asphalt-feedreader

Release 0.0.post9

Sep 27, 2017

Contents

This Asphalt framework component provides the ability to monitor syndication feeds like RSS and Atom. Arbitrary HTML pages can also be scraped as feeds by means of a custom feed reader class.

Each feed is polled periodically and subscribers are notified of any newly published entries. There is also support for persisting the state of each feed, so as not to report old items again when the application is restarted.

CHAPTER 1

Configuration

The component configuration of asphalt-feedreader lets you configure a number of syndication feeds and a number of *state stores* for them (see below).

A basic configuration that creates a single feed pointing to CNN's top stories might look like this:

```
components:
  feedreader:
    url: http://rss.cnn.com/rss/edition.rss
```

Once the component has started, the feed will be available as a resource of type `FeedReader` named default and accessible as `ctx.feed`.

Notice that the above configuration automatically detects the feed reader class. To avoid the overhead of the initial autodetection, we can tell the component directly what feed reader class to use. Since we know this particular feed is an RSS feed, we can specify the `reader` option accordingly:

```
components:
  feedreader:
    url: http://rss.cnn.com/rss/edition.rss
    reader: rss
```

For reference on what kinds of values are acceptable for the `reader` option, see the documentation of `create_feed()`.

Setting up state stores

State stores are repositories where a feed can save its persistent state, including its current metadata and the entry IDs it has already seen. You can use feeds without state stores but if you restart the application, they may then report previously seen entries again.

The following state stores are provided out of the box:

- `asphalt.feedreader.stores.sqlalchemy`

- `asphalt.feedreader.stores.redis`
- `asphalt.feedreader.stores.mongodb`

Each of these stores requires some database client resource, and the easiest way to get it is to use the corresponding Asphalt component. For example, to configure a simple sqlite based SQLAlchemy store, you'd first install asphalt-sqlalchemy and write the following configuration (following up from the previous examples):

```
components:
  sqlalchemy:
    url: sqlite:///feeds.sqlite
  feedreader:
    feeds:
      cnn:
        url: http://rss.cnn.com/rss/edition.rss
        reader: rss
        store: default
    stores:
      default:
        type: sqlalchemy
```

Any options under each state store configuration besides `type` will be directly passed to the constructor of the store class.

It is also possible to use a custom serializer with the built-in state stores, but that is usually unnecessary.

CHAPTER 2

Using feed readers

Feed readers are used by listening to their `entry_discovered` and `metadata_changed` signals. To continuously print new entries as they come in, just stream events from the signal:

```
from async_generator import aclosing

async def print_events(ctx):
    async with aclosing(ctx.feed.entry_discovered.stream_events()) as stream:
        async for event in stream:
            print('New event: {entry.title}'.format(entry=event.entry))
```

Or, if you prefer callbacks:

```
def new_entry_found(event):
    print('New event: {entry.title}'.format(entry=event.entry))

ctx.feed.entry_discovered.connect(new_entry_found)
```

Note: Each feed reader class may have its own set of entry attributes beyond the ones in `FeedEntry`. See the API documentation for each individual feed reader class.

Creating new feeds on the fly

If you need to create news feeds dynamically during the run time of your application, you can do so using the `create_feed()` function.

CHAPTER 3

Creating custom feed parsers

If you have a website that does not provide an RSS or Atom feed natively, but nonetheless contain news items structured in a manner that *could* be syndicated, it is possible to construct a tailored feed reader class for that particular website. The parser would take the HTML content, parse it into a structured form (using BeautifulSoup or a similar tool) and then extract the necessary information. The process is of course vulnerable to any structural changes made in the HTML, but it's still better than nothing.

To implement an HTML feed parser, you should inherit from the `BaseFeedReader` class and implement the `parse_document()` method. This method must return a two element tuple containing:

- a dictionary of metadata attributes and values (can be just an empty dict)
- a list of dictionaries, each dictionary representing the constructor keyword arguments for `EntryEvent`

How the method extracts this information is entirely up to the implementation, but using either lxml.html or BeautifulSoup directly is usually the most robust method. The implementation needs to return **all** the events found in the document. The matter of filtering already seen events is taken care of in the `update()` method.

The **only** required piece of information for each event is the `id` of the event. This is the unique identifier of the event which will be used for preventing already seen events from being dispatched from the `event_discovered` signal of the feed. Other than that, you can fill in as many of the fields of `EntryEvent` as you like, or subclass the class to contain extra attributes.

An example of a custom feed reader has been provided in examples/custom_html.py.

CHAPTER 4

Version history

This library adheres to Semantic Versioning.

1.0.0

- Initial release
- API reference