

Returns *None*

1.6 Extending MSA with Plugins

[[STUB]]

1.7 Architectural Overview

Note: This section is very techy. If you are not interested or knowledgeable in programming or how the internals of the MSA work, this section is likely not for you.

[[STUB]]

1.8 Contributor Guide

1.8.1 Running from source

Installation

1. If you do not have it already, please install the python package manager `poetry`: See poetry installation instructions.
2. Clone the repository `git clone https://github.com/moe-serifu-circle/moe-serifu-agent.git`
3. Open a terminal and navigate to the location you cloned the repository to.
4. Run `poetry install` and `poetry shell` to install the python requirements and enter a virtual environment.

Starting the daemon

1. Open a terminal and navigate to the location you cloned the repository to.
2. Run `python -m msa daemon` to start the daemon. The daemon should now be hosted on http://localhost:8080.

Starting the Developer CLI

Note: In order to use the CLI you must start the daemon first in another terminal.

1. Open a terminal and navigate to the location you cloned the repository to.
2. Run `python -m msa cli` to start the cli which will connect to the daemon. You should be greeted with a python interpreter looking interface, this is the developer cli.

1.8.2 Plugin Development

[STUB]

1.9 Changelog

1.9.1 Version 1.0

Initial release.

1.10 msa package

1.10.1 Subpackages

msa.api package

Submodules

msa.api.api_clients module

msa.api.base_methods module

`msa.api.base_methods.`**`check_connection`**(*self*)

Raises an exception if the cli cannot contact the daemon.

Async

Returns *None*

`msa.api.base_methods.`**`check_version`**(*self*, *quiet=False*)

Called automatically at startup, ensures that the cli version and the daemon versions match.

Async

Parameters **`quiet`** (`bool`) – Print's the response of *True*.

Returns *None*

`msa.api.base_methods.`**`get_version`**(*self*)

Fetches the daemon's version.

Async

Returns *None*

`msa.api.base_methods.`**`ping`**(*self*, *quiet=False*)

Send's a ping to the daemon. If the ping succeeds, you should see a pong response.

Async

Parameters **`quiet`** (`bool`) – Print's the response of *True*.

Returns *None*

`msa.api.base_methods.`**`register_base_methods`**(*api_wrapper*)

msa.api.context module

`class` `msa.api.context.`**`ApiContext`**

Bases: `enum.Enum`

An enumeration.

`local = 0`

`rest = 1`

`websocket = 2`

msa.api.patchable_api module

class msa.api.patchable_api.**MsaApi**(**args*, ***kwargs*)

Bases: `dict`

msa.api.patcher module

class msa.api.patcher.**ApiPatcher**(*api_context*, *api_client*, *plugin_whitelist*)

Bases: `object`

cache = {}

static load(*api_context*, *api_client=None*, *plugin_whitelist=None*)

register_method()

Module contents

msa.builtins package

Subpackages

msa.builtins.conversation package

Submodules

msa.builtins.conversation.client_api module

msa.builtins.conversation.events module

msa.builtins.conversation.handlers module

msa.builtins.conversation.server_api module

Module contents

msa.builtins.intents package

Submodules

msa.builtins.intents.events module

msa.builtins.intents.handlers module

Module contents

msa.builtins.scripting package

Submodules

msa.builtins.scripting.client_api module

msa.builtins.scripting.entities module

msa.builtins.scripting.events module

msa.builtins.scripting.handlers module

msa.builtins.scripting.script_execution_manager module

msa.builtins.scripting.server_api module

Module contents

msa.builtins.signals package

Submodules

msa.builtins.signals.client_api module

msa.builtins.signals.events module

msa.builtins.signals.handlers module

msa.builtins.signals.server_api module

Module contents

Module contents

msa.cli package

Submodules

msa.cli.interpreter module

Module contents

msa.core package

Submodules

msa.core.config_manager module

msa.core.event module

msa.core.event_bus module

class msa.core.event_bus.**EventBus**(*loop*)

Bases: object

The event bus is responsible for tracking event queues and pushing new events into the event queues so that the event handlers can wait until a new event is sent to them via their event queue.

fire_event(*new_event*)

Fires an event to each event handler via its corresponding event queue.

Parameters **new_event** (*msa.core.event.Event*) – A subclass of msa.core.event.Event to propagate to event handlers.

listen(*timeout=None*)

Listens for a new event to be passed into the event bus queue via EventBus.fire_event.

listen_for_result(*event_type*, *timeout=None*)

subscribe(*event_type*, *callback*)

unsubscribe(*event_type*, *callback*)

msa.core.event_handler module

class msa.core.event_handler.**EventHandler**(*loop: asyncio.events.AbstractEventLoop*, *event_bus: msa.core.event_bus.EventBus*, *logger: logging.Logger*, *config: Optional[Dict[KT, VT]] = None*)

Bases: object

The base event handler class, all other event handlers should be a subclass of this type.

Variables

- **loop** (*asyncio.AbstractEventLoop*) – the main event loop.
- ***event_bus*** (msa.core.event_bus.EventBus) – an event loop that this handler may attempt to read events out of by awaiting on it.

init()

An optional initialization hook, may be used for executing setup code before all handlers have benn fully started.

schedule() → List[Tuple[str, Callable[[], Coroutine[datetime.datetime, None, None]]]]

An optional hook, may be used for scheduling one or more methods/functions to be periodically called..

The expected return value is a list of tuples. Each tuple should be a crontab string, followed by a coroutine that should be executed periodically based on the given crontab.

Invalid crontabs will result in a warning in the log, and the coroutine will not be scheduled.

Returns

Return type List[Tuple[str, Callable[[], Coroutine[None]]]]

msa.core.loader module

`msa.core.loader.`**`load_builtin_modules`**`()`
Loads builtin modules.

`msa.core.loader.`**`load_plugin_modules`**`(`*plugin_module_names*`)`
Loads plugin modules as specified in the configuration file.

Parameters **plugin_module_names** (*List[str]*) – Plugin module names to load. Module names should be fully qualified modules existing in *msa.plugins*.

msa.core.supervisor module

Module contents

`msa.core.`**`get_supervisor`**`()` → Supervisor

msa.data package

Module contents

msa.plugins package

Subpackages

msa.plugins.notifications package

Submodules

msa.plugins.notifications.events module

msa.plugins.notifications.handlers module

msa.plugins.notifications.notification_providers module

Module contents

msa.plugins.rss_feed package

Submodules

msa.plugins.rss_feed.events module

msa.plugins.rss_feed.handlers module

Module contents

Module contents

msa.server package

Submodules

msa.server.default_routes module

msa.server.event_propagate module

msa.server.route_adapter module

msa.server.server_request module

msa.server.server_response module

msa.server.url_param_parser module

Module contents

msa.tools package

Submodules

msa.tools.msa_rest_client_loader module

Module contents

msa.utils package

Submodules

msa.utils.asyncio_utils module

`msa.utils.asyncio_utils.`**`async_read`**`(`**args, **kwargs*`)`

`msa.utils.asyncio_utils.`**`run_async`**`(`*coroutine*`)`

`msa.utils.asyncio_utils.`**`sync_to_async`**`(`*func*`)`

Module contents

1.10.2 Submodules

1.10.3 msa.version module

1.10.4 Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index