

Redux Toolkit: useSelector vs createSelector

This document explains how `useSelector` and `createSelector` work in Redux Toolkit, with examples, flow, and best practices.

1. useSelector

`useSelector` is a React-Redux hook that allows components to read **state from the Redux store**.

Syntax:

```
const value = useSelector(selectorFunction);
```

- `selectorFunction` receives the **entire state** as its argument
- Returns the selected slice of state
- The component re-renders **only if the returned value changes**

Example:

```
import { useSelector } from 'react-redux';

function UserProfile() {
  const user = useSelector(state => state.user.user);
  const loading = useSelector(state => state.user.loading);

  return (
    <div>
      {loading ? 'Loading...' : user?.name}
    </div>
  );
}
```

2. Input selectors

Input selectors are **functions that take state and return a slice of it**.

Example:

```
const selectUsers = state => state.users.list;
const selectSearchTerm = state => state.users.searchTerm;
```

- They are called “input selectors” when used with `createSelector` because their **outputs are fed as inputs to the result function**.
 - You can also call them **state selectors** since they return a slice of state.
-

3. `createSelector`

`createSelector` comes from `reselect` (built into Redux Toolkit) and **creates a memoized selector function**.

Syntax:

```
const memoizedSelector = createSelector(
  [inputSelector1, inputSelector2, ...], // array of input selectors
  (arg1, arg2, ...) => {           // result function
    return derivedValue;
}
);
```

- **Input selectors** are functions that extract parts of the state
- **Result function** receives the outputs of the input selectors as arguments
- The **memoized function** only recomputes if the inputs change
- Returns the **derived value**

Example:

```
const selectFilteredUsers = createSelector(
  [selectUsers, selectSearchTerm],
  (users, term) => users.filter(u => u.name.includes(term))
);
```

- `users` = output of `selectUsers(state)`
 - `term` = output of `selectSearchTerm(state)`
 - The function returns the filtered array
-

4. Using createSelector with useSelector

```
function UserList() {
  const filteredUsers = useSelector(selectFilteredUsers); // calls the function
  internally

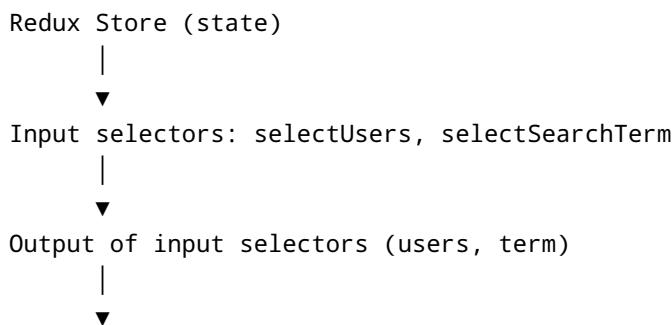
  return (
    <ul>
      {filteredUsers.map(u => (
        <li key={u.id}>{u.name}</li>
      ))}
    </ul>
  );
}
```

- `useSelector(selectFilteredUsers)` automatically passes the **current state** to the selector function
- The **result (filtered array)** is stored in `filteredUsers`
- You can now use array methods like `.map()`
- Memoization ensures that if `users` or `searchTerm` didn't change, the **result is cached**

5. Key points

1. `useSelector` reads Redux state in components
2. `createSelector` creates memoized derived data from state
3. Input selectors provide the **inputs** to the result function
4. The result function only sees the **outputs of input selectors**, not the full state
5. The memoized selector returns a **value** (array, object, etc.), which you can use in your component
6. Parameter names in the result function are arbitrary but should be descriptive
7. Memoization improves performance, especially for expensive calculations

6. Flow Summary



```
Result function: (users, term) => derived value (filteredUsers)
  |
  ▼
Returned from createSelector → memoized
  |
  ▼
useSelector calls selector internally → returns filteredUsers
  |
  ▼
Component renders using filteredUsers.map()
```

7. Best Practices

- Always define input selectors separately if you reuse them
- Keep parameter names in the result function descriptive (`users` , `term`)
- Use `createSelector` for **derived or computed data** to avoid unnecessary recalculations
- Use `useSelector` inside components to **trigger re-render only when the selected value changes**
- Input selectors can also be called “state selectors” — terminology depends on context