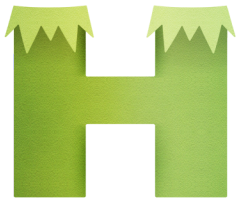
Henson Documentation

Release 1.0.0

iHeartRadio

March 03, 2016

Contents



Henson is a library for building services that are driven by consumers. Henson applications read from objects that implement the Consumer Interface and provide the message received to a callback for processing. The messsage can be processed before handing it off to the callback, and the callback's results can be processed after they are returned to the application.

Note: This documentation uses the `async`/`await` syntax introduced to Python 3.5 by way of PEP 492. If you are using an older version of Python, replace `async` with the `@asyncio.coroutine` decorator and `await` with `yield from`.

CHAPTER 1

Installation

You can install Henson using Pip:

```
$ python -m pip install henson
```

Warning: Henson hasn't been uploaded to the Python Package Index yet. Until that time, it must be installed from source.

You can also install it from source:

```
$ python setup.py install
```

CHAPTER 2

Quickstart

```
from henson import Abort, Application

class FileConsumer:
    """Read lines from a file."""

    def __init__(self, filename):
        self.filename = filename
        self._file = None

    def __iter__(self):
        """FileConsumer objects are iterators."""
        return self

    def __next__(self):
        """Return the next line of the file, if available."""
        if not self._file:
            self._file = open(self.filename)
        try:
            return next(self._file)
        except StopIteration:
            self._file.close()
            raise Abort('Reached end of file', None)

    async def read(self):
        """Return the next line in the file."""
        return next(self)

async def callback(app, message):
    """Print the message retrieved from the file consumer."""
    print(app.name, 'received:', message)
    return message

app = Application(
    __name__,
    callback=callback,
    consumer=FileConsumer(__file__),
)

@app.startup
async def print_header(app):
    """Print a header for the file being processed."""
    print('# Begin processing', app.consumer.filename)
```

```
@app.teardown
async def print_footer(app):
    """Print a footer for the file being processed."""
    print('# Done processing', app.consumer.filename)

@app.message_preprocessor
async def remove_comments(app, line):
    """Abort processing of comments (lines that start with #)."""
    if line.strip().startswith('#'):
        raise Abort('Line is a comment', line)
    return line
```

CHAPTER 3

Running Applications

Henson provides a `henson` command to run your applications from the command line. To run the application defined in the quickstart above, `cd` to the directory containing the module and run:

```
$ henson run file_printer
```

Henson's CLI can also be invoked by running the installed package as a script. To avoid confusion and prevent different installations of Henson from interfering with one another, this is the recommended way to run Henson applications:

```
$ python -m henson run file_printer
```

If a module contains only one instance of a Henson *`Application`*, `python -m henson run` will automatically detect and run it. If more than one instance exists, the desired application's name must be specified:

```
$ python -m henson run file_printer:app
```

This form always takes precedence over the former, and the `henson` command won't attempt to auto-detect an instance even if there is a problem with the name specified. If the attribute specified by the name after `:` is callable, `python -m henson run` will call it and use the returned value as the application. Any callable specified this way should require no arguments and return an instance of *`Application`*. Autodiscovery of callables that return applications is not currently supported.

When developing locally, applications often need to be restarted as changes are made. To make this easier, Henson provides a `--reloader` option to the `run` command. With this option enabled, Henson will watch an application's root directory and restart the application automatically when changes are detected:

```
$ python -m henson run file_printer --reloader
```

Note: The `--reloader` option is not recommended for production use.

It's also possible to enable Henson's *debug mode* through the `--debug` option:

```
$ python -m henson run file_printer --debug
```

CHAPTER 4

Logging

Henson applications provide a default logger. The logger returned by calling `logging.getLogger()` will be used. The name of the logger is the name given to the application. Any configuration needed (e.g., `logging.basicConfig()`, `logging.config.dictConfig()`, etc.) should be done before the application is started.

CHAPTER 5

Debug Mode

Debugging with asyncio can be tricky. Henson provides a debug mode enables asyncio's debug mode as well as debugging information through Henson's logger.

Debug mode can be enabled through a configuration setting:

```
app.settings['DEBUG'] = True
```

or by providing a truthy value for `debug` when calling *`run_forever()`*:

```
app.run_forever(debug=True)
```

Contents:

5.1 Consumer Interface

To work with Henson, a consumer must conform to the Consumer Interface. To conform to the interface, the object must expose a `coroutine()` function named `read`.

Below is a sample implementation.

```
from henson import Abort, Application

class FileConsumer:
    """Read lines from a file."""

    def __init__(self, filename):
        self.filename = filename
        self._file = None

    def __iter__(self):
        """FileConsumer objects are iterators."""
        return self

    def __next__(self):
        """Return the next line of the file, if available."""
        if not self._file:
            self._file = open(self.filename)
        try:
            return next(self._file)
        except StopIteration:
            self._file.close()
            raise Abort('Reached end of file', None)
```

```
    async def read(self):
        """Return the next line in the file."""
        return next(self)

async def callback(app, message):
    """Print the message retrieved from the file consumer."""
    print(app.name, 'received:', message)
    return message

app = Application(
    __name__,
    callback=callback,
    consumer=FileConsumer(__file__),
)

@app.startup
async def print_header(app):
    """Print a header for the file being processed."""
    print('# Begin processing', app.consumer.filename)

@app.teardown
async def print_footer(app):
    """Print a footer for the file being processed."""
    print('# Done processing', app.consumer.filename)

@app.message_preprocessor
async def remove_comments(app, line):
    """Abort processing of comments (lines that start with #)."""
    if line.strip().startswith('#'):
        raise Abort('Line is a comment', line)
    return line
```

5.2 Callbacks

Henson operates on messages through a series of `asyncio.coroutine()` callback functions. Each callback type serves a unique purpose.

5.2.1 `callback`

This is the only one of the callback settings that is required. Its purpose is to process the incoming message. If desired, it should return the result(s) of processing the message as an iterable.

```
async def callback(application, message):
    return ['spam']

Application('name', callback=callback)
```

Note: There can only be one function registered as `callback`.

5.2.2 error

These callbacks are called when an exception is raised while processing a message.

```
app = Application('name')

@app.error
async def log_error(application, message, exception):
    logger.error('spam')
```

Note: Exceptions raised while postprocessing a result will not be processed through these callbacks.

5.2.3 message_acknowledgement

These callbacks are intended to acknowledge that a message has been received and should not be made available to other consumers. They run after a message and its result(s) have been fully processed.

```
app = Application('name')

@app.message_acknowledgement
async def acknowledge_message(application, original_message):
    await original_message.acknowledge()
```

5.2.4 message_preprocessor

These callbacks are called as each message is first received. Any modifications they make to the message will be reflected in what is passed to `callback` for processing.

```
app = Application('name')

@app.message_preprocessor
async def add_process_id(application, message):
    message['pid'] = os.getpid()
    return message
```

5.2.5 result_postprocessor

These callbacks will operate on the result(s) of `callback`. Each callback is applied to each result.

```
app = Application('name')

@app.result_postprocessor
async def store_result(application, result):
    with open('/tmp/result', 'w') as f:
        f.write(result)
```

5.2.6 startup

These callbacks will run as an application is starting.

```
app = Application('name')

@app.startup
async def connect_to_database(application):
    await db.connect(application.settings['DB_HOST'])
```

5.2.7 teardown

These callbacks will run as an application is shutting down.

```
app = Application('name')

@app.teardown
async def disconnect_from_database(application):
    await db.close()
```

5.3 Extensions

Extensions provide additional functionality to applications. Configuration management is shared between applications and extensions in a central location.

5.3.1 Using Extensions

```
from henson import Application
from henson_sqlite import SQLite

app = Application(__name__)
db = SQLite(app)

db.connection.execute('SELECT 1;')
```

5.3.2 Developing Extensions

Henson provides an *Extension* base class to make extension development easier.

```
from henson import Extension

class SQLite(Extension):
    DEFAULT_SETTINGS = {'SQLITE_CONNECTION_STRING': ':memory:'}

    def __init__(self, app=None):
        self._connection = None
        super().__init__(app)

    @property
    def connection(self):
        if not self._connection:
            conn_string = self.app.settings['SQLITE_CONNECTION_STRING']
            self._connection = sqlite3.connect(conn_string)
        return self._connection
```

The *Extension* class provides two special attributes that are meant to be overridden:

- *DEFAULT_SETTINGS* provides default values for an extension's settings during the *init_app()* step. When a value is used by an extension and has a sensible default, it should be stored here (e.g., a database hostname).
- *REQUIRED_SETTINGS* provides a list of keys that are checked for existence during the *init_app()* step. If one or more required settings are not set on the application instance assigned to the extension, a `KeyError` is raised. Extensions should set this when a value is required but has no default (e.g., a database password).

5.3.3 Available Extensions

Several extensions are available for use:

- Henson-AMQP
- Henson-Database
- Henson-Logging

5.4 `contrib` Packages

While it is possible to build your own plugins, the Henson contrib package contains those that we think will most enhance your application.

5.4.1 Retry

Retry is a plugin to add the ability for Henson applications to automatically retry messages that fail to process.

Warning: Retry registers itself as an error callback on the *Application* instance. When doing so, it inserts itself at the beginning of the list of error callbacks. It does this so that it can prevent other callbacks from running. If you have an error callback that you want to run even when retrying a message, you will need to manually inject it into the list of error callbacks *after* initializing Retry.

Configuration

Retry provides a couple of settings to control how many times a message will be retried. `RETRY_THESHOLD` and `RETRY_TIMEOUT` work in tandem. If values are specified for both, whichever limit is reached first will cause Henson to stop retrying the message. By default, Henson will try forever (yes, this is literally insane).

`RETRY_BACKOFF`	A number that, if provided, will be used in conjunction with the number of retry attempts already made to calculate the total delay for the current retry. Defaults to 1.
`RETRY_CALLBACK`	A coroutine that encapsulates the functionality needed to retry the message. `TypeError` will be raised if the callback isn't a `coroutine()`.
`RETRY_DELAY`	The number of seconds to wait before scheduling a retry. If `RETRY_BACKOFF` has a value greater than 1, the delay will increase between each retry. Defaults to 0.
`RETRY_EXCEPTIONS`	An exception or tuple of exceptions that will cause Henson to retry the message. Defaults to *RetryableException*.
`RETRY_THRESHOLD`	The maximum number of times that a Henson application will try to process a message before marking it as a failure. if set to 0, the message will not be retried. If set to None, the limit will be controlled by `RETRY_TIMEOUT`. Defaults to None.
`RETRY_TIMEOUT`	The maximum number of seconds during which a message can be retried. If set to None, the limit will be controlled by `RETRY_THRESHOLD`. Defaults to None.

Usage

Application definition:

```
from henson import Application
from henson.contrib.retry import Retry

async def print_message(app, message):
    print(message)

app = Application('retryable-application', callback=my_callback)
app.settings['RETRY_CALLBACK'] = print_message
Retry(app)
```

Somwhere inside the application:

```
from henson.contrib.retry import RetryableException

async def my_callback(app, message):
    raise RetryableException
```

API

class henson.contrib.retry.**Retry**(*app=None*)
A class that adds retries to an application.

init_app(*app*)
Initialize an Application instance.

Parameters **app** (henson.base.Application) – Application instance to be initialized.

Raises

- TypeError – If the callback isn't a coroutine.
- ValueError – If the delay or backoff is negative.

class henson.contrib.retry.**RetryableException**
Exception to be raised when a message should be retried.

5.5 API

Here's the public API for Henson.

5.5.1 Application

class henson.base.**Application**(*name*, *settings=None*, ***, *consumer=None*, *callback=None*)
A service application.

Each message received from the consumer will be passed to the callback.

Parameters

- **name** (*str*) – The name of the application.
- **settings** (*Optional[object]*) – An object with attributed-based settings.

- **consumer** (*optional*) – Any object that is an iterator or an iterable and yields instances of any type that is supported by `callback`. While this isn't required, it must be provided before the application can be run.
- **callback** (*Optional[asyncio.coroutine]*) – A callable object that takes two arguments, an instance of *henson.base.Application* and the (possibly) preprocessed incoming message. While this isn't required, it must be provided before the application can be run.

error(*callback*)
Register an error callback.

Parameters **callback** (*asyncio.coroutine*) – A callable object that takes three arguments: an instance of *henson.base.Application*, the incoming message, and the exception that was raised. It will be called any time there is an exception while reading a message from the queue.

Returns The callback.

Return type asyncio.coroutine

Raises `TypeError` – If the callback isn't a coroutine.

message_acknowledgement(*callback*)
Register a message acknowledgement callback.

Parameters **callback** (*asyncio.coroutine*) – A callable object that takes two arguments: an instance of *henson.base.Application* and the original incoming message as its only argument. It will be called once a message has been fully processed.

Returns The callback.

Return type asyncio.coroutine

Raises `TypeError` – If the callback isn't a coroutine.

message_preprocessor(*callback*)
Register a message preprocessing callback.

Parameters **callback** (*asyncio.coroutine*) – A callable object that takes two arguments: an instance of *henson.base.Application* and the incoming message. It will be called for each incoming message with its result being passed to `callback`.

Returns The callback.

Return type asyncio.coroutine

Raises `TypeError` – If the callback isn't a coroutine.

result_postprocessor(*callback*)
Register a result postprocessing callback.

Parameters **callback** (*asyncio.coroutine*) – A callable object that takes two arguments: an instance of *henson.base.Application* and a result of processing the incoming message. It will be called for each result returned from `callback`.

Returns The callback.

Return type asyncio.coroutine

Raises `TypeError` – If the callback isn't a coroutine.

run_forever(*num_workers=1*, *loop=None*, *debug=False*)
Consume from the consumer until interrupted.

Parameters

- **num_workers** (*Optional[int]*) – The number of asynchronous tasks to use to process messages received through the consumer. Defaults to 1.
- **loop** (*Optional[asyncio.asyncio.BaseEventLoop]*) – An event loop that, if provided, will be used for running the application. If none is provided, the default event loop will be used.
- **debug** (*Optional[bool]*) – Whether or not to run with debug mode enabled. Defaults to True.

Raises TypeError – If the consumer is None or the callback isn't a coroutine.

startup(*callback*)
Register a startup callback.

Parameters **callback** (*asyncio.coroutine*) – A callable object that takes an instance of *Application* as its only argument. It will be called once when the application first starts up.

Returns The callback.

Return type asyncio.coroutine

Raises TypeError – If the callback isn't a coroutine.

teardown(*callback*)
Register a teardown callback.

Parameters **callback** (*asyncio.coroutine*) – A callable object that takes an instance of *Application* as its only argument. It will be called once when the application is shutting down.

Returns The callback.

Return type asyncio.coroutine

Raises TypeError – If the callback isn't a coroutine.

5.5.2 Configuration

class henson.config.**Config**
Custom mapping used to extend and override an app's settings.

from_mapping(*mapping*)
Convert a mapping into settings.

Uppercase keys of the specified mapping will be used to extend and update the existing settings.

Parameters **mapping** (*dict*) – A mapping encapsulating settings.

from_object(*obj*)
Convert an object into settings.

Uppercase attributes of the specified object will be used to extend and update the existing settings.

Parameters **obj** – An object encapsulating settings. This will typically be a module or class.

5.5.3 Exceptions

Custom exceptions used by Henson.

exception henson.exceptions.**Abort**(*reason*, *message*)
An exception that signals to Henson to stop processing a message.

When this exception is caught by Henson it will immediately stop processing the message. None of the remaining callbacks will be called.

If the exception is caught while processing a result, that result will no longer be processed. Any other results generated by the same message will still be processed.

Parameters

- **reason** (*str*) – The reason the message is being aborted. It should be in the form of "noun.verb" (e.g., "provider.ignored").
- **message** – The message that is being aborted. Usually this will be the incoming message, but it can also be the result.

5.5.4 Extensions

class henson.extensions.**Extension**(*app=None*)
A base class for Hension extensions.

Parameters **app** (*Optional[henson.base.Application]*) – An application instance that has an attribute named settings that contains a mapping of settings to interact with a database.

DEFAULT_SETTINGS
A dict of default settings for the extension.

When a setting is not specified by the application instance and has a default specified, the default value will be used. Extensions should define this where appropriate. Defaults to {}.

REQUIRED_SETTINGS
An iterable of required settings for the extension.

When an extension has required settings that do not have default values, their keys may be specified here. Upon extension initialization, an exception will be raised if a value is not set for each key specified in this list. Extensions should define this where appropriate. Defaults to ().

app
Return the registered app.

init_app(*app*)
Initialize the application.

In addition to associating the extension's default settings with the application, this method will also check for the extension's required settings.

Parameters **app** (henson.base.Application) – An application instance that will be initialized.

5.6 Changelog

5.6.1 Version 1.0.0

Released 2016-03-01

- Initial release

Todo

Testing

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

h