

---

## Autodux

coverage 95%

Redux on autopilot.

Brought to you by EricElliottJS.com and DevAnywhere.io.

### Install

```
1 npm install --save autodux
```

And then in your file:

```
1 import autodux from 'autodux';
```

Or using CommonJS syntax:

```
1 const autodux = require('autodux');
```

### Redux on Autopilot

Autodux lets you create reducers like this:

```
1 export const {
2   actions: {
3     setUser, setUsername, setAvatar
4   },
5   selectors: {
6     getUser, getUsername, getAvatar
7   }
8 } = autodux({
9   slice: 'user',
10  initial: {
11    username: 'Anonymous',
12    avatar: 'anon.png'
13  }
14 });
```

That creates a full set of action creators, selectors, reducers, and action type constants – everything you need for fully functional Redux state management.

Everything's on autopilot – but you can override everything when you need to.

---

## Why

Redux is great, but you have to make a lot of boilerplate:

- Action type constants
- Action creators
- Reducers
- Selectors

It's **great** that Redux is such a low-level tool. It's allowed a lot of flexibility and enabled the community to experiment with best practices and patterns.

It's **terrible** that Redux is such a low-level tool. It turns out that:

- Most reducers spend most of their logic switching over action types.
- Most of the actual state updates could be replaced with generic tools like “concat payload to an array”, “remove object from array by some prop”, or “increment a number”. Better to reuse utilities than to implement these things from scratch every time.
- Lots of action creators don't need arguments or payloads – just action types.
- Action types can be automatically generated by combining the name of the state slice with the name of the action, e.g., `counter/increment`.
- Lots of selectors just need to grab the state slice.

Lots of Redux beginners separate all these things into separate files, meaning you have to open and import a whole bunch of files just to get some simple state management in your app.

What if you could write some simple, declarative code that would automatically create your:

- Action type constants
- Reducer switching logic
- State slice selectors
- Action object shape - automatically inserting the correct action type so all you have to worry about is the payload
- Action creators if the input is the payload or no payload is required, i.e., `x => ({ type: 'foo', payload: x })`
- Action reducers if the state can be assigned from the action payload (i.e., `{ ...state, ...payload }`)

Turns out, when you add this simple logic on top of Redux, you can do a lot more with a lot less code.

```
1 import autodux, { id } from 'autodux';
2
3 export const {
```

---

```

 4   reducer,
 5   initial,
 6   slice,
 7   actions: {
 8     increment,
 9     decrement,
10     multiply
11   },
12   selectors: {
13     getValue
14   }
15 } = autodux({
16   // the slice of state your reducer controls
17   slice: 'counter',
18
19   // The initial value of your reducer state
20   initial: 0,
21
22   // No need to implement switching logic -- it's
23   // done for you.
24   actions: {
25     increment: state => state + 1,
26     decrement: state => state - 1,
27     multiply: (state, payload) => state * payload
28   },
29
30   // No need to select the state slice -- it's done for you.
31   selectors: {
32     getValue: id
33   }
34 });
```

As you can see, you can destructure and export the return value directly where you call `autodux()` to reduce boilerplate to a minimum. It returns an object that looks like this:

```

1  {
2    initial: 0,
3    actions: {
4      increment: { [Function]
5        type: 'counter/increment'
6      },
7      decrement: { [Function]
8        type: 'counter/decrement'
9      },
10     multiply: { [Function]
11       type: 'counter/multiply'
12     }
13   },
14   selectors: {
15     getValue: [Function: wrapper]
16   },
```

---

```
17   reducer: [Function: reducer],
18   slice: 'counter'
19 }
```

Let's explore that object a bit:

```
1  const actions = [
2    increment(),
3    increment(),
4    increment(),
5    decrement()
6  ];
7
8  const state = actions.reduce(reducer, initial);
9
10 console.log(getValue({ counter: state })); // 2
11 console.log(increment.type); // 'counter/increment'
```

## API Differences

Action creators, reducers and selectors have simplified APIs.

### Automate (Almost) Everything with Defaults

With `autodux`, you can omit (almost) everything.

**Default Actions** An action is an action creator/reducer pair. Usually, these line up in neat 1:1 mappings. It turns out, you can do a lot with some simple defaults:

- A `set${slice}` action lets you set any key in your reducer – basically: `{...state, ...payload}`. If your `slice` is called `user`, the `set` creator will be called `setUser`.
- Actions for `set{key}` will exist for each key in your `initial` state. If you have a key called `userName`, you'll have an action called `setUserName` created automatically.

**Default Selectors** Like action creators and reducers, selectors are automatically created for each key in your initial state. `get{key}` will exist for each key in your `initial` state., and `get{slice}` will exist for the entire reducer state.

For simple reducers, all the action creators, reducers, and selectors can be created for you automatically. All you have to do is specify the initial state shape and export the bindings.

---

## Action Creators

Action creators are optional! If you need to set a username, you might normally create an action creator like this:

```
1 const setUsername = userName => ({
2   type: 'userReducer/setUserName',
3   payload: userName
4 })
```

With autodux, if your action creator maps directly from input to payload, you can omit it. autodux will do it for you.

By omitting the action creator, you can shorten this:

```
1 actions: {
2   multiply: {
3     create: by => by,
4     reducer: (state, payload) => state * payload
5   }
6 }
```

To this:

```
1 actions: {
2   multiply: (state, payload) => state * payload
3 }
```

**No need to set the type** You don't need to worry about setting the type in autodux action creators. That's handled for you automatically. In other words, all an action creator has to do is return the payload.

With Redux alone you might write:

```
1 const setUsername = userName => ({
2   type: 'userReducer/setUserName',
3   payload: userName
4 })
```

With autodux, that becomes:

```
1 userName => userName
```

Since that's the default behavior, you can omit that one entirely.

You don't need to create action creators unless you need to map the inputs to a different payload output. For example, if you need to translate between an auth provider user and your own application user objects, you could use an action creator like this:

---

```
1 ({ userId, displayName }) => ({ uid: userId, userName: displayName })
```

Here's how you'd implement our multiply action if you want to use a named parameter for the multiplication factor:

```
1 //...
2 actions: {
3   multiply: {
4     // Destructure the named parameter, and return it
5     // as the action payload:
6     create: ({ by }) => by,
7     reducer: (state, payload) => state * payload
8   }
9 }
10 //...
```

## Reducers

Note: Reducers are optional. If your reducer would just assign the payload props to the state (`{...state, ...payload}`), you're already done.

**No switch required** Switching over different action types is automatic, so we don't need an action object that isolates the action type and payload. Instead, we pass the action payload directly, e.g:

With Redux:

```
1 const INCREMENT = 'INCREMENT';
2 const DECREMENT = 'DECREMENT';
3 const MULTIPLY = 'MULTIPLY';
4
5 const counter = (state = 0, action = {}) {
6   switch (action.type){
7     case INCREMENT: return state + 1;
8     case DECREMENT: return state - 1;
9     case MULTIPLY : return state * action.payload
10    default: return state;
11  }
12 };
```

With Autodux, action type handlers are switched over automatically. No more switching logic or manual juggling with action type constants.

```
1 const counter = autodux({
2   slice: 'counter',
3   initial: 0,
4   actions: {
```

---

```
5   increment: state => state + 1,
6   decrement: state => state - 1,
7   multiply: (state, payload) => state * payload
8 }
9 });
```

Autodux infers action types for you automatically using the slice and the action name, and eliminates the need to write switching logic or worry about (or forget) the default case.

Because the switching is handled automatically, your reducers don't need to worry about the action type. Instead, they're passed the payload directly.

## Selectors

Note: Selectors are optional. By default, every key in your initial state will have its own selector, prepended with `get` and camelCased. For example, if you have a key called `userName`, a `getUserName` selector will be created automatically.

Selectors are designed to take the application's complete root state object, but the slice you care about is automatically selected for you, so you can write your selectors as if you're only dealing with the local reducer.

This has some implications with unit tests. The following selector will just return the local reducer state (*Note: You'll automatically get a default selector that does the same thing, so you don't ever need to do this yourself*):

```
1 import { autodux, id } from 'autodux';
2
3 const counter = autodux({
4   // stuff here
5   selectors: {
6     getValue: id // state => state
7   },
8   // other stuff
```

In your unit tests, you'll need to pass the key for the state slice to mock the global store state:

```
1 test('counter.getValue', assert => {
2   const msg = 'should return the current count';
3   const { getValue } = counter.selectors;
4
5   const actual = getValue({ counter: 3 });
6   const expected = 3;
7
8   assert.same(actual, expected, msg);
9   assert.end();
10 });
```

---

Although you should avoid selecting state from outside the slice you care about, the root state object is passed as a convenience second argument to selectors:

```
1 import autodux from 'autodux';
2
3 const counter = autodux({
4   // stuff here
5   selectors: {
6     // other selectors
7     rootState: (_, root) => root
8   },
9   // other stuff
```

In your unit tests, this allows you to retrieve the entire root state:

```
1 test('counter.rootState', assert => {
2   const msg = 'should return the root state';
3   const { rootState } = counter.selectors;
4
5   const actual = rootState({ counter: 3, otherSlice: 'data' });
6   const expected = { counter: 3, otherSlice: 'data' };
7
8   assert.same(actual, expected, msg);
9   assert.end();
10 });
```

## Extras

### **assign = (key: String) => reducer: Function**

Often, we want our reducers to simply set a key in the state to the payload value. `assign()` makes that easy. e.g.:

```
1 const {
2   actions: {
3     setUsername,
4     setAvatar
5   },
6   reducer
7 } = autodux({
8   slice: 'user',
9   initial: {
10     userName: 'Anonymous',
11     avatar: 'anonymous.png'
12   },
13   actions: {
14     setUsername: assign('userName'),
15     setAvatar: assign('avatar')
```



---

```
16   }
17   });
18
19   const userName = 'Foo';
20   const avatar = 'foo.png';
21
22   const state = [
23     setUsername(userName),
24     setAvatar(avatar)
25   ].reduce(reducer, undefined);
26   // => { userName: 'Foo', avatar: 'foo.png' }
```

Since that's the default behavior when actions are omitted, you can also shorten that to:

```
1  const {
2    actions: {
3      setUsername,
4      setAvatar
5    },
6    reducer
7  } = autodox({
8    slice: 'user',
9    initial: {
10      userName: 'Anonymous',
11      avatar: 'anonymous.png'
12    }
13  });
```

**id = x => x**

Useful for selectors that simply return the slice state:

```
1  selectors: {
2    getValue: id
3  }
```