[Skip to content](https://github.com/reduxjs/reselect#start-of-content)

* Why GitHub?
* [Business](https://github.com/business)
* Explore
* [Marketplace](https://github.com/marketplace)
* Pricing

Top of Form



Bottom of Form

[Sign in](https://github.com/login?return_to=%2Freduxjs%2Freselect)[Sign up](https://github.com/join)

* [**Watch**](https://github.com/login?return_to=%2Freduxjs%2Freselect)[**185**](https://github.com/reduxjs/reselect/watchers)
* [**Star**](https://github.com/login?return_to=%2Freduxjs%2Freselect)[**13,449**](https://github.com/reduxjs/reselect/stargazers)
* [**Fork**](https://github.com/login?return_to=%2Freduxjs%2Freselect)[**444**](https://github.com/reduxjs/reselect/network/members)

[reduxjs](https://github.com/reduxjs)/[**reselect**](https://github.com/reduxjs/reselect)

[Code](https://github.com/reduxjs/reselect)[Issues **33**](https://github.com/reduxjs/reselect/issues)[Pull requests **17**](https://github.com/reduxjs/reselect/pulls)[Projects **0**](https://github.com/reduxjs/reselect/projects)[Insights](https://github.com/reduxjs/reselect/pulse)

Top of Form

Dismiss

Bottom of Form

**Join GitHub today**

GitHub is home to over 28 million developers working together to host and review code, manage projects, and build software together.

[**Sign up**](https://github.com/join?source=prompt-code)

Selector library for Redux

[memoized-selectors](https://github.com/topics/memoized-selectors)[redux](https://github.com/topics/redux)

* [**533**commits](https://github.com/reduxjs/reselect/commits/master)
* [**2**branches](https://github.com/reduxjs/reselect/branches)
* [**23**releases](https://github.com/reduxjs/reselect/releases)
* [**76**contributors](https://github.com/reduxjs/reselect/graphs/contributors)
* [MIT](https://github.com/reduxjs/reselect/blob/master/LICENSE)

1. [**JavaScript 54.0%**](https://github.com/reduxjs/reselect/search?l=javascript)
2. [**TypeScript 46.0%**](https://github.com/reduxjs/reselect/search?l=typescript)

JavaScriptTypeScript

Branch: master

New pull request

[**Find file**](https://github.com/reduxjs/reselect/find/master)

Clone or download

[**ellbee**](https://github.com/reduxjs/reselect/commits?author=ellbee) [Update CHANGELOG.md](https://github.com/reduxjs/reselect/commit/c93252ddaa77dce5e2096403c44813b38e8c3aaf)

Latest commit [c93252d](https://github.com/reduxjs/reselect/commit/c93252ddaa77dce5e2096403c44813b38e8c3aaf) on Oct 18

| **Type** | **Name** | **Latest commit message** | **Commit time** |
| --- | --- | --- | --- |
|  | [src](https://github.com/reduxjs/reselect/tree/master/src) | [Merge pull request](https://github.com/reduxjs/reselect/commit/fda697a5b21959d55123bb8a3fdb6192f9cdc984) [#297](https://github.com/reduxjs/reselect/pull/297) [from Mosho1/patch-1](https://github.com/reduxjs/reselect/commit/fda697a5b21959d55123bb8a3fdb6192f9cdc984) | 6 months ago |
|  | [test](https://github.com/reduxjs/reselect/tree/master/test) | [Expose selector dependencies for testing purposes](https://github.com/reduxjs/reselect/commit/fd3238cf5a9c350e02ee118ee1cd3ba7282774ad) | 2 years ago |
|  | [typescript\_test](https://github.com/reduxjs/reselect/tree/master/typescript_test) | [added typings for dynamic array of uniform selectors](https://github.com/reduxjs/reselect/commit/36f256b59b876705144147d409a73a3c4cb3c64d) | 8 months ago |
|  | [.babelrc](https://github.com/reduxjs/reselect/blob/master/.babelrc) | [Add jsnext build](https://github.com/reduxjs/reselect/commit/c9c0bead79198cf7067ecd2f8b36a4a820082995) | 3 years ago |
|  | [.eslintrc](https://github.com/reduxjs/reselect/blob/master/.eslintrc) | [Update dependencies](https://github.com/reduxjs/reselect/commit/efca6405989ab5bb7465d15ceef38b7cd89fd69f) | 3 years ago |
|  | [.gitignore](https://github.com/reduxjs/reselect/blob/master/.gitignore) | [Adds flow types with tests suites](https://github.com/reduxjs/reselect/commit/41965ef75215e23fdbd8c87ccca1128a704c035f) | 2 years ago |
|  | [.travis.yml](https://github.com/reduxjs/reselect/blob/master/.travis.yml) | [Merge branch '3.0.0' of](https://github.com/reduxjs/reselect/commit/28b3f9970a2d2267bc0354e37c4c904ba24fa4ef) [https://github.com/reactjs/reselect](https://github.com/reduxjs/reselect/commit/28b3f9970a2d2267bc0354e37c4c904ba24fa4ef) | 2 years ago |
|  | [AUTHORS](https://github.com/reduxjs/reselect/blob/master/AUTHORS) | [Update AUTHORS for v3.0.0](https://github.com/reduxjs/reselect/commit/d92cfa38466a6975d59b558cf2f826b1acd8f107) | 2 years ago |
|  | [CHANGELOG.md](https://github.com/reduxjs/reselect/blob/master/CHANGELOG.md) | [Update CHANGELOG.md](https://github.com/reduxjs/reselect/commit/c93252ddaa77dce5e2096403c44813b38e8c3aaf) | 2 months ago |
|  | [CREDITS.md](https://github.com/reduxjs/reselect/blob/master/CREDITS.md) | [Update credits for v3.0.0](https://github.com/reduxjs/reselect/commit/5561b9c967eba74041772aa5714bdd5668c81679) | 2 years ago |
|  | [LICENSE](https://github.com/reduxjs/reselect/blob/master/LICENSE) | [[Update] license to current year](https://github.com/reduxjs/reselect/commit/971616fe438c60a075d80ffc558f84925ec81e94) | 10 months ago |
|  | [README.md](https://github.com/reduxjs/reselect/blob/master/README.md) | [Update README.md](https://github.com/reduxjs/reselect/commit/29851a0561a51dafc87b2df9e5472802bad5f1a2) | 6 months ago |
|  | [README\_RU.md](https://github.com/reduxjs/reselect/blob/master/README_RU.md) | [Prepare for move to reduxjs](https://github.com/reduxjs/reselect/commit/1057c5893b300a9ca1fa4c23bea8008827923d4b) | 8 months ago |
|  | [package-lock.json](https://github.com/reduxjs/reselect/blob/master/package-lock.json) | [Update TypeScript](https://github.com/reduxjs/reselect/commit/3932567f7ca88d8dface5b46b0d2e0f23bd8ae07) | 6 months ago |
|  | [package.json](https://github.com/reduxjs/reselect/blob/master/package.json) | [Bump to version 4.0.0](https://github.com/reduxjs/reselect/commit/1f3fdeb22c3fef1b4a212f16b814b5fd9fdf07a9) | 2 months ago |

**README.md**

**Reselect**

Simple “selector” lib6767676767rary for Redux (and others) inspired by getters in [NuclearJS](https://github.com/optimizely/nuclear-js.git), [subscriptions](https://github.com/Day8/re-frame#just-a-read-only-cursor) in [re-frame](https://github.com/Day8/re-frame) and this [proposal](https://github.com/reduxjs/redux/pull/169)from [speedskater](https://github.com/speedskater).

* Selectors can compute derived data, allowing Redux to store the minimal possible state.
* Selectors are efficient. A selector is not recomputed unless one of its arguments changes.
* Selectors are composable. They can be used as input to other selectors.

You can play around with the following **example** in [this codepen](https://codepen.io/Domiii/pen/LzGNWj?editors=0010):

import { createSelector } from 'reselect'

const shopItemsSelector = state => state.shop.items

const taxPercentSelector = state => state.shop.taxPercent

const subtotalSelector = createSelector(

shopItemsSelector,

items => items.reduce((acc, item) => acc + item.value, 0)

)

const taxSelector = createSelector(

subtotalSelector,

taxPercentSelector,

(subtotal, taxPercent) => subtotal \* (taxPercent / 100)

)

export const totalSelector = createSelector(

subtotalSelector,

taxSelector,

(subtotal, tax) => ({ total: subtotal + tax })

)

let exampleState = {

shop: {

taxPercent: 8,

items: [

{ name: 'apple', value: 1.20 },

{ name: 'orange', value: 0.95 },

]

}

}

console.log(subtotalSelector(exampleState)) // 2.15

console.log(taxSelector(exampleState)) // 0.172

console.log(totalSelector(exampleState)) // { total: 2.322 }

**Table of Contents**

* [Installation](https://github.com/reduxjs/reselect#installation)
* [Example](https://github.com/reduxjs/reselect#example)
  + [Motivation for Memoized Selectors](https://github.com/reduxjs/reselect#motivation-for-memoized-selectors)
  + [Creating a Memoized Selector](https://github.com/reduxjs/reselect#creating-a-memoized-selector)
  + [Composing Selectors](https://github.com/reduxjs/reselect#composing-selectors)
  + [Connecting a Selector to the Redux Store](https://github.com/reduxjs/reselect#connecting-a-selector-to-the-redux-store)
  + [Accessing React Props in Selectors](https://github.com/reduxjs/reselect#accessing-react-props-in-selectors)
  + [Sharing Selectors with Props Across Multiple Component Instances](https://github.com/reduxjs/reselect#sharing-selectors-with-props-across-multiple-component-instances)
* [API](https://github.com/reduxjs/reselect#api)
  + [createSelector](https://github.com/reduxjs/reselect#createselectorinputselectors--inputselectors-resultfunc)
  + [defaultMemoize](https://github.com/reduxjs/reselect#defaultmemoizefunc-equalitycheck--defaultequalitycheck)
  + [createSelectorCreator](https://github.com/reduxjs/reselect#createselectorcreatormemoize-memoizeoptions)
  + [createStructuredSelector](https://github.com/reduxjs/reselect#createstructuredselectorinputselectors-selectorcreator--createselector)
* [FAQ](https://github.com/reduxjs/reselect#faq)
  + [Why isn't my selector recomputing when the input state changes?](https://github.com/reduxjs/reselect#q-why-isnt-my-selector-recomputing-when-the-input-state-changes)
  + [Why is my selector recomputing when the input state stays the same?](https://github.com/reduxjs/reselect#q-why-is-my-selector-recomputing-when-the-input-state-stays-the-same)
  + [Can I use Reselect without Redux?](https://github.com/reduxjs/reselect#q-can-i-use-reselect-without-redux)
  + [The default memoization function is no good, can I use a different one?](https://github.com/reduxjs/reselect#q-the-default-memoization-function-is-no-good-can-i-use-a-different-one)
  + [How do I test a selector?](https://github.com/reduxjs/reselect#q-how-do-i-test-a-selector)
  + [How do I create a selector that takes an argument?](https://github.com/reduxjs/reselect#q-how-do-i-create-a-selector-that-takes-an-argument)
  + [How do I use Reselect with Immutable.js?](https://github.com/reduxjs/reselect#q-how-do-i-use-reselect-with-immutablejs)
  + [Can I share a selector across multiple component instances?](https://github.com/reduxjs/reselect#q-can-i-share-a-selector-across-multiple-component-instances)
  + [Are there TypeScript typings?](https://github.com/reduxjs/reselect#q-are-there-typescript-typings)
  + [How can I make a curried selector?](https://github.com/reduxjs/reselect#q-how-can-i-make-a-curried-selector)
* [Related Projects](https://github.com/reduxjs/reselect#related-projects)
* [License](https://github.com/reduxjs/reselect#license)

**Installation**

npm install reselect

**Example**

If you prefer a video tutorial, you can find one [here](https://www.youtube.com/watch?v=6Xwo5mVxDqI).

**Motivation for Memoized Selectors**

The examples in this section are based on the [Redux Todos List example](http://redux.js.org/docs/basics/UsageWithReact.html).

**containers/VisibleTodoList.js**

import { connect } from 'react-redux'

import { toggleTodo } from '../actions'

import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {

switch (filter) {

case 'SHOW\_ALL':

return todos

case 'SHOW\_COMPLETED':

return todos.filter(t => t.completed)

case 'SHOW\_ACTIVE':

return todos.filter(t => !t.completed)

}

}

const mapStateToProps = (state) => {

return {

todos: getVisibleTodos(state.todos, state.visibilityFilter)

}

}

const mapDispatchToProps = (dispatch) => {

return {

onTodoClick: (id) => {

dispatch(toggleTodo(id))

}

}

}

const VisibleTodoList = connect(

mapStateToProps,

mapDispatchToProps

)(TodoList)

export default VisibleTodoList

In the above example, mapStateToProps calls getVisibleTodos to calculate todos. This works great, but there is a drawback: todos is calculated every time the state tree is updated. If the state tree is large, or the calculation expensive, repeating the calculation on every update may cause performance problems. Reselect can help to avoid these unnecessary recalculations.

**Creating a Memoized Selector**

We would like to replace getVisibleTodos with a memoized selector that recalculates todos when the value of state.todosor state.visibilityFilter changes, but not when changes occur in other (unrelated) parts of the state tree.

Reselect provides a function createSelector for creating memoized selectors. createSelector takes an array of input-selectors and a transform function as its arguments. If the Redux state tree is mutated in a way that causes the value of an input-selector to change, the selector will call its transform function with the values of the input-selectors as arguments and return the result. If the values of the input-selectors are the same as the previous call to the selector, it will return the previously computed value instead of calling the transform function.

Let's define a memoized selector named getVisibleTodos to replace the non-memoized version above:

**selectors/index.js**

import { createSelector } from 'reselect'

const getVisibilityFilter = (state) => state.visibilityFilter

const getTodos = (state) => state.todos

export const getVisibleTodos = createSelector(

[ getVisibilityFilter, getTodos ],

(visibilityFilter, todos) => {

switch (visibilityFilter) {

case 'SHOW\_ALL':

return todos

case 'SHOW\_COMPLETED':

return todos.filter(t => t.completed)

case 'SHOW\_ACTIVE':

return todos.filter(t => !t.completed)

}

}

)

In the example above, getVisibilityFilter and getTodos are input-selectors. They are created as ordinary non-memoized selector functions because they do not transform the data they select. getVisibleTodos on the other hand is a memoized selector. It takes getVisibilityFilter and getTodos as input-selectors, and a transform function that calculates the filtered todos list.

**Composing Selectors**

A memoized selector can itself be an input-selector to another memoized selector. Here is getVisibleTodos being used as an input-selector to a selector that further filters the todos by keyword:

const getKeyword = (state) => state.keyword

const getVisibleTodosFilteredByKeyword = createSelector(

[ getVisibleTodos, getKeyword ],

(visibleTodos, keyword) => visibleTodos.filter(

todo => todo.text.includes(keyword)

)

)

**Connecting a Selector to the Redux Store**

If you are using [React Redux](https://github.com/reduxjs/react-redux), you can call selectors as regular functions inside mapStateToProps():

**containers/VisibleTodoList.js**

import { connect } from 'react-redux'

import { toggleTodo } from '../actions'

import TodoList from '../components/TodoList'

import { getVisibleTodos } from '../selectors'

const mapStateToProps = (state) => {

return {

todos: getVisibleTodos(state)

}

}

const mapDispatchToProps = (dispatch) => {

return {

onTodoClick: (id) => {

dispatch(toggleTodo(id))

}

}

}

const VisibleTodoList = connect(

mapStateToProps,

mapDispatchToProps

)(TodoList)

export default VisibleTodoList

**Accessing React Props in Selectors**

This section introduces a hypothetical extension to our app that allows it to support multiple Todo Lists. Please note that a full implementation of this extension requires changes to the reducers, components, actions etc. that aren’t directly relevant to the topics discussed and have been omitted for brevity.

So far we have only seen selectors receive the Redux store state as an argument, but a selector can receive props too.

Here is an App component that renders three VisibleTodoList component instances, each of which has a listId prop:

**components/App.js**

import React from 'react'

import Footer from './Footer'

import AddTodo from '../containers/AddTodo'

import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (

<div>

<VisibleTodoList listId="1" />

<VisibleTodoList listId="2" />

<VisibleTodoList listId="3" />

</div>

)

Each VisibleTodoList container should select a different slice of the state depending on the value of the listId prop, so let’s modify getVisibilityFilter and getTodos to accept a props argument:

**selectors/todoSelectors.js**

import { createSelector } from 'reselect'

const getVisibilityFilter = (state, props) =>

state.todoLists[props.listId].visibilityFilter

const getTodos = (state, props) =>

state.todoLists[props.listId].todos

const getVisibleTodos = createSelector(

[ getVisibilityFilter, getTodos ],

(visibilityFilter, todos) => {

switch (visibilityFilter) {

case 'SHOW\_COMPLETED':

return todos.filter(todo => todo.completed)

case 'SHOW\_ACTIVE':

return todos.filter(todo => !todo.completed)

default:

return todos

}

}

)

export default getVisibleTodos

props can be passed to getVisibleTodos from mapStateToProps:

const mapStateToProps = (state, props) => {

return {

todos: getVisibleTodos(state, props)

}

}

So now getVisibleTodos has access to props, and everything seems to be working fine.

**But there is a problem!**

Using the getVisibleTodos selector with multiple instances of the VisibleTodoList container will not correctly memoize:

**containers/VisibleTodoList.js**

import { connect } from 'react-redux'

import { toggleTodo } from '../actions'

import TodoList from '../components/TodoList'

import { getVisibleTodos } from '../selectors'

const mapStateToProps = (state, props) => {

return {

// WARNING: THE FOLLOWING SELECTOR DOES NOT CORRECTLY MEMOIZE

todos: getVisibleTodos(state, props)

}

}

const mapDispatchToProps = (dispatch) => {

return {

onTodoClick: (id) => {

dispatch(toggleTodo(id))

}

}

}

const VisibleTodoList = connect(

mapStateToProps,

mapDispatchToProps

)(TodoList)

export default VisibleTodoList

A selector created with createSelector has a cache size of 1 and only returns the cached value when its set of arguments is the same as its previous set of arguments. If we alternate between rendering <VisibleTodoList listId="1" /> and <VisibleTodoList listId="2" />, the shared selector will alternate between receiving {listId: 1} and {listId: 2} as its props argument. This will cause the arguments to be different on each call, so the selector will always recompute instead of returning the cached value. We’ll see how to overcome this limitation in the next section.

**Sharing Selectors with Props Across Multiple Component Instances**

The examples in this section require React Redux v4.3.0 or greater

An alternative approach can be found in [re-reselect](https://github.com/toomuchdesign/re-reselect)

To share a selector across multiple VisibleTodoList instances while passing in props **and** retaining memoization, each instance of the component needs its own private copy of the selector.

Let’s create a function named makeGetVisibleTodos that returns a new copy of the getVisibleTodos selector each time it is called:

**selectors/todoSelectors.js**

import { createSelector } from 'reselect'

const getVisibilityFilter = (state, props) =>

state.todoLists[props.listId].visibilityFilter

const getTodos = (state, props) =>

state.todoLists[props.listId].todos

const makeGetVisibleTodos = () => {

return createSelector(

[ getVisibilityFilter, getTodos ],

(visibilityFilter, todos) => {

switch (visibilityFilter) {

case 'SHOW\_COMPLETED':

return todos.filter(todo => todo.completed)

case 'SHOW\_ACTIVE':

return todos.filter(todo => !todo.completed)

default:

return todos

}

}

)

}

export default makeGetVisibleTodos

We also need a way to give each instance of a container access to its own private selector. The mapStateToProps argument of connect can help with this.

**If the mapStateToProps argument supplied to connect returns a function instead of an object, it will be used to create an individual mapStateToProps function for each instance of the container.**

In the example below makeMapStateToProps creates a new getVisibleTodos selector, and returns a mapStateToProps function that has exclusive access to the new selector:

const makeMapStateToProps = () => {

const getVisibleTodos = makeGetVisibleTodos()

const mapStateToProps = (state, props) => {

return {

todos: getVisibleTodos(state, props)

}

}

return mapStateToProps

}

If we pass makeMapStateToProps to connect, each instance of the VisibleTodoList container will get its own mapStateToPropsfunction with a private getVisibleTodos selector. Memoization will now work correctly regardless of the render order of the VisibleTodoList containers.

**containers/VisibleTodoList.js**

import { connect } from 'react-redux'

import { toggleTodo } from '../actions'

import TodoList from '../components/TodoList'

import { makeGetVisibleTodos } from '../selectors'

const makeMapStateToProps = () => {

const getVisibleTodos = makeGetVisibleTodos()

const mapStateToProps = (state, props) => {

return {

todos: getVisibleTodos(state, props)

}

}

return mapStateToProps

}

const mapDispatchToProps = (dispatch) => {

return {

onTodoClick: (id) => {

dispatch(toggleTodo(id))

}

}

}

const VisibleTodoList = connect(

makeMapStateToProps,

mapDispatchToProps

)(TodoList)

export default VisibleTodoList

**API**

**createSelector(...inputSelectors | [inputSelectors], resultFunc)**

Takes one or more selectors, or an array of selectors, computes their values and passes them as arguments to resultFunc.

createSelector determines if the value returned by an input-selector has changed between calls using reference equality (===). Inputs to selectors created with createSelector should be immutable.

Selectors created with createSelector have a cache size of 1. This means they always recalculate when the value of an input-selector changes, as a selector only stores the preceding value of each input-selector.

const mySelector = createSelector(

state => state.values.value1,

state => state.values.value2,

(value1, value2) => value1 + value2

)

// You can also pass an array of selectors

const totalSelector = createSelector(

[

state => state.values.value1,

state => state.values.value2

],

(value1, value2) => value1 + value2

)

It can be useful to access the props of a component from within a selector. When a selector is connected to a component with connect, the component props are passed as the second argument to the selector:

const abSelector = (state, props) => state.a \* props.b

// props only (ignoring state argument)

const cSelector = (\_, props) => props.c

// state only (props argument omitted as not required)

const dSelector = state => state.d

const totalSelector = createSelector(

abSelector,

cSelector,

dSelector,

(ab, c, d) => ({

total: ab + c + d

})

)

**defaultMemoize(func, equalityCheck = defaultEqualityCheck)**

defaultMemoize memoizes the function passed in the func parameter. It is the memoize function used by createSelector.

defaultMemoize has a cache size of 1. This means it always recalculates when the value of an argument changes.

defaultMemoize determines if an argument has changed by calling the equalityCheck function. As defaultMemoize is designed to be used with immutable data, the default equalityCheck function checks for changes using reference equality:

function defaultEqualityCheck(currentVal, previousVal) {

return currentVal === previousVal

}

defaultMemoize can be used with createSelectorCreator to [customize the equalityCheck function](https://github.com/reduxjs/reselect#customize-equalitycheck-for-defaultmemoize).

**createSelectorCreator(memoize, ...memoizeOptions)**

createSelectorCreator can be used to make a customized version of createSelector.

The memoize argument is a memoization function to replace defaultMemoize.

The ...memoizeOptions rest parameters are zero or more configuration options to be passed to memoizeFunc. The selectors resultFunc is passed as the first argument to memoize and the memoizeOptions are passed as the second argument onwards:

const customSelectorCreator = createSelectorCreator(

customMemoize, // function to be used to memoize resultFunc

option1, // option1 will be passed as second argument to customMemoize

option2, // option2 will be passed as third argument to customMemoize

option3 // option3 will be passed as fourth argument to customMemoize

)

const customSelector = customSelectorCreator(

input1,

input2,

resultFunc // resultFunc will be passed as first argument to customMemoize

)

Internally customSelector calls the memoize function as follows:

customMemoize(resultFunc, option1, option2, option3)

Here are some examples of how you might use createSelectorCreator:

**Customize equalityCheck for defaultMemoize**

import { createSelectorCreator, defaultMemoize } from 'reselect'

import isEqual from 'lodash.isEqual'

// create a "selector creator" that uses lodash.isEqual instead of ===

const createDeepEqualSelector = createSelectorCreator(

defaultMemoize,

isEqual

)

// use the new "selector creator" to create a selector

const mySelector = createDeepEqualSelector(

state => state.values.filter(val => val < 5),

values => values.reduce((acc, val) => acc + val, 0)

)

**Use memoize function from lodash for an unbounded cache**

import { createSelectorCreator } from 'reselect'

import memoize from 'lodash.memoize'

let called = 0

const hashFn = (...args) => args.reduce(

(acc, val) => acc + '-' + JSON.stringify(val),

''

)

const customSelectorCreator = createSelectorCreator(memoize, hashFn)

const selector = customSelectorCreator(

state => state.a,

state => state.b,

(a, b) => {

called++

return a + b

}

)

**createStructuredSelector({inputSelectors}, selectorCreator = createSelector)**

createStructuredSelector is a convenience function for a common pattern that arises when using Reselect. The selector passed to a connect decorator often just takes the values of its input-selectors and maps them to keys in an object:

const mySelectorA = state => state.a

const mySelectorB = state => state.b

// The result function in the following selector

// is simply building an object from the input selectors

const structuredSelector = createSelector(

mySelectorA,

mySelectorB,

mySelectorC,

(a, b, c) => ({

a,

b,

c

})

)

createStructuredSelector takes an object whose properties are input-selectors and returns a structured selector. The structured selector returns an object with the same keys as the inputSelectors argument, but with the selectors replaced with their values.

const mySelectorA = state => state.a

const mySelectorB = state => state.b

const structuredSelector = createStructuredSelector({

x: mySelectorA,

y: mySelectorB

})

const result = structuredSelector({ a: 1, b: 2 }) // will produce { x: 1, y: 2 }

Structured selectors can be nested:

const nestedSelector = createStructuredSelector({

subA: createStructuredSelector({

selectorA,

selectorB

}),

subB: createStructuredSelector({

selectorC,

selectorD

})

})

**FAQ**

**Q: Why isn’t my selector recomputing when the input state changes?**

A: Check that your memoization function is compatible with your state update function (i.e. the reducer if you are using Redux). For example, a selector created with createSelector will not work with a state update function that mutates an existing object instead of creating a new one each time. createSelector uses an identity check (===) to detect that an input has changed, so mutating an existing object will not trigger the selector to recompute because mutating an object does not change its identity. Note that if you are using Redux, mutating the state object is [almost certainly a mistake](http://redux.js.org/docs/Troubleshooting.html).

The following example defines a simple selector that determines if the first todo item in an array of todos has been completed:

const isFirstTodoCompleteSelector = createSelector(

state => state.todos[0],

todo => todo && todo.completed

)

The following state update function **will not** work with isFirstTodoCompleteSelector:

export default function todos(state = initialState, action) {

switch (action.type) {

case COMPLETE\_ALL:

const areAllMarked = state.every(todo => todo.completed)

// BAD: mutating an existing object

return state.map(todo => {

todo.completed = !areAllMarked

return todo

})

default:

return state

}

}

The following state update function **will** work with isFirstTodoCompleteSelector:

export default function todos(state = initialState, action) {

switch (action.type) {

case COMPLETE\_ALL:

const areAllMarked = state.every(todo => todo.completed)

// GOOD: returning a new object each time with Object.assign

return state.map(todo => Object.assign({}, todo, {

completed: !areAllMarked

}))

default:

return state

}

}

If you are not using Redux and have a requirement to work with mutable data, you can use createSelectorCreator to replace the default memoization function and/or use a different equality check function. See [here](https://github.com/reduxjs/reselect#use-memoize-function-from-lodash-for-an-unbounded-cache) and [here](https://github.com/reduxjs/reselect#customize-equalitycheck-for-defaultmemoize) for examples.

**Q: Why is my selector recomputing when the input state stays the same?**

A: Check that your memoization function is compatible with your state update function (i.e. the reducer if you are using Redux). For example, a selector created with createSelector that recomputes unexpectedly may be receiving a new object on each update whether the values it contains have changed or not. createSelector uses an identity check (===) to detect that an input has changed, so returning a new object on each update means that the selector will recompute on each update.

import { REMOVE\_OLD } from '../constants/ActionTypes'

const initialState = [

{

text: 'Use Redux',

completed: false,

id: 0,

timestamp: Date.now()

}

]

export default function todos(state = initialState, action) {

switch (action.type) {

case REMOVE\_OLD:

return state.filter(todo => {

return todo.timestamp + 30 \* 24 \* 60 \* 60 \* 1000 > Date.now()

})

default:

return state

}

}

The following selector is going to recompute every time REMOVE\_OLD is invoked because Array.filter always returns a new object. However, in the majority of cases the REMOVE\_OLD action will not change the list of todos so the recomputation is unnecessary.

import { createSelector } from 'reselect'

const todosSelector = state => state.todos

export const visibleTodosSelector = createSelector(

todosSelector,

(todos) => {

...

}

)

You can eliminate unnecessary recomputations by returning a new object from the state update function only when a deep equality check has found that the list of todos has actually changed:

import { REMOVE\_OLD } from '../constants/ActionTypes'

import isEqual from 'lodash.isEqual'

const initialState = [

{

text: 'Use Redux',

completed: false,

id: 0,

timestamp: Date.now()

}

]

export default function todos(state = initialState, action) {

switch (action.type) {

case REMOVE\_OLD:

const updatedState = state.filter(todo => {

return todo.timestamp + 30 \* 24 \* 60 \* 60 \* 1000 > Date.now()

})

return isEqual(updatedState, state) ? state : updatedState

default:

return state

}

}

Alternatively, the default equalityCheck function in the selector can be replaced by a deep equality check:

import { createSelectorCreator, defaultMemoize } from 'reselect'

import isEqual from 'lodash.isEqual'

const todosSelector = state => state.todos

// create a "selector creator" that uses lodash.isEqual instead of ===

const createDeepEqualSelector = createSelectorCreator(

defaultMemoize,

isEqual

)

// use the new "selector creator" to create a selector

const mySelector = createDeepEqualSelector(

todosSelector,

(todos) => {

...

}

)

Always check that the cost of an alternative equalityCheck function or deep equality check in the state update function is not greater than the cost of recomputing every time. If recomputing every time does work out to be the cheaper option, it may be that for this case Reselect is not giving you any benefit over passing a plain mapStateToProps function to connect.

**Q: Can I use Reselect without Redux?**

A: Yes. Reselect has no dependencies on any other package, so although it was designed to be used with Redux it can be used independently. It is currently being used successfully in traditional Flux apps.

If you create selectors using createSelector make sure its arguments are immutable. See [here](https://github.com/reduxjs/reselect#createselectorinputselectors--inputselectors-resultfunc)

**Q: How do I create a selector that takes an argument?**

A: Keep in mind that selectors can access React props, so if your arguments are (or can be made available as) React props, you can use that functionality. [See here](https://github.com/reduxjs/reselect#accessing-react-props-in-selectors) for details.

Otherwise, Reselect doesn't have built-in support for creating selectors that accepts arguments, but here are some suggestions for implementing similar functionality...

If the argument is not dynamic you can use a factory function:

const expensiveItemSelectorFactory = minValue => {

return createSelector(

shopItemsSelector,

items => items.filter(item => item.value > minValue)

)

}

const subtotalSelector = createSelector(

expensiveItemSelectorFactory(200),

items => items.reduce((acc, item) => acc + item.value, 0)

)

The general consensus [here](https://github.com/reduxjs/reselect/issues/38) and [over at nuclear-js](https://github.com/optimizely/nuclear-js/issues/14) is that if a selector needs a dynamic argument, then that argument should probably be state in the store. If you decide that you do require a selector with a dynamic argument, then a selector that returns a memoized function may be suitable:

import { createSelector } from 'reselect'

import memoize from 'lodash.memoize'

const expensiveSelector = createSelector(

state => state.items,

items => memoize(

minValue => items.filter(item => item.value > minValue)

)

)

const expensiveFilter = expensiveSelector(state)

const slightlyExpensive = expensiveFilter(100)

const veryExpensive = expensiveFilter(1000000)

**Q: The default memoization function is no good, can I use a different one?**

A: We think it works great for a lot of use cases, but sure. See [these examples](https://github.com/reduxjs/reselect#customize-equalitycheck-for-defaultmemoize).

**Q: How do I test a selector?**

A: For a given input, a selector should always produce the same output. For this reason they are simple to unit test.

const selector = createSelector(

state => state.a,

state => state.b,

(a, b) => ({

c: a \* 2,

d: b \* 3

})

)

test("selector unit test", () => {

assert.deepEqual(selector({ a: 1, b: 2 }), { c: 2, d: 6 })

assert.deepEqual(selector({ a: 2, b: 3 }), { c: 4, d: 9 })

})

It may also be useful to check that the memoization function for a selector works correctly with the state update function (i.e. the reducer if you are using Redux). Each selector has a recomputations method that will return the number of times it has been recomputed:

suite('selector', () => {

let state = { a: 1, b: 2 }

const reducer = (state, action) => (

{

a: action(state.a),

b: action(state.b)

}

)

const selector = createSelector(

state => state.a,

state => state.b,

(a, b) => ({

c: a \* 2,

d: b \* 3

})

)

const plusOne = x => x + 1

const id = x => x

test("selector unit test", () => {

state = reducer(state, plusOne)

assert.deepEqual(selector(state), { c: 4, d: 9 })

state = reducer(state, id)

assert.deepEqual(selector(state), { c: 4, d: 9 })

assert.equal(selector.recomputations(), 1)

state = reducer(state, plusOne)

assert.deepEqual(selector(state), { c: 6, d: 12 })

assert.equal(selector.recomputations(), 2)

})

})

Additionally, selectors keep a reference to the last result function as .resultFunc. If you have selectors composed of many other selectors this can help you test each selector without coupling all of your tests to the shape of your state.

For example if you have a set of selectors like this:

**selectors.js**

export const firstSelector = createSelector( ... )

export const secondSelector = createSelector( ... )

export const thirdSelector = createSelector( ... )

export const myComposedSelector = createSelector(

firstSelector,

secondSelector,

thirdSelector,

(first, second, third) => first \* second < third

)

And then a set of unit tests like this:

**test/selectors.js**

// tests for the first three selectors...

test("firstSelector unit test", () => { ... })

test("secondSelector unit test", () => { ... })

test("thirdSelector unit test", () => { ... })

// We have already tested the previous

// three selector outputs so we can just call `.resultFunc`

// with the values we want to test directly:

test("myComposedSelector unit test", () => {

// here instead of calling selector()

// we just call selector.resultFunc()

assert(myComposedSelector.resultFunc(1, 2, 3), true)

assert(myComposedSelector.resultFunc(2, 2, 1), false)

})

Finally, each selector has a resetRecomputations method that sets recomputations back to 0. The intended use is for a complex selector that may have many independent tests and you don't want to manually manage the computation count or create a "dummy" selector for each test.

**Q: How do I use Reselect with Immutable.js?**

A: Selectors created with createSelector should work just fine with Immutable.js data structures.

If your selector is recomputing and you don't think the state has changed, make sure you are aware of which Immutable.js update methods **always** return a new object and which update methods only return a new object **when the collection actually changes**.

import Immutable from 'immutable'

let myMap = Immutable.Map({

a: 1,

b: 2,

c: 3

})

// set, merge and others only return a new obj when update changes collection

let newMap = myMap.set('a', 1)

assert.equal(myMap, newMap)

newMap = myMap.merge({ 'a': 1 })

assert.equal(myMap, newMap)

// map, reduce, filter and others always return a new obj

newMap = myMap.map(a => a \* 1)

assert.notEqual(myMap, newMap)

If a selector's input is updated by an operation that always returns a new object, it may be performing unnecessary recomputations. See [here](https://github.com/reduxjs/reselect#q-why-is-my-selector-recomputing-when-the-input-state-stays-the-same) for a discussion on the pros and cons of using a deep equality check like Immutable.is to eliminate unnecessary recomputations.

**Q: Can I share a selector across multiple component instances?**

A: Selectors created using createSelector only have a cache size of one. This can make them unsuitable for sharing across multiple instances if the arguments to the selector are different for each instance of the component. There are a couple of ways to get around this:

* Create a factory function which returns a new selector for each instance of the component. There is built-in support for factory functions in React Redux v4.3 or higher. See [here](https://github.com/reduxjs/reselect#sharing-selectors-with-props-across-multiple-component-instances) for an example.
* Create a custom selector with a cache size greater than one.

**Q: Are there TypeScript Typings?**

A: Yes! They are included and referenced in package.json. They should Just Work™.

**Q: How can I make a**[**curried**](https://github.com/hemanth/functional-programming-jargon#currying)**selector?**

A: Try these [helper functions](https://github.com/reduxjs/reselect/issues/159#issuecomment-238724788) courtesy of [MattSPalmer](https://github.com/MattSPalmer)

**Related Projects**

[**re-reselect**](https://github.com/toomuchdesign/re-reselect)

Enhances Reselect selectors by wrapping createSelector and returning a memoized collection of selectors indexed with the cache key returned by a custom resolver function.

Useful to reduce selectors recalculation when the same selector is repeatedly called with one/few different arguments.

[**reselect-tools**](https://github.com/skortchmark9/reselect-tools)

[Chrome extension](https://chrome.google.com/webstore/detail/reselect-devtools/cjmaipngmabglflfeepmdiffcijhjlbb?hl=en) and [companion lib](https://github.com/skortchmark9/reselect-tools) for debugging selectors.

* Measure selector recomputations across the app and identify performance bottlenecks
* Check selector dependencies, inputs, outputs, and recomputations at any time with the chrome extension
* Statically export a JSON representation of your selector graph for further analysis

[**reselect-map**](https://github.com/HeyImAlex/reselect-map)

Can be useful when doing **very expensive** computations on elements of a collection because Reselect might not give you the granularity of caching that you need. Check out the reselect-maps README for examples.

**The optimizations in reselect-map only apply in a small number of cases. If you are unsure whether you need it, you don't!**

**License**

MIT

* © 2018 GitHub, Inc.
* [Terms](https://github.com/site/terms)
* [Privacy](https://github.com/site/privacy)
* [Security](https://github.com/security)
* [Status](https://status.github.com/)
* [Help](https://help.github.com/)
* [Contact GitHub](https://github.com/contact)
* [Pricing](https://github.com/pricing)
* [API](https://developer.github.com/)
* [Training](https://training.github.com/)
* [Blog](https://blog.github.com/)
* [About](https://github.com/about)

Press h to open a hovercard with more details.