ReduxPy

Release r1

Dr. Carsten Leue

Dec 17, 2020

MODULES:

Implementation of a Redux store with support for adding feature modules, dynamically. The store exposes a reactive API based on RxPY.

class redux.**Action**(*type: str*, *payload: Any*)
Action implementation that takes a payload

payload: Any
The action action payload

type: str
Identifier for the action, must be globally unique.

redux.**Epic**
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Callable[[rx.core.observable.observable.Observable, rx.core.observable.observable.Observable], rx.core.observable.observable.Observable]

redux.**Reducer**
The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Callable[[StateType, redux._internal.types.Action], StateType]

class redux.**ReduxFeatureModule**(*id: str*, *reducer: Optional[Callable[[StateType, redux._internal.types.Action], StateType]]*, *epic: Optional[Callable[[rx.core.observable.observable.Observable, rx.core.observable.observable.Observable], rx.core.observable.observable.Observable]]*, *dependencies: Iterable[ReduxFeatureModule]*)
Defines the feature module. The ID identifies the section in the state and is also used to globally discriminate features.

After instantiating a feature store the store will fire an initialization action for that feature. Use `of_init_feature()` to register for these initialization actions.

dependencies: Iterable[redux._internal.types.ReduxFeatureModule]
Dependencies on other feature modules

epic: Optional[Callable[[rx.core.observable.observable.Observable, rx.core.observable
Epic that handles module specific asynchronous operations.

id: str
Identifier of the module, will also be used as a namespace into the state.

reducer: Optional[Callable[[StateType, redux._internal.types.Action], StateType]]
Reducer that handles module specific actions.

class redux.**ReduxRootStore**(*as_observable: Callable[], rx.core.observable.observable.Observable]*, *dispatch: Callable[[redux._internal.types.Action], None]*, *add_feature_module: Callable[[redux._internal.types.ReduxFeatureModule], None]*, *on_next: Callable[[redux._internal.types.Action], None]*, *on_completed: Callable[], None]*)

Implementation of a store that manages sub-state as features. Features are added to the store automatically, when required by the select method.

add_feature_module: Callable[[redux._internal.types.ReduxFeatureModule], None]
Adds a new feature module

as_observable: Callable[], rx.core.observable.observable.Observable]
Converts the store to an observable of state emissions

dispatch: Callable[[redux._internal.types.Action], None]
Dispatches a single action to the store

on_completed: Callable[], None]
Shuts down the store

on_next: Callable[[redux._internal.types.Action], None]
Alias for `dispatch()`

redux.**combine_epics**(**epics*)
Combines a sequence of epics into one single epic by merging them

Parameters **epics** (Iterable[Callable[[Observable, Observable], Observable]]) – the epics to merge

Return type Callable[[Observable, Observable], Observable]

Returns The merged epic

redux.**combine_reducers**(*reducers*)
Creates a new reducer from a mapping of reducers.

Parameters **reducers** (Mapping[str, Callable[[~StateType, Action], ~StateType]]) – the mapping from state partition to reducer

Return type Callable[[Mapping[str, ~StateType], Action], Mapping[str, ~StateType]]

Returns A reducer that dispatches actions against each of the mapped reducers

redux.**create_action**(*type_name*)
Creates a function that produces an action of the given type

Parameters **type_name** (str) – type of the action

Return type Callable[[~PayloadType], Action]

Returns A function that accepts the action payload and creates the action

redux.**create_feature_module**(*identifier*, *reducer=None*, *epic=None*, *dependencies=()*)
Constructs a new feature module descriptor

Parameters

- **identifier** (str) – the identifier of the feature
- **reducer** (Optional[Callable[[~StateType, Action], ~StateType]]) – optional reducer
- **epic** (Optional[Callable[[Observable, Observable], Observable]]) – optional epic
- **dependencies** (Iterable[ReduxFeatureModule]) – optional dependencies on other features

Return type ReduxFeatureModule

Returns The feature module descriptor

redux.**create_store**(*initial_state=None*)
Constructs a new store that can handle feature modules.

Parameters **initial_state** (Optional[Mapping[str, ~StateType]]) – optional initial state of the store, will typically be the empty dict

Return type ReduxRootStore

Returns An implementation of the store

redux.**handle_actions**(*action_map*, *initial_state=None*)
Creates a new reducer from a mapping of action name to reducer for that action.

Parameters

- **action_map** (Mapping[str, Callable[[~StateType, Action], ~StateType]]) – mapping from action name to reducer for that action
- **initial_state** (Optional[~StateType]) – optional initial state used if no reducer matches

Return type Callable[[~StateType, Action], ~StateType]

Returns A reducer function that handles the actions in the map

redux.**of_init_feature**(*identifier*)
Operator to test for the initialization action of a feature

Parameters **identifier** (str) – the identifier of the feature

Return type Callable[[Observable], Observable]

Returns Operator function that accepts init actions for the feature, once

redux.**of_type**(*type_name*)
Returns a reactive operator that filters for actions of the given type

Parameters **type_name** (str) – type of the action to filter for

Return type Callable[[Observable], Observable]

Returns The filter operator function

redux.**select**(*selector*)
Reactive operator that applies a selector and shares the result across multiple subscribers

Parameters **selector** (Callable[[~T1], ~T2]) – the selector function

Return type Callable[[Observable], Observable]

Returns The reactive operator

redux.**select_action_payload**(*action*)
Selects the payload from the action

Parameters **action** (Action) – the action object

Return type ~PayloadType

Returns the payload of the action

redux.**select_feature**(*identifier*, *initial_state=None*)
Returns a function that returns the feature state from the root state

Parameters

- **identifier** (str) – identifier of the feature

- **initial_state** (Optional[~StateType]) – fallback state used if the feature state is not defined

Return type Callable[[Mapping[str, ~StateType]], Optional[~StateType]]

Returns The selector function

Implementation of a Redux store with support for adding feature modules, dynamically. The store exposes a reactive API based on RxPY.

CHAPTER

ONE

WHAT IS REDUX AND WHY

Complex applications - client or server - often need to maintain state and the more complex the application becomes the harder it is to keep track of that state. The Redux pattern addresses the management of complex state by following the ideas of Flux, CQRS, and Event Sourcing.

The basic principle boils down to:

- **Single source of truth:** The state of your whole application is stored in an object tree within a single store.
- **State is read-only:** The only way to change the state is to emit an action, an object describing what happened.
- **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write pure reducers.

1.1 The State Tree

All state is kept in a single, read-only dictionary of type `ReduxRootState`. This state is maintained and managed by the `ReduxRootStore` object that can be created using the `create_store()` method. The store allows to dispatch actions, listen for state changes and add new features.

1.2 Actions

State cannot be changed but we can create new state based on existing state and an action. The action describes how the current state will be transformed.

All state transforms are **synchronous** operations and will be executed by a reducer.

1.3 Reducers

Reducers are pure functions that transform a current state object into a new state object given an action.

1.4 Epics

It is a basic redux principle that all operations that compute new state are executed by synchronous reducers. In order to implement asynchronous operations we introduce the concept of Epics. An epic transforms an action into another action or set of actions and this transform may be executed asynchronously. The resulting actions could in turn give rise to new actions via an epic or they could be interpreted by a reducer.

We represent an epic as a reactive operator that transforms an action input sequence (and optionally also a state sequence) into an action otput sequence.

1.5 Feature Module

There should only be one single redux store instance per application. In traditional redux this means that the set of reducers and epics must be known at instantiation time of the store. This makes it hard to compose the overall application from a set of reusable modules.

We introduce the concept of a feature module, motivated by dynamic modules and feature store.

A feature module defines a unique identifier and optionally a reducer, epic and dependencies. The identifier is used to scope state in a top level dictionary and it is possible to add a new feature module to an existing store at any point in time.

CHAPTER

TWO

PROVIDING A FEATURE MODULE

Create and export an instance of `ReduxFeatureModule` for your module. The module definition consists of:

- a unique module identifier. This identifier is also used as a namespace in the redux state
- an optional reducer that operates on that namespace
- an optional epic to handle asynchronous actions
- an optional list of other feature modules this module depends on

2.1 Example

```
from redux import create_feature_module, ReduxFeatureModule

sample_feature_module: ReduxFeatureModule = create_feature_module(
    'SAMPLE_FEATURE', sample_reducer, sample_epic, [dep1, dep2]
)
```

CHAPTER

THREE

REGISTERING A FEATURE MODULE

Register the feature module with the root store using the `add_feature_module()` method. This will also register all dependent modules in topology order.

```
from redux import create_store, ReduxRootStore

store: ReduxRootStore = create_store()
store.add_feature_module(sampleFeature)
```

CHAPTER

FOUR

CONSUMING A FEATURE MODULE

Use the `select_feature()` method to create a selector for the desired feature.

4.1 Example

```
from redux import select_feature

select_sample = select_feature(sample_feature)
```

CHAPTER

FIVE

SIDE EFFECTS IN FEATURE MODULES

Feature modules may provide side effects, aka epics, for asynchronous processing. Sometimes such epics require an initialization event to execute bootstrapping logic. The store sends an initialization event for this purpose, after a feature module has been initialized. Use the `of_init_feature()` method to subscribe to this event.

5.1 Example

```
from redux import of_init_feature, Epic
from rx.operators import map

initEpic: Epic = lambda actions_, state_: actions_.pipe(of_init_feature(sample_
↪feature), map(...))
```

INDEX