Moe Serifu Agent Documentation

Release 0.1

Moe Serifu Circle & Contributors

Sep 28, 2020

Contents:

CHAPTER 1

Overview

1.1 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.

1.2 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

1.3 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.

1.4 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

CHAPTER 2

Installation

2.1 Windows Installation

[STUB]

2.2 Linux/MacOS Installation

[STUB]

CHAPTER 3

Getting Started

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

3.1 Up and running

To start MSA run, `moe-serifu-agent` in a terminal. You will be presented with a prompt. Interacting with the prompt is the most basic way to interact with MSA.

To find available commands type `help` e.g.

```
>> help
Available Commands:
echo: Echos provided text back through the terminal
quit: Shuts down the current Moe Serifu Agent instance
help: Prints available commands and information about command usage.
```

To read the help text for a specific command, type `help [name of a command]` e.g.

```
>> help echo
Help text for command 'echo':
Usage: 'echo [text]'
Options: No available options.
Description: Echos provided text back through the terminal
```

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

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.
- *Built-in Commands*: Describes each of the builtin commands and what you can do with them.
- *Extending MSA with plugins*: Adding additional functionality through MSA plugins.

CHAPTER 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.

4.1 Configuration Values

4.1.1 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"
  }
}
```

4.1.2 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"
  ]
}
```

4.1.3 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"
    }
  }
}
```

4.1.4 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"}
    ]
  }
}
```

4.2 Example configuration

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

  ],

  "module_config": {

  },
```

(continues on next page)

(continued from previous page)

```
  "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"}
    ]
  }
}
```

CHAPTER 5

Built-in Commands

5.1 Getting Help

At any point a user can enter `help` into the prompt to get a list of available commands. To view help text for a specific command type `help [command name]` where `[command name]` is the name of the command you wish to know more about.

5.2 Echo

The echo command causes the MSA to repeat back to you what you enter. For example, entering `echo hello world` will cause the MSA to say `hello world`.

5.3 Quit

Shuts down the MSA and exits.

CHAPTER 6

Extending MSA with Plugins

[[STUB]]

CHAPTER 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.

7.1 Built-in Modules

7.1.1 Command Registry

The Command Registry is the heart and soul of the command system. When a user enters text, and the TTY module propagates a `TextInputEvent`, the Command Registry attempts to parse the input into an invoke keyword and a list of parameters. If the first token in the input matches the invoke keyword of a registered command type, the Command Registry will propagate a new event for the registered command type to handle.

The Command Registry also handles listening and displaying text for help queries.

7.1.2 Command

7.1.3 Echo

7.1.4 Time

The time module propogates a `TimeEvent` at the beginning of every minute.

7.1.5 TTY

The TTY module enable input and output from the terminal. The TTY modules input handler listsens to the TTY for terminal input and generates a `TextInputEvent` for other modules to handle.

CHAPTER 8

Contributor Guide

8.1 Running from source

1. Clone the repository `git clone https://github.com/moe-serifu-circle/moe-serifu-agent.git`
2. Open a terminal and navigate to the location you cloned the repository to.
3. Run `pipenv install` and `pipenv shell` to install the python requirements and enter a virtual environment.
4. Run `python -m msa` to start the system. You should be greeted with the default prompt.

8.2 Plugin Development

[STUB]

CHAPTER 9

Changelog

9.1 Version 1.0

Initial release.

CHAPTER 10

API Reference

10.1 Subpackages

10.1.1 msa.builtins package

Subpackages

msa.builtins.command package

Submodules

msa.builtins.command.events module

msa.builtins.command.handlers module

Module contents

msa.builtins.command_registry package

Submodules

msa.builtins.command_registry.events module

msa.builtins.command_registry.handlers module

Module contents

msa.builtins.echo package

Submodules

msa.builtins.echo.events module

msa.builtins.echo.handlers module

Module contents

msa.builtins.time package

Submodules

msa.builtins.time.events module

msa.builtins.time.handlers module

Module contents

msa.builtins.tty package

Submodules

msa.builtins.tty.events module

msa.builtins.tty.handlers module

msa.builtins.tty.prompt module

msa.builtins.tty.style module

Module contents

Module contents

10.1.2 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.

create_event_queue()
Creates a new event queue. Each handler should receive its own 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.

msa.core.event_handler module

msa.core.loader module

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

msa.core.loader.**load_plugin_modules**(*plugin_module_names*, *mode*)
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*.
- **mode** (*msa.core.RunMode*) – The mode the system is being run in.

msa.core.supervisor module

Module contents

10.2 Submodules

10.3 msa.config module

class msa.config.**Config**(*config_file: str*, *sections: Dict[str, msa.config.Section]*)
Bases: `object`

A wrapper class for storing and accessing multiple sections

exception msa.config.**ConfigError**(*sec: str*, *key: str*, *val: str*, *msg: str*, *index: int = 0*)
Bases: `Exception`

A type of exception to be used when encountering invalid configuration settings

index() → int

key() → str

message() → str

section() → str

value() → str

class msa.config.**Section**(*name: str*)
Bases: object

Holds a group of keys, each key can have multiple or no values assigned

create_key(*key: str*) → None

get_all(*key: str*) → List[str]
Returns a list of all values within a given key

get_entries() → List[str]
Returns a list of all existing keys, even if the keys are empty

has(*key: str*) → bool

push(*key: str*, *val: str*) → None
Adds a value to the end of a key, even if there are empty values

set(*key: str*, *index: int*, *val: str*) → None

msa.config.**load**(*filepath: str*) → msa.config.Config
Loads a configuration file into a Config object for use within the code

msa.config.**save**(*config: msa.config.Config*, *filepath: str*) → None
Saves a Config object as a file to the provided file path

10.4 msa.var module

class msa.var.**Expander**
Bases: object

Holds variables for substitution in strings

expand(*text: str*) → str
Replaces any variables within a string to their values if any, variables are preceded by $

get_value(*var: str*) → str

register_protected(*var: str*, *val: str*) → None

register_var(*var: str*) → None

set_value(*var: str*, *val: str*) → None

unregister_protected(*var: str*) → None

unregister_var(*var: str*) → None

10.5 Module contents

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index