Learn Redux

Codecademy

Core Concepts in Redux

Introduction to Redux

When building React applications, we share data between React components. When the application scales and sharing that data gets more complex, it's time to use a state-management library.

Redux is the most commonly used state-management library with React. Redux lets us use plain JS syntax, but enforces consistent patterns that make our applications reliable and predictable.

Redux is a state management library that follows a pattern known as the Flux architecture. In Flux pattern, and in Redux, shared information is not stored in components but in a single object. Components are just given data to render and can request changes using events called actions. The state is available throughout the application and updates are made in a predictable manner: components are **notified** whenever a change is made to the state.

One-Way Data Flow

In most applications, there are three parts:

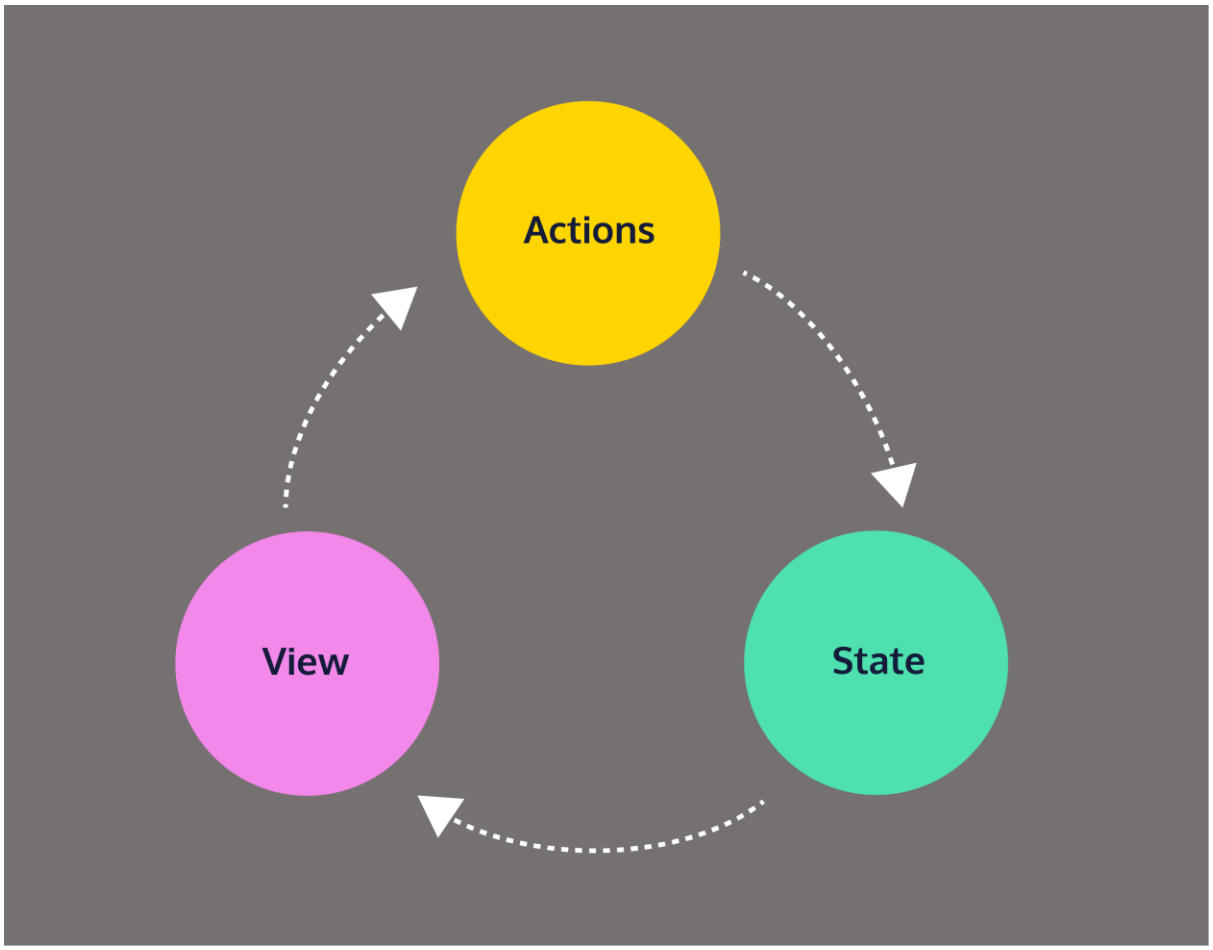
- **State**- the current data used in the app.
- **View**- the user interface displayed to users.
- **Actions**- events that a user can take to change the state.

The flow of information would go like this:

- The state holds the current data used by the app's components.
- The view components display that state data.
- When a user interacts with the view, like clicking a button, the state will be updated in some way.
- The view is updated to display the new state.

With plain React, these three parts overlap quite a bit. Components not only render the user interface, but also manage their own state. When actions that may change the state occur, components need to directly communicate these changes to each other.

Redux helps separate the state, the view, and actions by requiring that the state be managed by a single source. Requests to change the state are sent to this single source by view components in the form of an action. Any components of the view that would be affected by these changes are informed by this changes are informed by this single source. By imposing this structure, Redux makes our code more readable, reliable, and maintainable.



State

State is the current information behind a web application.

For a calendar application it includes the events, the current timezone, and the display filters. For a todo app it includes the todo items, the current order of the items, and display filters. For a word editor, it includes the contents of the document, the print settings, and comments.

With Redux, state can be any JS type, including: number, string, Boolean, array, and object.

Here’s an example state for a todo app:

```
const state = [‘Print trail map’, ‘Pack snacks’, ‘summit the mountain’];
```

Each piece of information in this state – an array in this case – would inform some part of the user interface.

Actions

Most well-designed applications have components communicating with each other. This interaction of sharing data is defined as an **action**. In Redux, actions are represented as plain JS objects.

```
const action = {
      type: ‘todos/addToo’,
      payload: ‘take selfies’
};
```

- Every action must have a **type** property with a string value. This describes the action.
- Typically, an action has a **payload** property with an object value. This includes any information related to the action.
- When an action is generated and notifies other parts of the application, we say that the action is dispatched.

Reducers

A **reducer**, or reducer function, is a plain JS function that defines how the current state and an action are used in combination to create the new state.

```
const initialState = [ 'Print trail map', 'Pack snacks', 'Summit the mountain'
];

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'todos/addTodo': {
      return [ ...state, action.payload];
    }
    case 'todos/removeAll': {
      return [];
    }
    default: {
      return state;
    }
  }
}
```

There are few things to note:

- It’s a plain JS function

- It defines the application's next state given a current state and a specific action
- It returns a default initial state if no action is provided
- It returns the current state if the action is not recognized

Rules of Reducers

1. They should only calculate the new state value based on the **state** and **action** arguments.
2. They are not allowed to modify the existing state. Instead, they must copy the existing state and make changes to the copied values.
3. They must not do any asynchronous logic or have other 'side effects'.

Immutable Updates and Pure Functions

In programming, there is a more general way to describe the three rules of reducers in Redux: reducers must make **immutable** updates and be pure functions.

If a function makes immutable updates to its arguments, it does not change the argument but instead makes a copy and changes that copy. Its called updating immutably because the function doesn't change, or mutate, the arguments.

This function mutates its argument:

```
const mutableUpdater = (obj) => {
  obj.completed = !obj.completed;
  return obj;
}

const task = { text: 'do dishes', completed: false };
const updatedTask = mutableUpdater(task);
console.log(updatedTask);
// Prints { text: 'do dishes', completed: true };

console.log(task);
// Prints { text: 'do dishes', completed: true };
```

This function immutably updates its argument:

```
const immutableUpdater = (obj) => {
  return {
    ...obj,
    completed: !obj.completed
```

```
  }
}

const task = { text: 'iron clothes', completed: false };
const updatedTask = immutableUpdater(task);
console.log(updatedTask);
// Prints { text: 'iron clothes', completed: true };

console.log(task);
// Prints { text: 'iron clothes', completed: false };
```

By copying the contents of the argument obj into a new object ({…obj}) and updating the completed property of the copy, the argument obj will remain unchanged.

In JS plain strings, numbers, and Booleans are immutable.

If a function is **pure**, then it will always have the same outputs given the same inputs.

Example, of not pure function:

```
const addItemToList = (list) => {
  let item;
  fetch('https://anything.com/endpoint')
    .then(response => {
      if (!response.ok) {
        item = {};
      }

      item = response.json();
    });

   return [...list, item];
};
```

It can be made pure like this:

```
let item;
  fetch('https://anything.com/endpoint')
    .then(response => {
      if (!response.ok) {
        item = {};
      }

      item = response.json();
    });

const addItemToList = (list, item) => {
    return [...list, item];
};
```

Store

Redux uses a special object called the **store**. The store acts as a container for state, it provides a way to dispatch actions, and it calls the reducer when actions are dispatched. In nearly every Redux application, there will only be one store.

We can rephrase data flow as:

1. The **store** initialized the **state** with a default value.
2. The **view** displays that **state**.
3. When a user interacts with the **view**, an **action** is dispatched to the **store**.
4. The dispatched **action** and the **current state** are combined in the **store's reducer** to determine the **next state**.
5. The **view** is updated to display the **new state**.