# What is Redux?

It helps to understand what this "Redux" thing is in the first place. What does it do? What problems does it help me solve? Why would I want to use it?

**Redux is a pattern and library for managing and updating application state, using events called "actions"**. It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

## Why Should I Use Redux?

Redux helps you manage "global" state - state that is needed across many parts of your application.

**The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur**. Redux guides you towards writing code that is predictable and testable, which helps give you confidence that your application will work as expected.

## When Should I Use Redux?

Redux helps you deal with shared state management, but like any tool, it has tradeoffs. There's more concepts to learn, and more code to write. It also adds some indirection to your code, and asks you to follow certain restrictions. It's a trade-off between short term and long term productivity.

Redux is more useful when:

- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex
- The app has a medium or large-sized codebase, and might be worked on by many people

**Not all apps need Redux. Take some time to think about the kind of app you're building, and decide what tools would be best to help solve the problems you're working on.**

# Redux Terms and Concepts

Before we dive into some actual code, let's talk about some of the terms and concepts you'll need to know to use Redux.

## State Management

Let's start by looking at a small React counter component. It tracks a number in component state, and increments the number when a button is clicked:

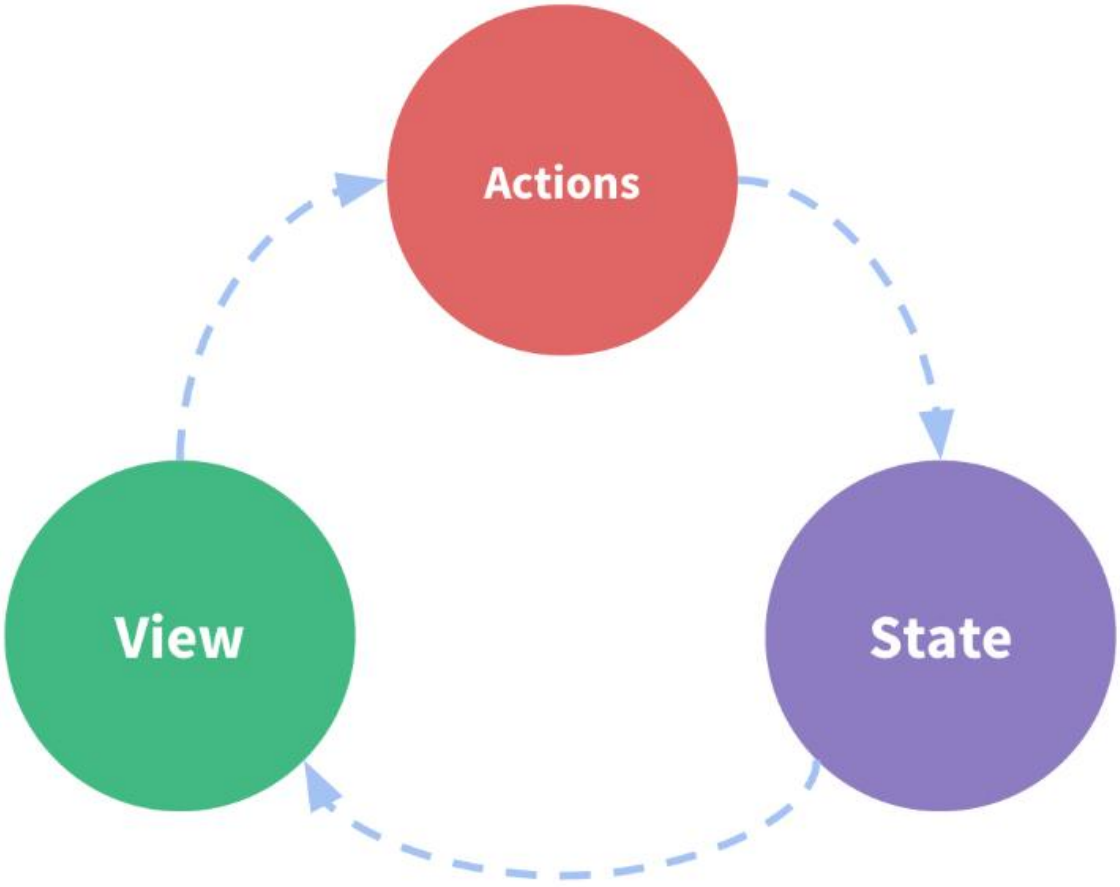
```
function Counter() {
  // State: a counter value
  const [counter, setCounter] = useState(0)

  // Action: code that causes an update to the state when something happens
  const increment = () => {
    setCounter(prevCounter => prevCounter + 1)
  }

  // View: the UI definition
  return (
    <div>
      Value: {counter} <button onClick={increment}>Increment</button>
    </div>
  )
}
```

It is a self-contained app with the following parts:

- The **state**, the source of truth that drives our app;
- The **view**, a declarative description of the UI based on the current state
- The **actions**, the events that occur in the app based on user input, and trigger updates in the state

This is a small example of **"one-way data flow"**:

- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state



## Immutability

"Mutable" means "changeable". If something is "immutable", it can never be changed.

JavaScript objects and arrays are all mutable by default. If I create an object, I can change the contents of its fields. If I create an array, I can change the contents as well:

```
const obj = { a: 1, b: 2 }
// still the same object outside, but the contents have changed
obj.b = 3

const arr = ['a', 'b']
// In the same way, we can change the contents of this array
arr.push('c')
arr[1] = 'd'
```

This is called *mutating* the object or array. It's the same object or array reference in memory, but now the contents inside the object have changed.

**In order to update values immutably, your code must make *copies* of existing objects/arrays, and then modify the copies**.

We can do this by hand using JavaScript's array / object spread operators, as well as array methods that return new copies of the array instead of mutating the original array:

```
const obj = {
  a: {
    // To safely update obj.a.c, we have to copy each piece
    c: 3
  },
  b: 2
}

const obj2 = {
  // copy obj
  ...obj,
  // overwrite a
  a: {
    // copy obj.a
    ...obj.a,
    // overwrite c
    c: 42
  }
}

const arr = ['a', 'b']
// Create a new copy of arr, with "c" appended to the end
const arr2 = arr.concat('c')

// or, we can make a copy of the original array:
const arr3 = arr.slice()
// and mutate the copy:
arr3.push('c')
```

**Redux expects that all state updates are done immutably**. We'll look at where and how this is important a bit later, as well as some easier ways to write immutable update logic.

## Redux Application Data Flow

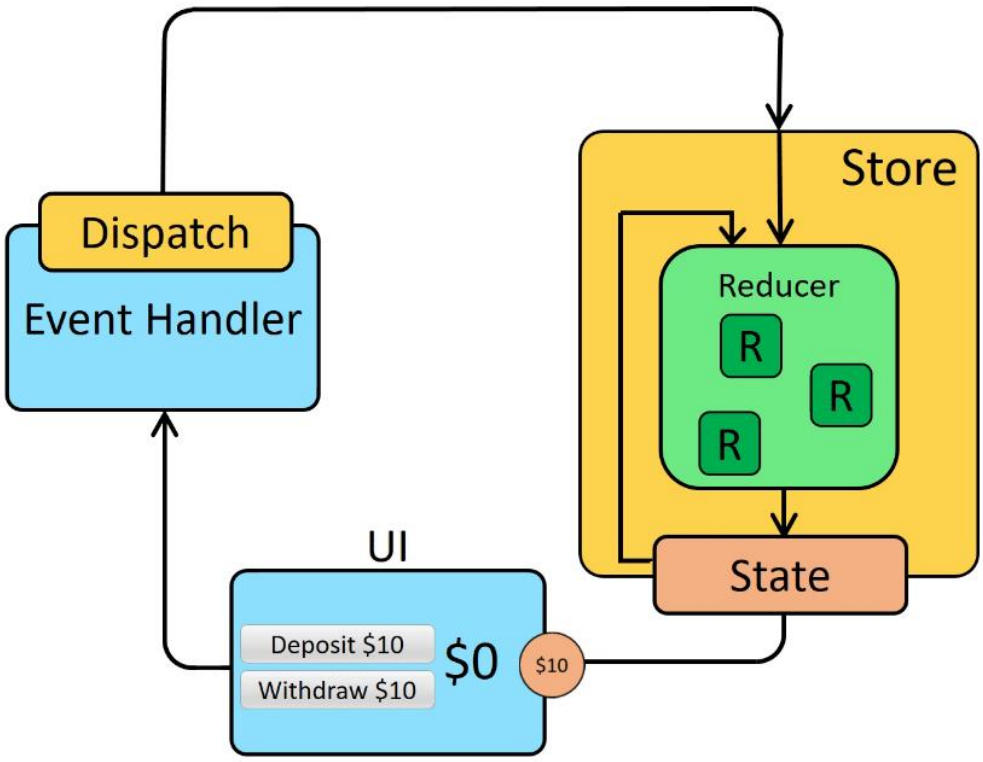
Earlier, we talked about "one-way data flow", which describes this sequence of steps to update the app:

- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state

For Redux specifically, we can break these steps into more detail:

- Initial setup:
  - A Redux store is created using a root reducer function
  - The store calls the root reducer once, and saves the return value as its initial `state`
  - When the UI is first rendered, UI components access the current state of the Redux store, and use that data to decide what to render. They also subscribe to any future store updates so they can know if the state has changed.
- Updates:
  - Something happens in the app, such as a user clicking a button
  - The app code dispatches an action to the Redux store, like `dispatch({type: 'counter/increment'})`
  - The store runs the reducer function again with the previous `state` and the current `action`, and saves the return value as the new `state`
  - The store notifies all parts of the UI that are subscribed that the store has been updated
  - Each UI component that needs data from the store checks to see if the parts of the state they need have changed.
  - Each component that sees its data has changed forces a re-render with the new data, so it can update what's shown on the screen

Here's what that data flow looks like visually:



**TIP**

- **Redux is a library for managing global application state**
  - Redux is typically used with the React-Redux library for integrating Redux and React together
  - Redux Toolkit is the recommended way to write Redux logic
- **Redux uses a "one-way data flow" app structure**
  - State describes the condition of the app at a point in time, and UI renders based on that state
  - When something happens in the app:
    - The UI dispatches an action
    - The store runs the reducers, and the state is updated based on what occurred
    - The store notifies the UI that the state has changed
  - The UI re-renders based on the new state
- **Redux uses several types of code**
  - *Actions* are plain objects with a `type` field, and describe "what happened" in the app
  - *Reducers* are functions that calculate a new state value based on previous state + an action
  - A Redux *store* runs the root reducer whenever an action is *dispatched*

First, let's define some actions.

**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 `store.dispatch()`.

Here's an example action which represents adding a new todo item:

```
const ADD_TODO = 'ADD_TODO'
```

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Actions are plain JavaScript objects. Actions must have a `type` property that indicates the type of action being performed. Types should typically be defined as string constants. Once your app is large enough, you may want to move them into a separate module.

```
import { ADD_TODO, REMOVE_TODO } from '../actionTypes'
```

Other than `type`, the structure of an action object is really up to you. If you're interested, check out Flux Standard Action for recommendations on how actions could be constructed.

We'll add one more action type to describe a user ticking off a todo as completed. We refer to a particular todo by `index` because we store them in an array. In a real app, it is wiser to generate a unique ID every time something new is created.

```
{
  type: TOGGLE_TODO,
  index: 5
}
```

It's a good idea to pass as little data in each action as possible. For example, it's better to pass `index` than the whole todo object.

Finally, we'll add one more action type for changing the currently visible todos.

```
{
  type: SET_VISIBILITY_FILTER,
  filter: SHOW_COMPLETED
}
```

A basic Flux Standard Action:

```
{
  type: 'ADD_TODO',
  payload: {
    text: 'Do something.'
  }
}
```

An FSA that represents an error, analogous to a rejected Promise:

```
{
  type: 'ADD_TODO',
  payload: new Error(),
  error: true
}
```

## Actions

An action MUST

- be a plain JavaScript object.
- have a `type` property.

An action MAY

- have an `error` property.
- have a `payload` property.
- have a `meta` property.

An action MUST NOT include properties other than `type`, `payload`, `error`, and `meta`.

## `type`

The `type` of an action identifies to the consumer the nature of the action that has occurred. `type` is a string constant. If two types are the same, they MUST be strictly equivalent (using `===`).

## `payload`

The optional `payload` property MAY be any type of value. It represents the payload of the action. Any information about the action that is not the `type` or status of the action should be part of the `payload` field.

By convention, if `error` is `true`, the `payload` SHOULD be an error object. This is akin to rejecting a promise with an error object.

## `error`

The optional `error` property MAY be set to `true` if the action represents an error.

An action whose `error` is true is analogous to a rejected Promise. By convention, the `payload` SHOULD be an error object.

If `error` has any other value besides `true`, including `undefined` and `null`, the action MUST NOT be interpreted as an error.

## `meta`

The optional `meta` property MAY be any type of value. It is intended for any extra information that is not part of the payload.

# Action Creators

**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.

In Redux, action creators simply return an action:

```
function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  }
}
```

This makes them portable and easy to test.

In traditional Flux, action creators often trigger a dispatch when invoked, like so:

```
function addTodoWithDispatch(text) {
  const action = {
    type: ADD_TODO,
    text
  }
  dispatch(action)
}
```

In Redux this is *not* the case. Instead, to actually initiate a dispatch, pass the result to the `dispatch()` function:

```
dispatch(addTodo(text))
dispatch(completeTodo(index))
```

Alternatively, you can create a **bound action creator** that automatically dispatches:

```
const boundAddTodo = text => dispatch(addTodo(text))
const boundCompleteTodo = index => dispatch(completeTodo(index))
```

Now you'll be able to call them directly:

```
boundAddTodo(text)
boundCompleteTodo(index)
```

The `dispatch()` function can be accessed directly from the store as `store.dispatch()`, but more likely you'll access it using a helper like react-redux's `connect()`. You can use `bindActionCreators()` to automatically bind many action creators to a `dispatch()` function.

Action creators can also be asynchronous and have side-effects. You can read about async actions in the advanced tutorial to learn how to handle AJAX responses and compose action creators into async control flow. Don't skip ahead to async actions until you've completed the basics tutorial, as it covers other important concepts that are prerequisite for the advanced tutorial and async actions.

## actions.js

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO'
export const TOGGLE_TODO = 'TOGGLE_TODO'
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

/*
 * other constants
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

/*
 * action creators
 */

export function addTodo(text) {
  return { type: ADD_TODO, text }
}

export function toggleTodo(index) {
  return { type: TOGGLE_TODO, index }
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

**Reducers** specify how the application's state changes in response to actions sent to the store. Remember that actions only describe *what happened*, but don't describe how the application's state changes.

# Designing the State Shape

In Redux, all the application state is stored as a single object. It's a good idea to think of its shape before writing any code. What's the minimal representation of your app's state as an object?

For our todo app, we want to store two different things:

- The currently selected visibility filter.
- The actual list of todos.

You'll often find that you need to store some data, as well as some UI state, in the state tree. This is fine, but try to keep the data separate from the UI state.

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
```

**Note on Relationships**

In a more complex app, you're going to want different entities to reference each other. We suggest that you keep your state as normalized as possible, without any nesting. Keep every entity in an object stored with an ID as a key, and use IDs to reference it from other entities, or lists. Think of the app's state as a database. This approach is described in normalizr's documentation in detail. For example, keeping `todosById: { id -> todo }` and `todos: array<id>` inside the state would be a better idea in a real app, but we're keeping the example simple.

# Handling Actions

Now that we've decided what our state object looks like, we're ready to write a reducer for it. The reducer is a pure function that takes the previous state and an action, and returns the next state.

```
(previousState, action) => nextState
```

It's called a reducer because it's the type of function you would pass to `Array.prototype.reduce(reducer, ?initialValue)`. It's very important that the reducer stays pure. Things you should **never** do inside a reducer:

- Mutate its arguments;
- Perform side effects like API calls and routing transitions;
- Call non-pure functions, e.g. `Date.now()` or `Math.random()`.

We'll explore how to perform side effects in the advanced walkthrough. For now, just remember that the reducer must be pure. **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.**

With this out of the way, let's start writing our reducer by gradually teaching it to understand the actions we defined earlier.

We'll start by specifying the initial state. Redux will call our reducer with an `undefined` state for the first time. This is our chance to return the initial state of our app:

```
import { VisibilityFilters } from './actions'

const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: []
}

function todoApp(state, action) {
  if (typeof state === 'undefined') {
    return initialState
  }

  // For now, don't handle any actions
  // and just return the state given to us.
  return state
}
```

One neat trick is to use the ES6 default arguments syntax to write this in a more compact way:

```
function todoApp(state = initialState, action) {
  // For now, don't handle any actions
  // and just return the state given to us.
  return state
}
```

Now let's handle `SET_VISIBILITY_FILTER`. All it needs to do is to change `visibilityFilter` on the state. Easy:

```
import {
  SET_VISIBILITY_FILTER,
  VisibilityFilters
} from './actions'

...

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Note that:

1. **We don't mutate the `state`.** We create a copy with `Object.assign()`. `Object.assign(state, { visibilityFilter: action.filter })` is also wrong: it will mutate the first argument. You **must** supply an empty object as the first parameter. You can also enable the object spread operator proposal to write `{ ...state, ...newState }` instead.

2. **We return the previous `state` in the `default` case.** It's important to return the previous `state` for any unknown action.

# Handling More Actions

We have two more actions to handle! Just like we did with `SET_VISIBILITY_FILTER`, we'll import the `ADD_TODO` and `TOGGLE_TODO` actions and then extend our reducer to handle `ADD_TODO`.

```
import {
  ADD_TODO,
  TOGGLE_TODO,
  SET_VISIBILITY_FILTER,
  VisibilityFilters
} from './actions'

...

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    default:
      return state
  }
}
```

Just like before, we never write directly to `state` or its fields, and instead we return new objects. The new `todos` is equal to the old `todos` concatenated with a single new item at the end. The fresh todo was constructed using the data from the action.

Finally, the implementation of the `TOGGLE_TODO` handler shouldn't come as a complete surprise:

```
case TOGGLE_TODO:
  return Object.assign({}, state, {
    todos: state.todos.map((todo, index) => {
      if (index === action.index) {
        return Object.assign({}, todo, {
          completed: !todo.completed
        })
      }
      return todo
    })
  })
```

Because we want to update a specific item in the array without resorting to mutations, we have to create a new array with the same items except the item at the index. If you find yourself often writing such operations, it's a good idea to use a helper like immutability-helper, updeep, or even a library like Immutable that has native support for deep updates. Just remember to never assign to anything inside the `state` unless you clone it first.

## JavaScript Demo: Object.assign()

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }

```

## Splitting Reducers

Here is our code so far. It is rather verbose:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: state.todos.map((todo, index) => {
          if (index === action.index) {
            return Object.assign({}, todo, {
              completed: !todo.completed
            })
          }
          return todo
        })
      })
    default:
      return state
  }
}
```

Is there a way to make it easier to comprehend? It seems like `todos` and `visibilityFilter` are updated completely independently. Sometimes state fields depend on one another and more consideration is required, but in our case we can easily split updating `todos` into a separate function:

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    default:
      return state
  }
}
```

Note that `todos` also accepts `state`—but `state` is an array! Now `todoApp` gives `todos` just a slice of the state to manage, and `todos` knows how to update just that slice. **This is called *reducer composition*, and it's the fundamental pattern of building Redux apps.**

Let's explore reducer composition more. Can we also extract a reducer managing just `visibilityFilter`? We can.

Below our imports, let's use ES6 Object Destructuring to declare `SHOW_ALL`:

```
const { SHOW_ALL } = VisibilityFilters
```

Then:

```
function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}
```

Now we can rewrite the main reducer as a function that calls the reducers managing parts of the state, and combines them into a single object. It also doesn't need to know the complete initial state any more. It's enough that the child reducers return their initial state when given `undefined` at first.

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

**Note that each of these reducers is managing its own part of the global state. The `state` parameter is different for every reducer, and corresponds to the part of the state it manages.**

This is already looking good! When the app is larger, we can split the reducers into separate files and keep them completely independent and managing different data domains.

Finally, Redux provides a utility called `combineReducers()` that does the same boilerplate logic that the `todoApp` above currently does. With its help, we can rewrite `todoApp` like this:

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Note that this is equivalent to:

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

## Source Code

`reducers.js`

```
import { combineReducers } from 'redux'
import {
  ADD_TODO,
  TOGGLE_TODO,
  SET_VISIBILITY_FILTER,
  VisibilityFilters
} from './actions'
const { SHOW_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

## Next Steps

Next, we'll explore how to create a Redux store that holds the state and takes care of calling your reducer when you dispatch an action.

# Store

In the previous sections, we defined the actions that represent the facts about "what happened" and the reducers that update the state according to those actions.

The **Store** is the object that brings them together. The store has the following responsibilities:

- Holds application state;
- Allows access to state via `getState()`;
- Allows state to be updated via `dispatch(action)`;
- Registers listeners via `subscribe(listener)`;
- Handles unregistering of listeners via the function returned by `subscribe(listener)`.

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

It's easy to create a store if you have a reducer. In the previous section, we used `combineReducers()` to combine several reducers into one. We will now import it, and pass it to `createStore()`.

```
import { createStore } from 'redux'
import todoApp from './reducers'
const store = createStore(todoApp)
```

You may optionally specify the initial state as the second argument to `createStore()`. This is useful for hydrating the state of the client to match the state of a Redux application running on the server.

# Dispatching Actions

Now that we have created a store, let's verify our program works! Even without any UI, we can already test the update logic.

```
import {
  addTodo,
  toggleTodo,
  setVisibilityFilter,
  VisibilityFilters
} from './actions'

// Log the initial state
console.log(store.getState())

// Every time the state changes, log it
// Note that subscribe() returns a function for unregistering the listener
const unsubscribe = store.subscribe(() => console.log(store.getState()))

// Dispatch some actions
store.dispatch(addTodo('Learn about actions'))
store.dispatch(addTodo('Learn about reducers'))
store.dispatch(addTodo('Learn about store'))
store.dispatch(toggleTodo(0))
store.dispatch(toggleTodo(1))
store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED))

// Stop listening to state updates
unsubscribe()
```

You can see how this causes the state held by the store to change:

```
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[0]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[1]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[2]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▼ Object {visibleTodoFilter: "SHOW_COMPLETED", todos: Array[3]}
  ▼ todos: Array[3]
    ▼ 0: Object
        completed: true
        text: "Learn about actions"
      ▶ __proto__: Object
    ▼ 1: Object
        completed: true
        text: "Learn about reducers"
      ▶ __proto__: Object
    ▼ 2: Object
        completed: false
        text: "Learn about store"
      ▶ __proto__: Object
      length: 3
    ▶ __proto__: Array[0]
    visibleTodoFilter: "SHOW_COMPLETED"
  ▶ __proto__: Object
```

We specified the behavior of our app before we even started writing the UI. We won't do this in this tutorial, but at this point you can write tests for your reducers and action creators. You won't need to mock anything because they are just pure functions. Call them, and make assertions on what they return.

# Data Flow

Redux architecture revolves around a **strict unidirectional data flow**.

The data lifecycle in any Redux app follows these 4 steps:

1. **You call** `store.dispatch(action)`.

An action is a plain object describing *what happened*. For example:

```
{ type: 'LIKE_ARTICLE', articleId: 42 }
{ type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'Mary' } }
{ type: 'ADD_TODO', text: 'Read the Redux docs.' }
```

Think of an action as a very brief snippet of news. "Mary liked article 42." or "'Read the Redux docs.' was added to the list of todos."

You can call `store.dispatch(action)` from anywhere in your app, including components and XHR callbacks, or even at scheduled intervals.

2. **The Redux store calls the reducer function you gave it.**

The store will pass two arguments to the reducer: the current state tree and the action. For example, in the todo app, the root reducer might receive something like this:

```
// The current application state (list of todos and chosen filter)
let previousState = {
  visibleTodoFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Read the docs.',
      complete: false
    }
  ]
}

// The action being performed (adding a todo)
let action = {
  type: 'ADD_TODO',
  text: 'Understand the flow.'
}

// Your reducer returns the next application state
let nextState = todoApp(previousState, action)
```

Note that a reducer is a pure function. It only *computes* the next state. It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an action is dispatched.

3. **The root reducer may combine the output of multiple reducers into a single state tree.**

How you structure the root reducer is completely up to you. Redux ships with a `combineReducers()` helper function, useful for "splitting" the root reducer into separate functions that each manage one branch of the state tree.

Here's how `combineReducers()` works. Let's say you have two reducers, one for a list of todos, and another for the currently selected filter setting:

```
function todos(state = [], action) {
  // Somehow calculate it...
  return nextState
}

function visibleTodoFilter(state = 'SHOW_ALL', action) {
  // Somehow calculate it...
  return nextState
}

let todoApp = combineReducers({
  todos,
  visibleTodoFilter
})
```

When you emit an action, `todoApp` returned by `combineReducers` will call both reducers:

```
let nextTodos = todos(state.todos, action)
let nextVisibleTodoFilter = visibleTodoFilter(state.visibleTodoFilter, action)
```

It will then combine both sets of results into a single state tree:

```
return {
  todos: nextTodos,
  visibleTodoFilter: nextVisibleTodoFilter
}
```

While `combineReducers()` is a handy helper utility, you don't have to use it; feel free to write your own root reducer!

4. **The Redux store saves the complete state tree returned by the root reducer.**

This new tree is now the next state of your app! Every listener registered with `store.subscribe(listener)` will now be invoked; listeners may call `store.getState()` to get the current state.

Now, the UI can be updated to reflect the new state. If you use bindings like React Redux, this is the point at which `component.setState(newState)` is called.

# Usage with React

From the very beginning, we need to stress that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

That said, Redux works especially well with libraries like React and Deku because they let you describe UI as a function of state, and Redux emits state updates in response to actions.

We will use React to build our simple todo app, and cover the basics of how to use React with Redux.

> **Note**: see **the official React-Redux docs at https://react-redux.js.org** for a complete guide on how to use Redux and React together.

# Installing React Redux

React bindings are not included in Redux by default. You need to install them explicitly:

```
npm install react-redux
```

# Presentational and Container Components

React bindings for Redux separate *presentational* components from *container* components. This approach can make your app easier to understand and allow you to more easily reuse components. Here's a summary of the differences between presentational and container components (but if you're unfamiliar, we recommend that you also read Dan Abramov's original article describing the concept of presentational and container components):

| | Presentational Components | Container Components |
|---|---|---|
| Purpose | How things look (markup, styles) | How things work (data fetching, state updates) |
| Aware of Redux | No | Yes |
| To read data | Read data from props | Subscribe to Redux state |
| To change data | Invoke callbacks from props | Dispatch Redux actions |
| Are written | By hand | Usually generated by React Redux |

Most of the components we'll write will be presentational, but we'll need to generate a few container components to connect them to the Redux store. This and the design brief below do not imply container components must be near the top of the component tree. If a container component becomes too complex (i.e. it has heavily nested presentational components with countless callbacks being passed down), introduce another container within the component tree as noted in the FAQ.

Technically you could write the container components by hand using `store.subscribe()`. We don't advise you to do this because React Redux makes many performance optimizations that are hard to do by hand. For this reason, rather than write container components, we will generate them using the `connect()` function provided by React Redux, as you will see below.

# Designing Component Hierarchy

Remember how we designed the shape of the root state object? It's time we design the UI hierarchy to match it. This is not a Redux-specific task. Thinking in React is a great tutorial that explains the process.

Our design brief is simple. We want to show a list of todo items. On click, a todo item is crossed out as completed. We want to show a field where the user may add a new todo. In the footer, we want to show a toggle to show all, only completed, or only active todos.

## Designing Presentational Components

I see the following presentational components and their props emerge from this brief:

- `TodoList` is a list showing visible todos.
  - `todos: Array` is an array of todo items with `{ id, text, completed }` shape.
  - `onTodoClick(id: number)` is a callback to invoke when a todo is clicked.
- `Todo` is a single todo item.
  - `text: string` is the text to show.
  - `completed: boolean` is whether the todo should appear crossed out.
  - `onClick()` is a callback to invoke when the todo is clicked.
- `Link` is a link with a callback.
  - `onClick()` is a callback to invoke when the link is clicked.
- `Footer` is where we let the user change currently visible todos.
- `App` is the root component that renders everything else.

They describe the *look* but don't know *where* the data comes from, or *how* to change it. They only render what's given to them. If you migrate from Redux to something else, you'll be able to keep all these components exactly the same. They have no dependency on Redux.

## Designing Container Components

We will also need some container components to connect the presentational components to Redux. For example, the presentational `TodoList` component needs a container like `VisibleTodoList` that subscribes to the Redux store and knows how to apply the current visibility filter. To change the visibility filter, we will provide a `FilterLink` container component that renders a `Link` that dispatches an appropriate action on click:

- **`VisibleTodoList`** filters the todos according to the current visibility filter and renders a `TodoList`.
- **`FilterLink`** gets the current visibility filter and renders a `Link`.
  - `filter: string` is the visibility filter it represents.

## Designing Other Components

Sometimes it's hard to tell if some component should be a presentational component or a container. For example, sometimes form and function are really coupled together, such as in the case of this tiny component:

- **`AddTodo`** is an input field with an "Add" button

Technically we could split it into two components but it might be too early at this stage. It's fine to mix presentation and logic in a component that is very small. As it grows, it will be more obvious how to split it, so we'll leave it mixed.

## Implementing Presentational Components

These are all normal React components, so we won't examine them in detail. We write functional stateless components unless we need to use local state or the lifecycle methods. This doesn't mean that presentational components *have to* be functions—it's just easier to define them this way. If and when you need to add local state, lifecycle methods, or performance optimizations, you can convert them to classes.

`components/Todo.js`

```
import React from 'react'
import PropTypes from 'prop-types'

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

components/TodoList.js

```
import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map((todo, index) => (
      <Todo key={todo.id} {...todo} onClick={() => onTodoClick(todo.id)} />
    ))}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,
      completed: PropTypes.bool.isRequired,
      text: PropTypes.string.isRequired
    }).isRequired
  ).isRequired,
  onTodoClick: PropTypes.func.isRequired
}

export default TodoList
```

components/Link.js

```
import React from 'react'
import PropTypes from 'prop-types'

const Link = ({ active, children, onClick }) => (
  <button
    onClick={onClick}
    disabled={active}
    style={{
      marginLeft: '4px'
    }}
  >
    {children}
  </button>
)

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link
```

components/Footer.js

```
import React from 'react'
import FilterLink from '../containers/FilterLink'
import { VisibilityFilters } from '../actions'

const Footer = () => (
  <p>
    Show: <FilterLink filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
    {', '}
    <FilterLink filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
    {', '}
    <FilterLink filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
  </p>
)

export default Footer
```

## Implementing Container Components

Now it's time to hook up those presentational components to Redux by creating some containers. Technically, a container component is just a React component that uses `store.subscribe()` to read a part of the Redux state tree and supply props to a presentational component it renders. You could write a container component by hand, but we suggest instead generating container components with the React Redux library's `connect()` function, which provides many useful optimizations to prevent unnecessary re-renders. (One result of this is that you shouldn't have to worry about the React performance suggestion of implementing `shouldComponentUpdate` yourself.)

To use `connect()`, you need to define a special function called `mapStateToProps` that describes how to transform the current Redux store state into the props you want to pass to a presentational component you are wrapping. For example, `VisibleTodoList` needs to calculate `todos` to pass to the `TodoList`, so we define a function that filters the `state.todos` according to the `state.visibilityFilter`, and use it in its `mapStateToProps`:

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
    case 'SHOW_ALL':
    default:
      return todos
  }
}

const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

In addition to reading the state, container components can dispatch actions. In a similar fashion, you can define a function called `mapDispatchToProps()` that receives the `dispatch()` method and returns callback props that you want to inject into the presentational component. For example, we want the `VisibleTodoList` to inject a prop called `onTodoClick` into the `TodoList` component, and we want `onTodoClick` to dispatch a `TOGGLE_TODO` action:

```
const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}
```

Finally, we create the `VisibleTodoList` by calling `connect()` and passing these two functions:

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(mapStateToProps, mapDispatchToProps)(TodoList)

export default VisibleTodoList
```

`containers/FilterLink.js`

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(mapStateToProps, mapDispatchToProps)(Link)

export default FilterLink
```

`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
```

## Implementing Other Components

`containers/AddTodo.js`

Recall as mentioned previously, both the presentation and logic for the `AddTodo` component are mixed into a single definition.

```
import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'

let AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form
        onSubmit={e => {
          e.preventDefault()
          if (!input.value.trim()) {
            return
          }
          dispatch(addTodo(input.value))
          input.value = ''
        }}
      >
        <input
          ref={node => {
            input = node
          }}
        />
        <button type="submit">Add Todo</button>
      </form>
    </div>
  )
}
AddTodo = connect()(AddTodo)

export default AddTodo
```

If you are unfamiliar with the `ref` attribute, please read this documentation to familiarize yourself with the recommended use of this attribute.

# Tying the containers together within a component

`components/App.js`

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App
```

# Passing the Store

All container components need access to the Redux store so they can subscribe to it. One option would be to pass it as a prop to every container component. However it gets tedious, as you have to wire `store` even through presentational components just because they happen to render a container deep in the component tree.

The option we recommend is to use a special React Redux component called `<Provider>` to magically make the store available to all container components in the application without passing it explicitly. You only need to use it once when you render the root component:

**`index.js`**

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

const store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```