Jaffle Documentation

Release 0.2.1

Jaffle Development Team

May 21, 2018

User Documentation

Jaffle is an automation tool for Python software development, which has the following features:

- Instantiate Python applications in a Jupyter kernel and allows them to interact each other
- Launch external processes
- Combine all log messages and allows filtering and reformatting by regular expressions and functions
- Built-in *WatchdogApp* watches filesystem events and triggers another app, arbitrary code, and functions, which make it possible to setup various automations.

CHAPTER 1

Screenshot

Fig. 1: Developing a single-page web app using Tornado and React

Warning: Jaffle is intended to be a development tool and does not care much about security. Arbitrary Python code can be executed in `jaffle.hcl` and you should not use it as a part of production environment. `jaffle.hcl` is like a Makefile or a shell script included in a source code repository.

1.1 Installation

1.1.1 Prerequisite

- **UNIX-like OS**
  - Windows is not supported
- Python >= 3.4
- Jupyter Notebook >= 5.0
- Tornado >= 4.5, < 5

Jupyter Notebook and Tornado will be installed automatically if they do not exist in your environment. Tornado 5 is not yet supported.

1.1.2 Installation

```
$ pip install jaffle
```

You will also probably need pytest:

```
$ pip install pytest
```

1.2 Commands

Jaffle consists of the following commands:

1.2.1 jaffle start

Starts Jaffle.

Type `ctrl-c` to stop it.

Usage

```
jaffle start [options] [conf_file, ...]
```

The default value for `conf_file` is `"jaffle.hcl"`.

If multiple config files are provided, they will be merged into one configuration.

Options

- **–debug**

  Set log level to logging.DEBUG (maximize logging output)

- **-y**

  Answer yes to any questions instead of prompting.

- **–disable-color**

  Disable color output.

- **–log-level=<Enum>** (Application.log_level)

  Default: 30

  Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL') Set the log level by value or name.

- **–log-datefmt=<Unicode>** (Application.log_datefmt)

  Default: '%Y-%m-%d %H:%M:%S'

  The date format used by logging formatters for %(asctime)s

- **–log-format=<Unicode>** (Application.log_format)

  Default: '%(time_color)s%(asctime)s.%(msecs).03d%(time_color_end)s %(name_color)s%(name)14s%(name_color_end)s %(level_color)s %(levelname)1.1s %(level_color_end)s %(message)s'

  The Logging format template

- **–runtime-dir=<Unicode> (BaseJaffleCommand.runtime_dir)** Default: '.jaffle'

  Runtime directory path.

- **–variables=<List>** (JaffleStartCommand.variables)

  Default: []

  Value assignments to the *variables*.

Merging Multiple Configurations

If you provide multiple configuration files, Jaffle read the first file and then merges the rest one by one. Maps are merged deeply and other elements are overwritten.

Given that we have the following three configurations.

a.hcl:

```
process "server" {
  command = "start_server"
  env = {
    FOO = 1
  }
}
```

b.hcl:

```
process "server" {
  command = "start_server"
  env = {
    BAR = 2
  }
}
```

c.hcl:

```
process "server" {
  command = "start_server"
  env = {
    FOO = 4
    BAZ = 3
  }
}
```

```
```

When we start Jaffle by typing `jaffle start a.hcl b.hcl c.hcl`, the configuration will be as below:

```
process "server" {
  command = "start_server"
  env = {
    FOO = 4
    BAR = 2
    BAZ = 3
  }
}
```

Resolved variables are passed to the later configurations. Given that we have the following two configurations and use them as `jaffle start a.hcl b.hcl`.

a.hcl:

```
variable "server_command" {
  default = "start_server"
}

variable "disable_server" {
  default = false
}

process "server" {
  command  = "${var.server_command}"
  disabled = "${var.disable_server}"
}
```

b.hcl:

```
variable "disable_server" {
  default = true # switch the default value to true
}

process "server" {
  command  = "${var.server_command} --debug"
}
```

The configurations will be merged as follows:

```
variable "server_command" {
  default = "start_server"
}

variable "disable_server" {
  default = true
}

process "server" {
  command  = "${var.server_command} --debug"
  disabled = "${var.disable_server}"
}
```

Tip: The configuration merging is useful when you have a default configuration in your repository and you want to overwrite some part of it.

Example:

```
$ jaffle start jaffle.hcl debug.hcl log_filter.hcl
```

1.2.2 jaffle stop

Stops the running Jaffle process. If it is not running, removes runtime files if they exist.

Usage

```
jaffle stop [options]
```

Options

- **–runtime-dir=<Unicode>** (BaseJaffleCommand.runtime_dir)

  Default: '.jaffle'

  Runtime directory path.

1.2.3 jaffle console

Opens an interactive shell and attaches to the specified kernel instance.

Type `ctrl-c` or `ctrl-d` to stop it.

Usage

```
jaffle console <kernel_instance_name> [options]
```

The default value for `conf_file` is `"jaffle.hcl"`.

Options

- **–debug**

  Set log level to logging.DEBUG (maximize logging output)

- **-y**

  Answer yes to any questions instead of prompting.

- **–disable-color**

  Disable color output.

- **–log-level=<Enum>** (Application.log_level)

  Default: 30

  Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL') Set the log level by value or name.

- **–log-datefmt=<Unicode>** (Application.log_datefmt)

  Default: '%Y-%m-%d %H:%M:%S'

  The date format used by logging formatters for %(asctime)s

- **–log-format=<Unicode>** (Application.log_format)

  Default: '%(time_color)s%(asctime)s.%(msecs).03d%(time_color_end)s %(name_color)s%(name)14s%(name_color_end)s %(level_color)s %(levelname)1.1s %(level_color_end)s %(message)s'

  The Logging format template

- **–runtime-dir=<Unicode> (BaseJaffleCommand.runtime_dir)** Default: '.jaffle'

  Runtime directory path.

1.2.4 jaffle attach

Opens an interactive shell and attaches to the specified app. The app must support attaching. Only *PyTestRunnerApp* supports this.

Type `ctrl-c` or `ctrl-d` to stop it.

Usage

```
jaffle attach <app> [options]
```

Options

- **–debug**

  Set log level to logging.DEBUG (maximize logging output)

- **-y**

  Answer yes to any questions instead of prompting.

- **–disable-color**

  Disable color output.

- **–log-level=<Enum>** (Application.log_level)

  Default: 30

  Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL') Set the log level by value or name.

- **–log-datefmt=<Unicode>** (Application.log_datefmt)

  Default: '%Y-%m-%d %H:%M:%S'

  The date format used by logging formatters for %(asctime)s

- **–log-format=<Unicode>** (Application.log_format)

  Default: '%(time_color)s%(asctime)s.%(msecs).03d%(time_color_end)s %(name_color)s%(name)14s%(name_color_end)s %(level_color)s %(levelname)1.1s %(level_color_end)s %(message)s'

  The Logging format template

- **–runtime-dir=<Unicode> (BaseJaffleCommand.runtime_dir)** Default: '.jaffle'

  Runtime directory path.

1.3 Configuration

Note: Currently Jaffle does not check the configuration file syntax. If Jaffle does not work as you expect, please check the configuration carefully. Jaffle will have the configuration validation in the future release.

1.3.1 Syntax

Configuration Syntax

The configuration language of `jaffle.hcl` is HCL (HashiCorp Configuration Language).

The top-level of the configuration can have the following items:

- *kernel*
- *app*
- *process*
- *job*
- *logger*
- *variable*

Example

```
kernel "py_kernel" {}

app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  logger {
    level = "info"
  }

  options {
    handlers = [{
      watch_path         = "my_module"
      patterns           = ["*.py"]
      ignore_directories = true
      functions          = ["pytest.handle_watchdog_event({event})"]
    }]
  }
}

app "pytest" {
  class  = "jaffle.app.pytest.PyTestRunnerApp"
  kernel = "py_kernel"

  logger {
    level = "info"
  }

  options {
    args = ["-s", "-v", "--color=yes"]

    auto_test = [
      "my_module/tests/test_*.py",
    ]

    auto_test_map {
```

(continues on next page)

(continued from previous page)

```
      "my_module/**/*.py" = "my_module/tests/{}/test_{}.py"
    }
  }
}
```

JSON

Since JSON is a valid HCL, you can also write the configuration file as JSON. The previous HCL example is same as the following JSON.

```
{
  "kernel": {
    "py_kernel": {}
  },
  "app": {
    "watchdog": {
      "class": "jaffle.app.watchdog.WatchdogApp",
      "kernel": "py_kernel",
      "logger": {
        "level": "info"
      },
      "options": {
        "handlers": [
          {
            "watch_path": "my_module",
            "patterns": [
              "*.py"
            ],
            "ignore_directories": true,
            "functions": [
              "pytest.handle_watchdog_event({event})"
            ]
          }
        ]
      }
    },
    "pytest": {
      "class": "jaffle.app.pytest.PyTestRunnerApp",
      "kernel": "py_kernel",
      "logger": {
        "level": "info"
      },
      "options": {
        "args": [
          "-s",
          "-v",
          "--color=yes"
        ],
        "auto_test": [
          "my_module/tests/test_*.py"
        ],
        "auto_test_map": {
          "my_module/**/*.py": "my_module/tests/{}/test_{}.py"
        }
      }
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

Interpolation Syntax

Jaffle configuration supports interpolation syntax wrapped by `${}`. You can get *environment varialbes*, call *functions*, and execute Python code in it:

Example:

```
${'hello'.upper()}
```

The above produces `'HELLO'`.

Environment Variables

All environment variables consist of alphanumeric uppercase characters are available in the interpolation syntax.

Example:

```
${HOME}/etc
```

The above produces `/home/your_account/etc` if your `HOME` is `'/home/your_account'`.

If you need a default value for an environment variable, use *env()* function instead.

Variables

Defined variables can be embedded with `${var.name}` syntax in arbitrary HCL value part.

Example:

```
disabled = "${var.enable_debug}"
```

See *variable* section for details.

Functions

env()

env (*name*, *default=''*)

Gets an environment variable.

Parameters

- **name** (*str*) – Environment variable name.
- **default** (*str*) – Default value.

Returns **env** – Value of the environment variable.

Return type str

exec()

exec(*command*)

Executes a command and returns the result of it.

Parameters **command** (*str*) – Command name and arguments separated by whitespaces.

Returns **result** – Result of the command.

Return type str

fg()

fg(*color*)

Inserts the escape sequence of the foreground color.

Available colors are 'black', 'red', 'green', 'yellow', 'blue', 'magenta', 'cyan', 'white', 'bright_black', 'bright_red', 'bright_green', 'bright_yellow', 'bright_blue' , 'bright_magenta', 'bright_cyan' and 'bright_white'.

Parameters **color** (*str*) – Foreground color in str (e.g. 'red').

Returns **seq** – Escape sequence of the foreground color.

Return type str

Raises `ValueError` – Invalid color name.

bg()

bg(*color*)

Inserts the escape sequence of the background color.

Available colors are 'black', 'red', 'green', 'yellow', 'blue', 'magenta', 'cyan', 'white', 'bright_black', 'bright_red', 'bright_green', 'bright_yellow', 'bright_blue' , 'bright_magenta', 'bright_cyan' and 'bright_white'.

Parameters **color** (*str*) – Background color in str (e.g. 'red').

Returns **seq** – Escape sequence of the background color.

Return type str

Raises `ValueError` – Invalid color name.

reset()

reset()

Inserts the escape sequence of display reeet.

Returns **seq** – Escape sequence of display reeet.

Return type str

jq_all()

jq_all(*query*, *data_str*, **args*, ***kwargs*)
Queries the nested data and returns all results as a list.

Parameters **data_str** (*str*) – Nested data in Python dict's representation format. If must be loadable by `yaml.safe_load()`.

Returns **result** – String representation of the result list.

Return type str

pyjq processes the query. `jq()` is an alias to `jq_all()`.

jq_first()

jq_first(*query*, *data_str*, **args*, ***kwargs*)
Queries the nested data and returns the first result.

Parameters **data_str** (*str*) – Nested data in Python dict's representation format. If must be loadable by `yaml.safe_load()`.

Returns **result** – String representation of the result object.

Return type str

pyjq processes the query. `jqf()` is an alias to `jq_first()`.

Filters

The `|` operator can be used in a `${}` expression to apply filters.

Example:

```
${'hello world' | u}
```

The `u` filter applies URL escaping to the string, and produces `'hello+world'`.

To apply more than one filter, separate them by a comma:

```
${'  hello world  ' | trim,u}
```

The above produces `'hello+world'`.

Available Filters

u URL escaping.

```
${"hello <b>world</b>" | x} => 'hello+world'
```

h HTML escaping.

```
${"hello <b>world</b>" | x} => 'hello &lt;b&gt;world&lt;/b&gt;'
```

x XML escaping.

```
${"hello <b>world</b>" | x} => 'hello &lt;b&gt;world&lt;/b&gt;'
```

trim Whitespace trimming.

```
${" hello world " | x} => 'hello world'
```

entity Produces HTML entity references for applicable strings.

```
${"→" | entit} => '&rarr;'
```

1.3.2 Configuration Blocks

kernel

Example

The `kernel` block defines a kernel instance name and configures the kernel.

```
kernel "py_kernel" {
  kernel_name = "python3"
  pass_env = ["PATH", "HOME"]
}
```

Description

- **kernel_name** (str | optional | default: `""`)

  `kernel_name` is a Jupyter kernel name. You can install multiple kernels and switch them by specifying `kernel_name`. If it is not specified, the default kernel will be launched. The kernel must be IPython kernel and the Python version must be greater than or equal to 3.4. See also Installing the IPython kernel in the IPython document.

- **pass_env** ([str] | optional | default: [])

  `pass_env` defines environment variables which will be passed to the kernel. Jaffle itself has the environment variables defined in your environment, but the kernel will be launched as an independent process and the environment variables are not passed by default.

  Tip: If the kernel executes a Python console script in a virtualenv, you will have to pass `PATH` environment variable to the kernel.

  ---

app

The `app` block configures a *Jaffle app* which will be launched in a kernel. The name next to `app` keyword will be the variable name in the kernel and will be accessed from other configuration blocks. The name must be valid in an IPython kernel.

Example

```
app "pytest" {
  class  = "jaffle.app.pytest.PyTestRunnerApp"
  kernel = "py_kernel"

```

(continues on next page)

(continued from previous page)

```
  options {
    args = ["-s", "-v", "--color=yes"]

    auto_test = [
      "my_module/tests/test_*.py",
    ]

    auto_test_map {
      "my_module/**/*.py" = "my_module/tests/{}/test_{}.py"
    }
  }
}
```

Description

- **class** (str | required)

  The class name of the Jaffle app. It must begin with the top-level module name. e.g.: `"jaffle.app.pytest.PyTestRunnerApp"`.

- **kernel** (str | required)

  The kernel in which the app is instantiated. The specified kernel must be defined in a *kernel* block.

- **start** (str | optional | default: `null`)

  Python code to be executed just after the app is instanticated in a kernel.

- **logger** (*logger* | optional | default: `{}`)

  The app logger configuration.

- **options** (map | optional | default: `{}`)

  `options` will be passed to the app initializer (`__init__()` method) as keyword arguments. The format of `options` depends on each *app*.

process

The `process` block configures an external process. The output to `stdout` and `stderr` are redirected to the logger with level `info` and `warning` respectively.

Example

```
process "webdev" {
  command = "yarn start"
  tty     = true

  env {
    BROWSER = "none"
  }
}
```

Description

- **command** (str | required)

  The command and arguments separated by whitespaces.

- **tty** (bool | optional | default: `false`)

  Whether to enable special care for a TTY application. Some applications require a foreground TTY access and/or send escape sequences aggressively. When `tty` is true, Jaffle runs the process via Pexpect and filters the output. Font style sequences are still available but all other escape sequences will be dropped. Try this option if your command does not work or makes the log output collapse.

- **env** (map | optional | default: `{}`)

  The environment variables to be passed to the process.

- **logger** (*logger* | optional | default: `{}`)

  The process logger configuration.

job

The `job` block configures a job which can be executed from a Jaffle app.

Example

```
job "sphinx" {
  command = "sphinx-build -M html docs docs/_build"
}
```

Here is an *WatchdogApp* configuration which executes the job:

```
app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  options {
    handlers = [
      {
        patterns           = ["*/my_module/*.py", "*/docs/*.*"]
        ignore_patterns    = ["*/_build/*"]
        ignore_directories = true
        jobs               = ["sphinx"]
      },
    ]
  }
}
```

Description

- **command** (str | required)

  The command and arguments separated by whitespaces.

- **logger** (*logger* | optional | default: `{}`)

The job logger configuration.

Jaffle Apps

Only *WatchdogApp* supports executing jobs.

logger

The `logger` block configures log suppressing and replacing rules by regular expressions. `logger` is available in the root, `app` and `process` blocks. The root `logger` configures the global rules which are applied after each app- or process-level rule.

Example

```
logger {
  suppress_regex = ["^\\s*$"] # drop empty line
  replace_regex = [
    {
      from = "(some_keyword)"
      to   = "\033[31m\\1\033[0m" # red color
    },
  ]
}
```

Description

- **name** (str | optional | default: <object name>)

  The logger name. The root `logger` does not have this.

  Note: Each logger should have a unique logger name. If multiple loggers of apps, process or jobs have the same logger name, `level`, `suppress_regex`, etc. are overwritten multiple times and the last configuration takes effect. That may not be the expected behavior.

- **level** (str | optional | default: `'info'`)

  The logger level. Log messages are filtered by this level. Available levels are 'critical', 'error', 'warning', 'info' and 'debug'. See Python `logging` reference for more information.

- **suppress_regex** ([str] | optional | default: `[]`)

  Regular expression patterns to suppress log messages. If one of the patterns matches the log message, the message will be omitted.

- **replace_regex** ([{"from": str, "to": str}] | optional | default: `[]`)

  The matched groups can be used in `to` string as `\\1`, `\\2`, and so on. Note that `\` (backslash) must be escaped by an extra `\`, such as `\\n`.

  Tip: `replace_regex` is especially useful to emphasize keywords on debugging like the example below.

variable

The `variable` block defines a variable which will be used in another blocks. The variables can be set from environment variables (`J_VAR_name=value`) or the command argument (`--variables='["name=value"]'`).

Example

```
variable "disable_frontend" {
  type = "bool"
  default = false
}

process "frontend" {
  command  = "yarn start"
  tty      = true
  disabled = "${var.disable_frontend}"
}
```

Description

- **type** (str | optional | default: undefined)

  The type of the variable. Available types are 'str', 'bool', 'int', 'float', 'list' and 'dict'.

- **default** (object | optional | default: undefined)

  The default value of the variable. If it is not defined, the value must be provided at runtime from an environment variable or the command-line argument.

If `type` is not provided, it will be inferred based on `default`. If `default` is not provided, it is assumed to be `str`.

Embedding Variables

The variable embedding can be used only in a string:

```
disabled = "${var.disable_frontend}" # OK
```

It cannot be used outside of a string even though the target attribute requires bool or int because it is not a valid HCL:

```
disabled = ${var.disable_frontend} # NG
```

In Jaffle, the following strings can be treated as boolean values:

- `'true'` and `'1'` => `true`
- `'false'` and `'0'` => `false`

```
disabled = false
```

Setting Variables

Your can set values to the variables from environment variables (`J_VAR_name=value`) or the command argument (`--variables='["name=value"]'`).

Example: Setting `true` to `disable_frontend` from an environment variable:

```
$ J_VAR_disable_frontend=true jaffle start
```

Example: Setting `true` to `disable_frontend` from the command-line argument:

```
$ jaffle start --variables='["disable_frontend=true"]'
```

1.4 Jaffle Apps

1.4.1 Built-in Apps

WatchdogApp

WatchdogApp launches Watchdog handlers with given patterns and callback code blocks. Since Jaffle is initially designed to be an automation tool, WatchdogApp is regarded as the central app among other Jaffle apps.

Watchdog is a Python API library and shell utilities to monitor file system events.

Example Configuration

```
app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  options {
    handlers = [{
      patterns           = ["*.py"]
      ignore_patterns    = ["*/tests/*.py"]
      ignore_directories = true
      functions          = ["pytest.handle_watchdog_event({event})"]
    }]
  }
}
```

Options

- **handlers** (list[dict] | optional | default: [])

  Watchdog handler definitions. The dict format is described below.

Handler dict Format

- **watch_path** (str | optional | default: current_directory)

  The directory to be watched by the handler. Both absolute and relative paths are available.

- **patterns** (list[str] | optional | default: [])

  The path matching patterns to execute handler code blocks and jobs. The pattern syntax is the same as Python's fnmatch. Since the Watchdog event has an absolute file path, you will probably need `*` at the beginning of the pattern (e.g.: `patterns = ["*/foo/*.py"]`).

> **Note:** The Watchdog pattern syntax and the *PyTestRunner* pattern syntax are difference from each other. They may be changed to be identical in the future release.

- **ignore_patterns** (list[str] | optional | default: [])

  The path matching patterns to be ignored. The pattern syntax is the same as `patterns`.

- **ignore_directories** (bool | optional | default: false)

  Whether to ignore Watchdog events of directories.

- **throttle** (float | optional | default: 0.0)

  The throttle time in seconds for event handling. When an event is handled, the event handling is disabled until the throttle time passes by. If it is `0`, the throttling is disabled.

- **debounce** (float | optional | default: 0.0)

  The debounce time in seconds for event handling. The event will be handled only when the debounce time has passed without receiving any other events. If it is `0`, the debouncing is disabled.

> **Tip:** Throttling and debouncing are useful when your editor or any other app does multiple file-system operations at once. For example, when you save a file in an editor, the editor may write the file twice to do auto-formatting. In this case, two events are going to be handled each time you save a file and you might want to handle the event only once. `throttle` and `debounce` come into play in this situation.

- **code_blocks** (list[str] | optional | default: [])

  The code blocks to be executed by the handler.

- **jobs** (list[str] optional | default: [])

  The jobs to be executed by the handler. Jobs must be defined in *job* blocks.

- **clear_cache** (list[str] | optional | default: <modules found under the current directory>)

  The module names which will be removed from the module cache (`sys.modules`) before executing handler code blocks.

Integration with Other Apps

WatchdogApp handler executes Python code written in `code_blocks`, with replacing the interpolation keyword `{event}` with an watchdog.events.FileSystemEvent object.

Example:

```
code_blocks = ["pytest.handle_watchdog_event({event})"]
```

PyTestRunnerApp and *TornadoBridgeApp* has `handle_watchdog_event()` to handle the Watchdog event.

PyTestRunnerApp

PyTestRunnerApp runs pytest on receiving Watchdog events sent from *WatchdogApp*. That works very fast because PyTestRunnerApp runs pytest as a Python function in a Jupyter kernel process instead of executing the external `py.test` command, and it also keeps cache of imported modules which do not require reloading.

PyTestRunnerApp also has the *interactive shell* which allows you to run tests interactively.

Example Configuration

```
app "pytest" {
  class  = "jaffle.app.pytest.PyTestRunnerApp"
  kernel = "py_kernel"

  options {
    args = ["-s", "-v", "--color=yes"]

    auto_test = [
      "jaffle_tornado_spa_example/tests/test_*.py",
    ]

    auto_test_map {
      "jaffle_tornado_spa_example/**/*.py" = "jaffle_tornado_spa_example/tests/{}/
↪test_{}.py"
    }
  }
}
```

Optionns

- **args** (list[str] | optional | default: [])

  The pytest arguments.

- **auto_test**

  The file path patterns to be executed by pytest. The pattern syntax is the same as shell glob but supports only `*` and `**`. `*` matches arbitrary characters except for `/` (slash), whereas `**` matches all characters.

- **auto_test_map**

  The file path patterns map to determine test files to be executed. If the event path matches to the left-hand-side pattern, the files which match the right-hand-side will be executed. The pattern syntax is the same as `auto_test`. The strings matched to `*` or `**` in the left-hand-side will be expanded into `{}` in the right-hand-side one by one.

  Tip: It is recommended to create a Python implimentation file and a unit test file to have one-to-one correspondence to each other. That makes easy to setup `auto_test_map`.

  If you editor supports jumping to alternative file like vim-projectionist, it also helps a lot.

- **clear_cache** (list[str] | optional | default: <modules found under the current directory>)

  The module names which will be removed from the module cache (`sys.modules`) before restarting the app. If it is not provided, TornadoBridgeApp searches modules by calling `setuptools.find_packages()`. Note that the root Python module must be in the current working directory to be found by TornadoBridgeApp. If it is included in a sub-directory, you must specify `clear_cache` manually.

Interactive Shell

You can use an interactive shell which attaches the session to PyTestRunnerApp running in a Jupyter kernel.

Example:

```
$ jaffle attach pytest
```

You can type test case names with auto-completion. The tests are executed in the Jupyter kernel.

TornadoBridgeApp

TornadoBridgeApp manages a Tornado application in IPython kernels running in a Jaffle.

Example Configuration

```
app "tornado_app" {
  class  = "jaffle.app.tornado.TornadoBridgeApp"
  kernel = "py_kernel"
  start  = "tornado_app.start()"

  logger {
    level = "info",
  }

  options {
    app_class          = "my_module.app.ExampleApp"
    argv               = ["--port=9999"]
    threaded           = true
    clear_cache = ["my_module"]
  }
}
```

Options

- **app_class** (str | required | default: undefined)

  The Tornado application class to be launched in a kernel. It must be a fully qualified class name which begins from the top module name joined with `.`, e.g. `foo.bar.BazApp`.

- **argv** (list[str] | optional | default: [])

  The arguments to the Tornado application. They will be passed directly to `__init__()` of the class.

- **threaded** (bool | optional | default: false)

  Whether to launch the app in an independent IO loop thread. Tornado applications can basically be launched in the main thread and share the IO loop with other apps and the Jaffle itself. However, some apps cannot dispose all running functions from the IO loop and that makes troubles on calling `start()` and `stop()` several times, because the remaining functions may cause errors. When `threaded` is true, the app uses its own IO loop which will be stopped together with the app itself.

- **clear_cache** (list[str] | optional | default: <modules found under the current directory>)

The module names which will be removed from the module cache (`sys.modules`) before restarting the app. If it is not provided, TornadoBridgeApp searches modules by calling `setuptools.find_packages()`. Note that the root Python module must be in the current working directory to be found by TornadoBridgeApp. If it is included in a sub-directory, you must specify `clear_cache` manually.

Available Tornado Applications

TornadoBridgeApp assumes that the Tornado application has `start()` and `stop()` and they meet the following requirements:

- `start()` gets the IOLoop by calling `tornado.ioloop.IOLoop.current()`.
- `IOLoop.start()` is called only from `start()`.
- `IOLoop.stop()` is called only from an IOLoop callback which is added by `stop()`.

Example:

```
class ExampleApp(Application):

    def start(self):
        self.io_loop = ioloop.IOLoop.current()
        try:
            self.io_loop.start()
        except KeyboardInterrupt:
            self.log.info('Interrupt')

    def stop(self):
        def _stop():
            self.http_server.stop()
            self.io_loop.stop()
        self.io_loop.add_callback(_stop)
```

They are required because Jaffle must protect the main IOLoop not to be terminated or overwritten by the app. If your application cannot meet the requirements, you can create a custom Jaffle app inheriting `TornadoBridgeApp`.

Integration with WatchdogApp

`TornadoBridgeApp.handle_watchdog_event()` handles an Watchdog event sent from WatchdogApp. It restarts the Tornado application.

Example WatchdogApp configuration:

```
app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  options {
    handlers = [
      {
        patterns           = ["*.py"]
        ignore_directories = true
        functions          = ["my_app.handle_watchdog_event({event})"]
      },
    ]
```

(continues on next page)

(continued from previous page)

```
  }
}

app "my_app" {
  class  = "jaffle.app.tornado.TornadoBridgeApp"
  kernel = "py_kernel"
  start  = "tornado_app.start()"

  options {
    app_class = "my_module.app.ExampleApp"
  }
}
```

1.4.2 Custom Apps

You can create your own Jaffle app by inheriting *BaseJaffleApp*. See *Developers Guide* for more information.

1.5 Cookbook

1.5.1 Auto-testing with pytest

You can setup auto-testing by using *WatchdogApp* and *PyTestRunnerApp*.

Here is the example `jaffle.hcl`, which can be used by `jaffle start`.

```
kernel "py_kernel" {}

app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  options {
    handlers = [{
      watch_path         = "pytest_example"
      patterns           = ["*.py"]
      ignore_directories = true
      code_blocks        = ["pytest.handle_watchdog_event({event})"]
    }]
  }
}

app "pytest" {
  class  = "jaffle.app.pytest.PyTestRunnerApp"
  kernel = "py_kernel"

  options {
    args = ["-s", "-v", "--color=yes"]

    auto_test = [
      "pytest_example/tests/test_*.py",
    ]

    auto_test_map {
```

(continues on next page)

(continued from previous page)

```
29        "pytest_example/**/*.py" = "pytest_example/tests/{}/test_{}.py"
30      }
31    }
32  }
```

- L1: Define the kernel `py_kernel` which is used by `watchdog` and `pytest`.
- L3-5: Create WatchdogApp with name `watchdog` in the kernel `py_kernel`.
- L9-11: Let Watchdog watch the directory `pytest_example` with provided patterns.
- L12: When an event comes, the handler executes this code block. The variable `pytest` is an app created later (L17).
- L17-19: Define PyTestRunnerApp with name `pytest` in the kernel `py_kernel`.
- L24-26: When `pytest_example/tests/test_*.py` is modified, pytest executes it.
- L28-30: When `pytest_example/**/*.py` is modified, pytest executes the file matched to the pattern `pytest_example/tests/{}/test_{}.py`.

Interactive Shell

You can also use the interactive shell which attaches the session to the running pytest instance:

```
$ jaffle attach pytest
```

When you hit `t TAB /`, test cases are auto-completed.

Screenshot

Note: The source package of Jaffle contains example projects in `examples` directory. You can see the latest version of them here: https://github.com/yatsu/jaffle/tree/master/examples

A pytest example is here: https://github.com/yatsu/jaffle/tree/master/examples/pytest

1.5.2 Automatic Sphinx Document Build

Here is a simple example which generates a Sphinx document on detecting `*.rst` update. It assumes `.rst` files are stored in `docs` directory and the result HTML will be stored in `docs/_build`.

jaffle.hcl:

```
1  kernel "py_kernel" {
2    pass_env = ["PATH"] # required to run sphinx-build in virtualenv
3  }
4
5  app "watchdog" {
6    class  = "jaffle.app.watchdog.WatchdogApp"
7    kernel = "py_kernel"
8
9    options {
```

(continues on next page)

(continued from previous page)

```
    handlers = [{
      patterns           = ["*/docs/*.*"]
      ignore_patterns    = ["*/_build/*"]
      ignore_directories = true
      jobs               = ["sphinx"]
    }]
  }
}

job "sphinx" {
  command = "sphinx-build -M html docs docs/_build"
}
```

- L1-3: Define the kernel `py_kernel` which is used by `watchdog` and `pytest`. You need to pass `PATH` environment variable if `sphinx-build` is installed in a virtualenv.
- L5-7: Create WatchdogApp with name `watchdog` in the kernel `py_kernel`.
- L10-13: Let **Watchdog_** watch the directory `pytest_example` with provided patterns.
- L14: When an event comes, the handler executes the job `sphinx` which will be defined later (L19-21)
- L19-21: Define `sphinx` job to execute `sphinx-build`

Note: Ignoreing `_build` directory is important (L12 of the above example). If you forget that, `sphinx` job updates `_build` directory and that triggers `sphinx` job again. That will be an infinite loop.

Refreshing Browser

Here is another example on macOS which also refreshes Google Chrome's current tab on detecting file updates.

```
kernel "py_kernel" {
  pass_env = ["PATH"]
}

app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  options {
    handlers = [{
      patterns           = ["*/docs/*.*"]
      ignore_patterns    = ["*/_build/*"]
      ignore_directories = true
      jobs = [
        "sphinx",
        "chrome_refresh",
      ]
    }]
  }
}

job "sphinx" {
  command = "sphinx-build -M html docs docs/_build"
}
```

(continues on next page)

(continued from previous page)

```
job "chrome_refresh" {
  command = "osascript chrome_refresh.scpt"
}
```

You also need the AppleScript file `chrome_refresh.scpt` in the current directory as below.

```
tell application "Google Chrome" to tell the active tab of its first window
    reload
end tell
```

Tip: On Linux, maybe you can use xdotool to refresh your browser.

Note: The source package of Jaffle contains example projects in `examples` directory. You can see the latest version of them here: https://github.com/yatsu/jaffle/tree/master/examples

Jaffle uses the above configuration to generate this Sphinx document: https://github.com/yatsu/jaffle/tree/master/jaffle.hcl

1.5.3 Web Development with Tornado and React

This is an example Jaffle configuration for the web development which uses Tornado and React to build the back-end API and the front-end web interface respectively.

It does:

- Launch the Tornado application including HTTP server
- Launch the Webpack dev server as an external process by executing `yarn start`
- Launch Jest as an external process by executing `yarn test`
- Restart the Tornado application when a related file is updated
- Execute pytest when a related file is updated

This page assumes that you have already know the basic configuration for a pytest. If not, please read the section *Auto-testing with pytest*.

jaffle.hcl:

```
kernel "py_kernel" {}

app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  options {
    handlers = [
      {
        watch_path         = "tornado_spa"
        patterns           = ["*.py"]
        ignore_patterns    = ["*/tests/*.py"]
        ignore_directories = true
```

(continues on next page)

(continued from previous page)

```
        clear_cache        = ["tornado_spa"]

        code_blocks = [
          "tornado_app.handle_watchdog_event({event})",
          "pytest.handle_watchdog_event({event})",
        ]
      },
      {
        watch_path         = "tornado_spa/tests"
        patterns           = ["*/test_*.py"]
        ignore_directories = true
        clear_cache        = ["tornado_spa.tests"]

        code_blocks = [
          "pytest.handle_watchdog_event({event})",
        ]
      },
    ]
  }
}

app "tornado_app" {
  class  = "jaffle.app.tornado.TornadoBridgeApp"
  kernel = "py_kernel"
  start  = "tornado_app.start()"

  options {
    app_class   = "tornado_spa.app.ExampleApp"
    args        = ["--port=9999"]
    clear_cache = []
  }
}

app "pytest" {
  class  = "jaffle.app.pytest.PyTestRunnerApp"
  kernel = "py_kernel"

  options {
    args = ["-s", "-v", "--color=yes"]

    auto_test = [
      "tornado_spa/tests/test_*.py",
    ]

    auto_test_map {
      "tornado_spa/**/*.py" = "tornado_spa/tests/{}/test_{}.py"
    }

    clear_cache = []
  }
}

process "frontend" {
  command = "yarn start"
  tty     = true

  env {
```

(continues on next page)

(continued from previous page)

```
    BROWSER = "none"
  }
}

process "jest" {
  command = "yarn test"
  tty     = true
}
```

Clearing Module Cache

Since two applications `tornado_app` and `pytest` run in the same Jupyter kernel and share the same Python modules in memory, you should manually configure the cache clear. By default *TornadoBridgeApp* and *PyTestRunnerApp* clear the modules found under the current directory on receiving an Watchdog event. That causes duplicated cache clear on the same module. To prevent that, the configuration above has `clear_cache = []` in both `tornado_app` and `pytest` to disable cache clear and has `clear_cache = ["tornado_spa"]` in `watchdog` to let *WatchdogApp* clear the module cache instead.

Note: If `clear_cache` configuration is incorrect, *TornadoBridgeApp* or *PyTestRunnerApp* may not reload Python modules.

Screenshot

Note: The source package of Jaffle contains example projects in `examples` directory. You can see the latest version of them here: https://github.com/yatsu/jaffle/tree/master/examples

A Tornado and React example is here: https://github.com/yatsu/jaffle/tree/master/examples/tornado_spa

---

1.5.4 Jupyter Extension Development

This page assumes that you have already know the basic configuration for a Tornado application. If not, please read the section *Web Development with Tornado and React*.

To execute examples/jupyter_ext, you need to setup the Python project and install Jupyter serverextension and nbextension first.

Example setup:

```
$ cd example/jupyter_ext
$ pip install -e .
$ jupyter serverextension install jupyter_myext --user
$ jupyter nbextension install jupyter_myext --user
```

Here is the Jaffle configuration.

jaffle.hcl:

```
kernel "py_kernel" {}

app "watchdog" {
  class  = "jaffle.app.watchdog.WatchdogApp"
  kernel = "py_kernel"

  options {
    handlers = [
      {
        patterns           = ["*.py"]
        ignore_patterns    = ["*/tests/*.py"]
        ignore_directories = true
        clear_cache        = ["jupyter_myext"]

        code_blocks = [
          "notebook.handle_watchdog_event({event})",
          "pytest.handle_watchdog_event({event})",
        ]
      },
      {
        patterns           = ["*/tests/test_*.py"]
        ignore_directories = true
        clear_cache        = ["jupyter_myext.tests"]

        code_blocks = [
          "pytest.handle_watchdog_event({event})",
        ]
      },
      {
        patterns           = ["*.js"]
        ignore_directories = true

        code_blocks = [
          "nbext_install.handle_watchdog_event({event})",
        ]
      },
    ]
  }
}

app "notebook" {
  class  = "jaffle.app.tornado.TornadoBridgeApp"
  kernel = "py_kernel"

  options {
    app_class = "notebook.notebookapp.NotebookApp"

    args = [
      "--port=9999",
      "--NotebookApp.token=''",
    ]

    clear_cache = []
  }

  start = "notebook.start()"
}
```

(continues on next page)

(continued from previous page)

```
58
59  app "pytest" {
60    class  = "jaffle.app.pytest.PyTestRunnerApp"
61    kernel = "py_kernel"
62
63    options {
64      args = ["-s", "--color=yes"]
65
66      auto_test = [
67        "jupyter_myext/tests/test_*.py",
68      ]
69
70      auto_test_map {
71        "jupyter_myext/**/*.py" = "jupyter_myext/tests/{}/test_{}.py"
72      }
73
74      clear_cache = []
75    }
76  }
77
78  app "nbext_install" {
79    class  = "jupyter_myext._devel.NBExtensionInstaller"
80    kernel = "py_kernel"
81  }
```

- L10-28: The handler configuration of pytest execution and Tornado restart, same as the example: *Web Development with Tornado and React*.
- L29-36: The handler configuration to install nbextension on detecting `.js` file update.
- L41-57: Launch Jupyter notebook server via `TornadoBridgeApp` with the main IO loop of the kernel process.
- L78-81: The definition of an app that installs the nbextension.

Tip: This example uses `NBExtensionInstaller` to install the Jupyter nbextension. You can define a *job* that executes `jupyter nbextension install --overwrite` instead. If you do so, be sure to set `pass_env = ["PATH"]` in the *kernel* section if Jupyter is installed in a virtualenv.

Note: The source package of Jaffle contains example projects in `examples` directory. You can see the latest version of them here: https://github.com/yatsu/jaffle/tree/master/examples

A Jupyter extension example is here: https://github.com/yatsu/jaffle/tree/master/examples/jupyter_ext

---

1.5.5 Overwriting the Configuration

You might want to add `jaffle.hcl` to your source code repository to share it within your team. At the same time, you might want to run Jaffle with your own customized log filtering. Editing the same `jaffle.hcl` is hard and it may cause an accidental repository commit. Jaffle provides the following two features to overwrite and customize the base configuration.

1. Merging multiple configurations
2. Setting variable from command-line

examples/tornado_spa_advanced is the example which demonstrates them.

Merging Multiple Configurations

You can provide multiple configuration file to `jaffle start`. For example:

```
$ jaffle start jaffle.hcl my_jaffle.hcl
```

Jaffle read the first file and then merges the other files one by one. Maps are merged deeply and other elements are overwritten.

Let's say you have this `jaffle.hcl`.

```
variable "watchdog_log_level" {
  default = "info"
}

app "watchdog" {
  # ...
  logger {
    level = "${var.watchdog_log_level}"
  }
  # ...
}
```

And this `my_jaffle.hcl`.

```
variable "watchdog_log_level" {
  default = "debug" # overwrite "info" => "debug"
}
```

The configuration will be merged as follows.

```
variable "watchdog_log_level" {
  default = "debug"
}

app "watchdog" {
  # ...
  logger {
    level = "${var.watchdog_log_level}"
  }
  # ...
}
```

Please refer to the *Merging Multiple Configurations* section of the *jaffle start Command Reference*.

Setting Variable from Command-line

You can provide *variables* from command-line. The example shown in the previous section can be executed with `debug` log-level as follows.

```
$ J_VAR_watchdog_log_level=debug jaffle start
```

You can also set it by `--variables` option.

```
$ jaffle start --variables='["watchdog_log_level=debug"]'
```

Please refer to the *variable* document.

Note: The source package of Jaffle contains example projects in `examples` directory. You can see the latest version of them here: https://github.com/yatsu/jaffle/tree/master/examples

1.6 Troubleshooting

1.6.1 Debug Logging

`--debug` option enables the debug logging of Jaffle itself.

```
$ jaffle start --debug
```

Each app has its own log-level setting. You can set it in `jaffle.hcl`.

```
app "myapp" {
  # ...

  logger {
    level = "debug"
  }
}
```

You can also set the log-level using a variable like this.

```
variable "myapp_log_level" {
  default = "info"
}

app "myapp" {
  # ...

  logger {
    level = "${var.myapp_log_level}"
  }
}
```

You can switch the log-level by providing the value as an environment variable.

```
$ J_VAR_myapp_log_level=debug jaffle start
```

The command-line argument `--variables` is also avilable to do the same thing.

```
$ jaffle start --variables='["myapp_log_level=debug"]'
```

1.6.2 Jaffle Console

`jaffle console` allows you to open an interactive shell and attaches the session into the running kernel. You can inspect or set variables of running apps in it.

```
$ jaffle console my_kernel
```

1.7 Version History

1.7.1 0.2.1 (May 20, 2018)

- Fix: String interpolations in app options are not evaluated

1.7.2 0.2.0 (May 16, 2018)

- Now String interpolations are evaluated at runtime instead of on loading the configuration
- Add functions `jq_all()` and `jq_first()` and their aliases `jq()` and `jqf()`
- Change environment variable prefix `T_VAR_` to `J_VAR_`
- Simplify `BaseJaffleApp` I/F
- Improve Tornado app stability on syntax errors and exceptions raised in `start()`
- Fix hidden Tornado log messages

1.7.3 0.1.2 (May 8, 2018)

- Add `fg()`, `bg()` and `reset()` function
- Fix errors on starting/stopping threaded Tornado app

1.7.4 0.1.0 (May 6, 2018)

- Initial release

1.8 Related Work

Watchdog Python API and shell utilities to monitor file system events. Jaffle depends on it.

pytest-testmon pytest plugin to select tests affected by recent changes. It looks code coverage to determine which tests should be executed, whereas Jaffle uses simple pattern mapping.

pytest-watch Continuous pytest runner using Watchdog, which also supports notification, before/after hoooks and using a custom runner script. It executes pytest as a subprocess.

Foreman Procfile-based process manager.

coloredlogcat_py and pid_cat Android logcat modifier. Jaffle's log formatter was inspired by them.

1.9 Developers Guide

1.10 API

1.10.1 jaffle.app.base

BaseJaffleApp

class BaseJaffleApp(*app_conf_data*)
Base class for Jaffle apps.

completer_class
The completer class for the interactive shell. It is required only if the app supports interactive shell.

lexer_class
The lexer class for the interactive shell. It is required only if the app supports interactive shell.

classmethod command_to_code(*app_name*, *command*)
Converts a command comes from `jaffle attach <app>` to a code to be executed.

If the app supports `jaffle attach`, this method must be implemented.

Parameters

- **app_name** (*str*) – App name defined in jaffle.hcl.
- **command** (*str*) – Command name received from the shell of `jaffle attach`.

Returns **code** – Code to be executed.

Return type str

execute_code(*code*, **args*, ***kwargs*)
Executes a code.

Parameters

- **code** (*str*) – Code to be executed. It will be formateed as `code.format(*args, **kwargs)`.
- **args** (*list*) – Positional arguments to `code.format()`.
- **kwargs** (*dict*) – Keyward arguments to `code.formmat()`.

Returns **future** – Future which will have the execution result.

Return type tornado.gen.Future

execute_command(*command*, *logger=None*)
Executes a command.

Parameters

- **command** (*str*) – Command to be executed.
- **logger** (*logging.Logger*) – Logger.

Returns **future** – Future which will have the execution result.

Return type tornado.gen.Future

execute_job(*job_name*)
Executes a job.

Parameters **job_name** (*str*) – Job to be executed.

Returns **future** – Future which will have the execution result.

Return type tornado.gen.Future

Utility Functions

capture_method_output(*method*)

Decorator for an app method to capture standard output and redirects it to the logger. `stdout` and `stderr` are logged with level `INFO` and `WARNING` respectively.

Parameters **method** (*function*) – Method to be wrapped.

CHAPTER 2

Source Code

GitHub repository: yatsu/jaffle

CHAPTER 3

Bugs/Requests

Please use the GitHub issue tracker to submit bugs or request features.

CHAPTER 4

License

Jaffle is available under BSD 3-Clause License.

This web site and all documentation are licensed under Creative Commons 3.0.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index