revived Documentation

Release

Lorenzo Berni

Feb 17, 2017

Contents:

CHAPTER 1

Revived

A predictable state container for python *heavily* inspired by Redux

While not being a *strict 1:1 port* of Redux API, **Revived** is supposed to do pretty much the same job in the most pythonic way possible.

NOTE: I needed this piece of code to work with the **latest python available at the moment** (3.6). While I am not really caring about other versions, the Travis build is running the test suites on **all the 3.5+ versions**, including the dev ones.

Contents

Documentation

Currently the documentation is not buliding into ReadTheDocs (see issue #11). You can build the documentation locally. Check out *Contribute* section.

Installation

Revived package is available on pypi: to install it use the following command:

```
pip install revived
```

Examples

Usage examples are **coming soon**.

Contribute

1. Clone the repository.
2. Create the virtualenv.
   - using virtualenv:

   ```
   virtualenv ENV
   bin/activate
   ```

   - using virtualfish:

   ```
   vf new ENV
   # optional: automatically load the virtualenv when entering the dir
   vf connect
   ```

3. Update pip and install pip-tools:

   ```
   pip install --upgrade pip  # pip-tools needs pip==6.1 or higher (!)
   pip install pip-tools
   ```

4. Install the dependencies:

   ```
   pip install -r requirements.txt
   ```

5. Build the documentation:

   ```
   cd docs
   make html  # or whatever format you prefer
   ```

6. Work on the revived module. This project uses pip-tools so you want to add your new direct dependencies in `requirements.in` and then compile the `requirements.txt` using:

   ```
   pip-compile requirements.in
   ```

7. Write tests.
8. Run tests:

   ```
   # to have coverage in command line
   pytest --cov revived --pep8 revived tests

   # to have html coverage file in the htmlcov directory
   pytest --cov revived --cov-report html --pep8 revived tests
   ```

9. Check type hints:

   ```
   mypy revived tests
   ```

10. Create a pull request.
11. Profit :)

CHAPTER 2

API Documentation

Store (`revived.store`) module documentation

This module implements the **global state store**, and the `INIT` action and action_creator. This is the entry point of the revived module.

Rationale behind the `Store`

revived.store.Store is the object that brings `actions` and `reducers`. The store has the following responsibilities:

- Holds application state;
- Allows access to state via *revived.store.Store.get_state*;
- Allows state to be updated via *revived.store.Store.dispatch*;
- Registers listeners via *revived.store.Store.subscribe* or *revived.store.Store.subscriber* decorator;
- Handles unregistering of listeners via the function returned by *revived.store.Store.subscribe* or via the property *revived.store.Subscriber.unsubscribe* of the *revived.store.Subscriber* instance.

It's important to note that you'll only have a single store in your application. When you want to split your data handling logic, you'll use `reducer` composition instead of many stores.

Dispatch actions

To dispatch actions the *revived.store.Store.dispatch* method should be used, passing as parameter the result of an action_creator. See more in *revived.action.action* and *revived.action.Action*.

```
# create the store object
store = Store(root_reducer)

# register subscribers
# ...

# dispatch an action using the action_creator <an_action_creator>
store.dispatch(an_action_creator(a_parameter, another_parameter))
```

Subscribe and unsubscribe to state changes

There are two ways to **subscribe** and **usubscribe** to store changes: using the *revived.store.Store.subscribe* method or the *revived.store.Store.subscriber* decorator. Both approaches are equivalent and the choice should be just made based on your taste.

Subscribe using `revived.store.Store.subscribe`

```
# create the store object
store = Store(root_reducer)

# define the function
def a_subscriber():
    # do something!
    pass

# subscribe the function
unsubscribe = store.subscribe(a_subscriber)

# unsubscribe the function
unsubscribe()
```

Subscribe using `revived.store.Store.subscriber`

```
# create the store object
store = Store(root_reducer)

# define and subscribe the function
@store.subscriber
def a_subscriber():
    # do something!
    pass

# unsubscribe the function
a_subscriber.unsubscribe()
```

class revived.store.**ActionType**(**args*, ***kwargs*)
Action types for the store module.

Basically the only type here is the `INIT` one. Reducers should wait for this action to create the initial state for the state subpath they are responsible of.

exception revived.store.**DispatchInReducerError**(**args*, ***kwargs*)
Raised when *revived.store.Store.dispatch* is called in a reducer.

class revived.store.**Store**(*reducer*)

Container object for the global state.

This object is responsible of the global state. Its main responsibilities are:

- Keeping track of all the *subscribers*, and call them on **state changes**.
- Keeping reference to the *reducer* to be used and call it to properly handle **state changes**.

Creates the store, using the given function as `reducer`. At the beginning no callback is subscribed to *store changes*. It is possible to add subscribers later, while there is no way - *at the moment* - to replace the reducer.

Parameters **reducer** (Union[Callable, *Module*]) – The root reducer.

Return type None

dispatch(*action*)

Dispatches an `action`.

This is the only piece of code responsible of *dispatching actions*. When an `action` is dispatched, the state is changed according to the defined root reducer and all the subscribers are called.

The calling order is not guaranteed.

Parameters **action** (*Action*) – The `action` that should be dispatched.

Raises *revived.store.DispatchInReducerError*

Return type None

get_state()

Getter for the global state.

Return type Any

Returns The global state contained into the store.

subscribe(*callback*)

Subscribes a callback to *state changes*.

Every time the state changes, the callback is called. No parameters are passed to the callback. It is responsibility of the callback to actually connect the store with the caller. The returned function can be called without arguments to unsubscribe the callback.

Parameters **callback** (Callable) – The callback to be subscribed.

Return type Callable

Returns The unsubscribe function.

subscriber(*callback*)

Decorator function to subscribe a function to *store changes*.

The subscribed function will be called every time the internal state of the store changes.

NOTE: The decorator function will return the function itself. To unsubscribe the callback the user should use the *revived.store.Subscriber.unsubscribe* function attached into the callback.

Parameters **callback** (Callable) – The callback to be subscribed. :returns: The callback itself.

Return type *Subscriber*

Returns The wrapping subscriber.

class revived.store.**Subscriber**(*callback*, *unsubscribe*)
Wrapper around a subscriber function with the unsubscribe property.

While creating a subscriber using the decorator it is not possible to return the unsubscribe function. So a *revived.store.Subscriber* is created wrapping the callback, that contains the *revived.store.Subscriber.unsubscribe* function to be used to properly unregister the subscriber.

Parameters

- **callback** (Callable) – The callback to be wrapped into the subscriber.
- **unsubscribe** (Callable) – The unsubscribe function for the subscriber.

Return type None

unsubscribe
Property containing the unsubscribe function.

Returns The unsubscribe function for the subscriber.

revived.store.**init**()
Action creator for the init action.

Reducer (revived.reducer) module documentation

This module implements helper functions and classes that can be used to define reducers in the same fashion of redux ones, but using decorators instead of anonymous functions.

Things you **should never do** inside a reducer:

- Mutate its arguments;
- Perform side effects like API calls and routing transitions;
- Call **non-pure** functions.

Given the same arguments, it should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.

Create a reducer

A reducer is a function that looks like this:

```
def dummy(prev, action):
    next = prev
    if action.type == ActionType.DUMMY_ACTION_TYPE:
        # Do something
        return next
```

In order to decrease the amount of required boilerplate revived makes use of a lot of python goodies, especially **decorators**.

While every function can be used as reducer (as long as it takes the proper parameters), the easiest way to create a reducer that handles a specific type of actions is to use the *revived.reducer.reducer* decorator.

```
@reducer(ActionType.DUMMY_ACTION_TYPE)
def dummy(prev, action):
    next = prev
    # Do something
    return next
```

Combine reducers

You can naively combine several `reducers` in this way:

```
def dummy(prev, action):
    next = prev
    if action.type == ActionType.DUMMY_ACTION_TYPE1:
        # Do something
        return next
    elif action.type == ActionType.DUMMY_ACTION_TYPE2:
        # Do something different
        return next
    else:
        return next
```

but this is going to make your `reducer` function huge and barely readable. *revived.reducer* contains utility functions that allows you to create much more readable `reducers`.

Reducers can (*and should*) be combined. You can easily do this combination using *revived.reducer.combine_reducers*.

The following example will produce a `combined reducer` where both the `reducers` will handle the whole subtree passed to it: exactly the same result of the previous snippet of code!

```
@reducer(ActionType.DUMMY_ACTION_TYPE1)
def dummy1(prev, action):
    next = prev
    # Do something
    return next

@reducer(ActionType.DUMMY_ACTION_TYPE2)
def dummy2(prev, action):
    next = prev
    # Do something
    return next

combined_reducer = combine_reducers(dummy1, dummy2)
```

Note: a `combined reducer` is a `reducer` and can be combined again with other reducers allowing you to creare every structure you will ever need in your app.

Pass a subtree of the state

If you want it is possible to pass to a reducer only a subtree of the state passed to the `combined reducer`. To do this you should use keyword arguments in this way:

```
@reducer(ActionType.DUMMY_ACTION_TYPE1)
def dummy1(prev, action):
    next = prev
    # Do something
    return next

@reducer(ActionType.DUMMY_ACTION_TYPE2)
def dummy2(prev, action):
```

```
    next = prev
    # Do something
    return next

combined_reducer = combine_reducers(dummy1, dummy_subtree=dummy2)
```

In this example `dummy1` will receive the whole subtree passed to the `combined_reducer` while `dummy2` will only receive the `dummy_subtree` subtree.

Create a reducer module

A `reducer module` is an utility object that behave exactly like a single `reducer`, but permits to register more `reducers` into it. You will use it to define a bunch of `reducers` that are all handling the same subtree of the `state`.

Note that this is *only a helper construct*, because the following snippet of code:

```
mod = Module()

@mod.reduder(ActionType.DUMMY_ACTION_TYPE1)
def dummy1(prev, action):
    next = prev
    # Do something
    return next

@mod.reduder(ActionType.DUMMY_ACTION_TYPE2)
def dummy2(prev, action):
    next = prev
    # Do something
    return next
```

has exactly the same result of:

```
@reducer(ActionType.DUMMY_ACTION_TYPE1)
def dummy1(prev, action):
    next = prev
    # Do something
    return next

@reducer(ActionType.DUMMY_ACTION_TYPE2)
def dummy2(prev, action):
    next = prev
    # Do something
    return next

module_reducer = combine_reducers(dummy1, dummy2)
```

And of course **you can combine** a `reducer module` with other `reducers` and `reducer modules`.

class `revived.reducer.`**`Module`**

Helper class for module creations.

This is just an helper class: you can obtain the same result using the reducer decorator and then combining all the defined reducers as top-level reducers. The module instance will work exactly as a reducer function, but will call all the registered reducers. The call order is not guaranteed.

Return type None

reducer(*action_type*)
Decorator function to create a reducer.

Creates a reducer attached to the module. This reducer is handling the specified action type and it is going to be ignored in case the action is of a different type.

Parameters **action_type** (*ActionType*) – The action type.

Return type Callable

Returns The reducer function.

revived.reducer.**combine_reducers**(**top_reducers*, ***reducers*)
Create a reducer combining the reducers passed as parameters.

It is possible to use this function to combine top-level reducers or to assign to reducers a specific subpath of the state. The result is a reducer, so it is possible to combine the resulted function with other reducers creating at-will complex reducer trees.

Parameters

- **top_reducers** (Union[Callable, *Module*]) – An optional list of top-level reducers.
- **reducers** (Union[Callable, *Module*]) – An optional list of reducers that will handle a subpath.

Return type Callable

Returns The combined reducer function.

revived.reducer.**reducer**(*action_type*)
Decorator function to create a reducer.

Creates a reducer. This reducer is handling the specified action type and it is going to be ignored in case the action is of a different type.

Parameters **action_type** (*ActionType*) – The action type. :returns: The reducer function.

Return type Callable

Returns The reducer function.

Action (revived.action) module documentation

This module implements helper functions and classes that can be used to define actions and action creators, in the same fashion of redux ones, but using decorators instead of anonymous functions.

Actions and action creators

Actions are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using *revived.store.Store.dispatch*.

Actions are instances of *revived.action.Action*. They have a type property. Types should be defined in an enum inheriting *revived.action.ActionType*. Once your app is large enough, you may want to move them into a separate module.

Action creators are exactly that: functions that create actions. It's easy to conflate the terms action and action creator, so do your best to use the *proper term*.

Define action types

While you are free to define the action type enum as he prefers, it is **strongly suggested** to write them down in this way:

```
from revived.actions import ActionType as BaseActionType

# custom action types enum
class ActionType(BaseActionType):
    AN_ACTION_TYPE = 'an_action_type'
```

Define action creators

While it is possible to explicitly build *revived.action.Action* instances directly, it is **strongly suggested** to create `actions` using `action creators`.

Assuming you are in the same module of the `action types` defined previously, you can define `action creators` in this way:

```
# define the action creator that takes two arguments and returns a
# dictionary with those arguments in an arbitrary way.
@action(ActionTypes.AN_ACTION_TYPE)
def an_action_type_with_parameters(param1, param2):
    return {'1': param1, '2': param2}

# create the action object
action_obj = an_action_type_with_parameters(1, 2)
```

class revived.action.**Action**(*action_type*, *data=None*)

Structure that stores all the required data for an action.

Redux actions are plain objects - ie: python dicts - but having a specific class here helps for type hinting. The rationale behind this is that we store the type as `metadata` instead of part of the action data itself.

While `action_type` is going to be stored as `metadata`, the *revived.action.Action* instance itself is going to behave exactly as a dict, with all the action data inside.

Parameters

- **action_type** (*ActionType*) – The type of the action.
- **data** (Optional[Dict]) – An optional dict containing data. No restriction on depth and members type, as long as the keys are strings.

Return type None

class revived.action.**ActionType**(**args*, ***kwargs*)

Action type base class.

The idea behind this class is to use an unique enum to store action types for each module. Usually there would be no need for such a feature-less class, but it is pretty handy while using type hints.

revived.action.**action**(*action_type*)

Decorator function to use as an `action creator` factory.

This helper function is used to create action creators. The idea behind this is that we just want to define the relevant data as a `dict`, instead of complex objects. This decorator will take care of simple-dict-returning functions preparing the proper *revived.action.Action* instance that is needed by the revived API.

Parameters **action_type** (*ActionType*) – The type of the action.

Return type `Callable`

Returns The `action creator`.

revived - A redux-inspired predictable state container for python

CHAPTER 3

Changelog

0.1.3 - 2017-02-17

- Integrate pip-tools: thanks to Lorenzo Villani for pointing me out to this solution

0.1.2 - 2017-02-16

- Add documentation, tests and test coverage

0.1.1 - 2017-02-13

- Add subscriber decorator in revived.store.Store

0.1.0 - 2017-02-12

- Initial release

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index