Lecture 10

State Management with **Redux**

Contents

- Why Redux
- Redux: single store, actions, reducers
- Redux middleware for dev tooling
- React-Redux library
- Redux middleware for async actions
- Selectors with Reselect library
- Recommendations

Why Redux?

Managing state w/o Redux

- React state is used as a data store for React/React Native app.
  - This is the traditional approach

```
export default function TodoList() {
    const [state, setState] = useState({
        todos: [],
        searchTerm: '',
    });
}
```

- An app's state is often *spread* across multiple components, each with their own state.

Pros & Cons of using state

- Upsides
  - Low barrier to entry
  - Doesn’t require any additional libraries
- Downsides
  - Can lead to *prop drilling* problem
  - Can’t be used to server-side render React components
  - No single source of truth (multiple component states)
  - Often leads to tight coupling of presentation and data model

Alternatives to state

- MobX
  - https://github.com/mobxjs/mobx
- Redux
  - http://redux.js.org
- Reflux
  - https://github.com/reflux/refluxjs
- Any flux-related libraries

Why *choose* Redux?

- Huge ecosystem of middleware
- Great developer tools
- Great documentation
  - http://redux.js.org
- State-of-the-art choice for React state management
- Well supported by other libraries/frameworks
- Supports React server side rendering

A single source of truth

- One of Redux's core principle is: "One object, One store".

```
{
    isLoading: false,
    todos: [{}, {}, {}],
    searchTerm: 'spring 2025',
    userProfile: {
        firstName: 'Quan',
        lastName: 'Dang Dinh',
        email: 'quandd@hanu.edu.vn',
        loginToken: 'aaaa-bbbb-cccc-dddd'
    }
}
```

Synchronous

- Redux only handles synchronous data flow
- Must apply middleware to work asynchronously
- Example:
  - redux-thunk
  - redux-promise
  - redux-promise-middleware
  - redux-saga

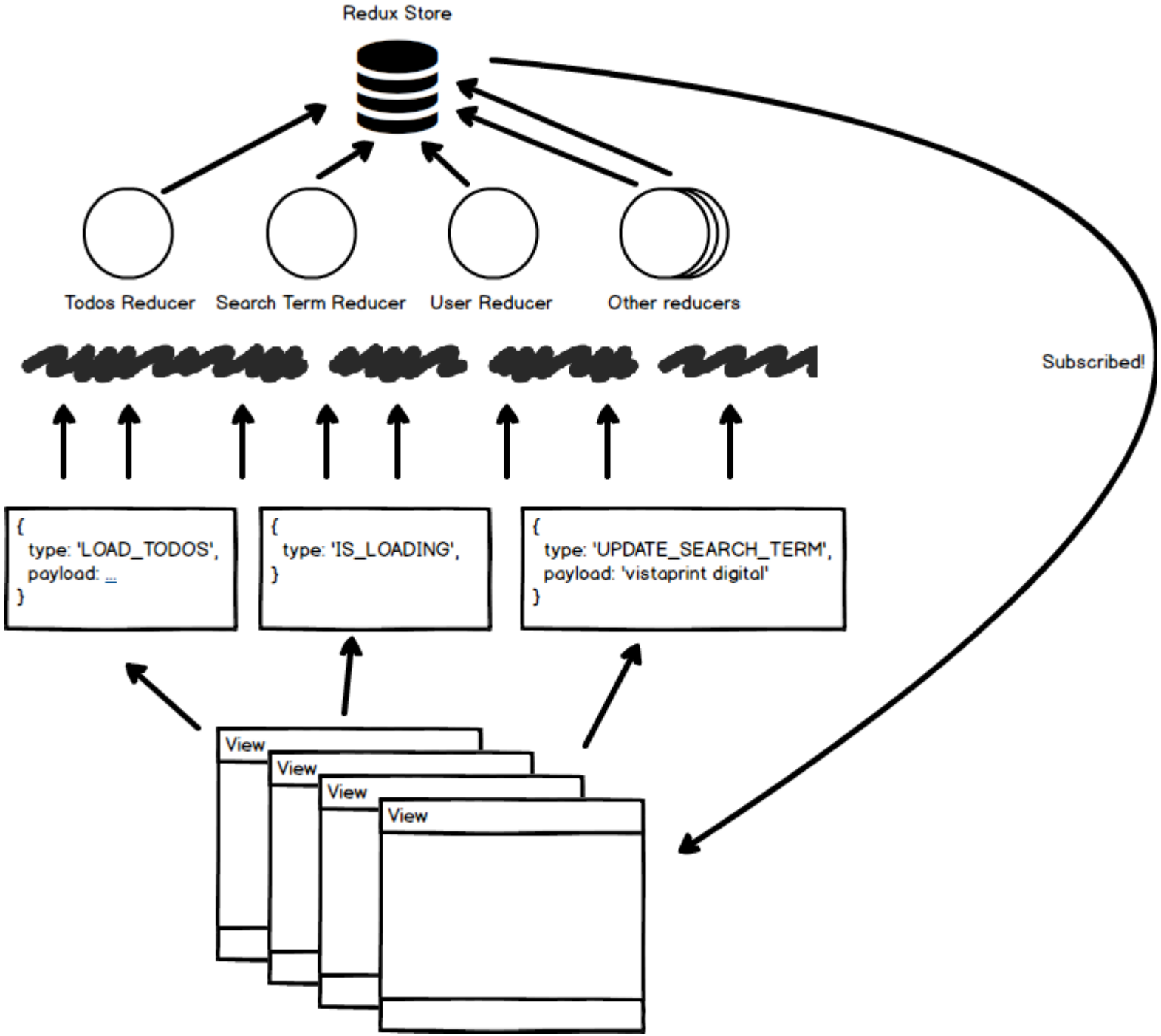
Redux Data Flow

4 core components of Redux data flow

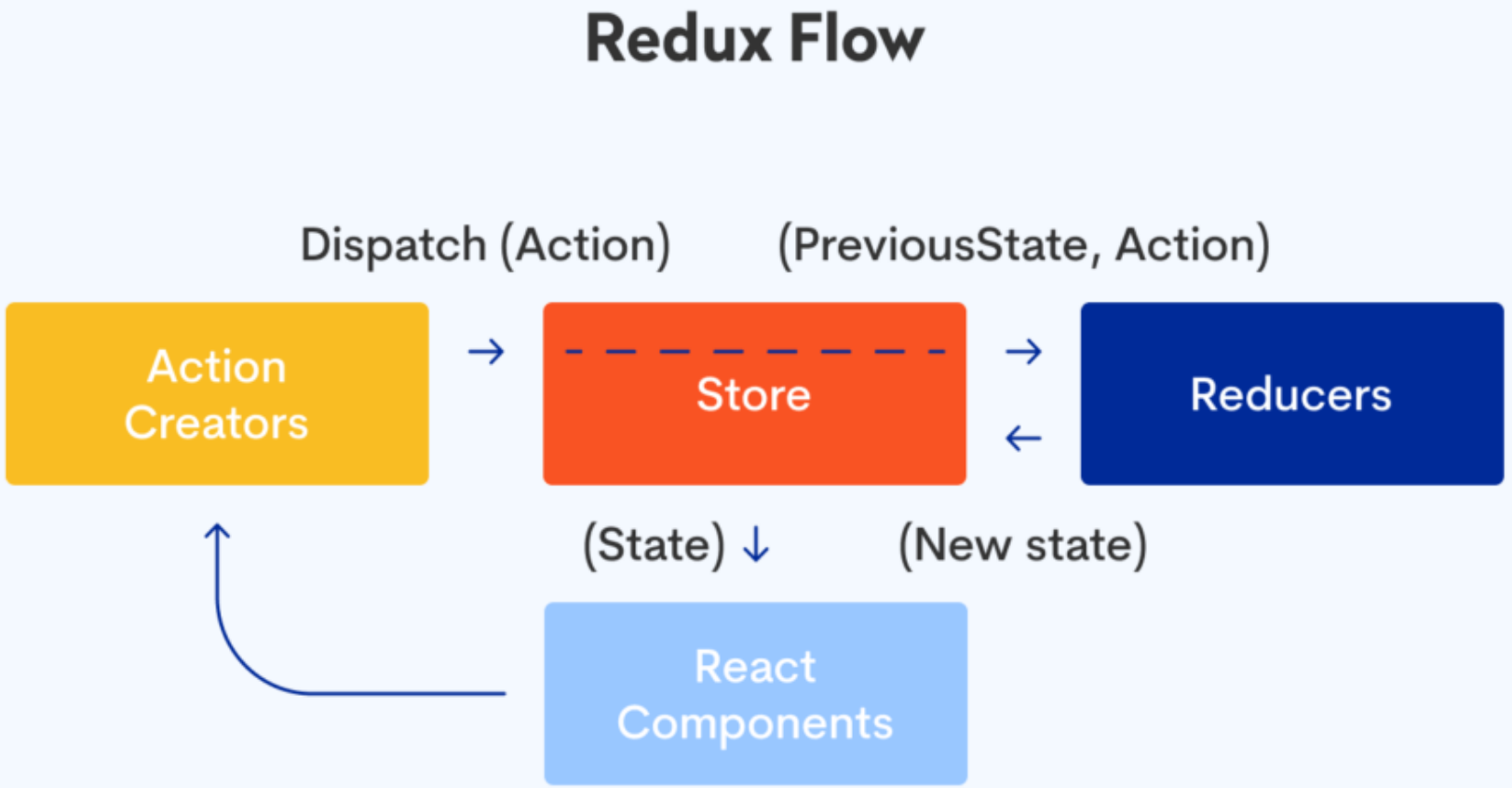
- To use Redux effectively, we need to understand its **four main components:**
  1. *Store*: The centralized storage that holds the entire application state.
  2. *Views (UI Components):* React components that connect to the store to display data and trigger actions.
  3. *Actions*: Plain JavaScript objects that describe what happened (e.g., user interactions or API responses).
  4. *Reducers*: Pure functions that take the current state and an action, then return a new state.

➞ These four components work together to create a predictable and scalable state management system in Redux.

Redux Data Flow



Redux data flow



Redux Actions

Updating State with Actions

- Redux state is read-only
- Changes are made via dispatched **actions**
- Actions have a `type` property
  - And often the `payload` property
- An action should *describe what's changed*
- Example:

```
store.dispatch({ type: 'BEGIN_LOADING' });

store.dispatch({ type: 'DONE_LOADING' })

store.dispatch({ type: 'UPDATE_SEARCH_TERM', payload: 'mpr' })
```

Redux Action convention: https://github.com/redux-utilities/flux-standard-action

Action Creators

- A factory function that returns an action is often used.
  - Although this is NOT necessary
- Action factory functions can be *synchronous* or *asynchronous*
  - Asynchronous functions require the use of middleware to dispatch more than one actions

```
// searchTermActions.js
function updateSearchTerm(searchTerm) {
    return {
        type: 'UPDATE_SEARCH_TERM',
        payload: searchTerm,
    };
}

// SearchComponent.js
store.dispatch(updateSearchTerm('homework'));
```

Redux Reducers

What is JavaScript's pure function?

A Redux *reducer* must be a *pure* function.

- A *pure* function always return the same result for the same inputs.
  - It is not affected by any state, data or changes in the app.
- Pure function does not introduce any side effect to the app.
  - e.g. sending a request, modifying data outside of the function…

Redux Reducers

- Functions that decide how each action transforms their respective piece of state
- Must be pure, side-effect free functions
- Each reducer maps to exactly 1 part of your state tree object
- The reducer **never modifies the original state directly** but returns a new state object instead.

Redux Reducers

- Reducers **return new state** for their respective piece of the state tree object.
  - Note: We are not changing state directly!
  - Reducers calculate a new state given the previous state and an action.

```
function todoReducer(state = [], action) {
    switch (action.type) {
        case 'ADD_TODO':
            // creates new array, adds todo at the end
            return [...state, action.payload];
        case 'REMOVE_TODO':
            // creates new array filtering out matching todo by id
            return state.filter(todo => todo.id !== action.payload);
        default:
            return state;
    }
}
```

The Root Reducer

- A Redux app really only has one reducer function:
  - The “root reducer” that you will pass to `createStore` later on.
- This root reducer function handles *all* actions that are dispatched.
  - Therefore, it calculates the *entire* new state.

Root Reducer example:

```
export default function appReducer(state = initialState, action) {
    switch (action.type) {
        case 'todos/todoAdded': {
            return {
                ...state,
                todos: [...state.todos, newTodo]
            };
        }
        case 'todos/todoToggled': {
            return {
                ...state,
                todos: [...state.todos, toggledTodo]
            };
        }
        default: return state;
    }
}
```

Splitting Reducers

- Redux reducers are typically split apart based on the section of the Redux state that they update.
  - Avoid putting too many reducers in a single file.
- The reducer for a specific section of the Redux app state is called a "slice reducer".
  - Actions related to a slice reducer should have the same prefix.
  - E.g. `todos/todoAdd`, `todos/todoDelete`

Example of a slice reducer

```
// todosSlice.js
const initialState = [
    { id: 0, text: 'Learn React', completed: true },
    { id: 1, text: 'Learn Redux', completed: false },
]

export default function todosReducer(state = initialState, action) {
    switch (action.type) {
        case 'todos/todoAdded': {
            // code to handle this action
        }
        case 'todos/todoToggled': {
            // code to handle this action
        }
        default:
            return state
    }
}
```

Combining Reducers

- The Redux store needs only one root reducer function when we create it.
  - Therefore, we need to combine all slice reducers into one.
  - We can do it manually or with the `combineReducers` utility function

```
import { combineReducers } from 'redux';

import todosReducer from './features/todos/todosSlice';
import filtersReducer from './features/filters/filtersSlice';

const rootReducer = combineReducers({
  todos: todosReducer,
  filters: filtersReducer
});

export default rootReducer;
```

Combining Reducers

- The Redux store needs only one root reducer function when we create it.
  - Therefore, we need to combine all slice reducers into one.
  - We can do it manually or with the `combineReducers` utility function

```
import { combineReducers } from 'redux';

import todosReducer from './features/todos/todosSlice';
import filtersReducer from './features/filters/filtersSlice';

const rootReducer = combineReducers({
  todos: todosReducer,
  filters: filtersReducer
});

export default rootReducer;
```

Combining Reducers (manually)

```
import todosReducer from './features/todos/todosSlice';
import filtersReducer from './features/filters/filtersSlice';

export default function rootReducer(state = {}, action) {
    return {
        // the value of `state.todos` is whatever
        // the todos reducer returns
        todos: todosReducer(state.todos, action),
        // For both reducers, we only pass in
        // their slice of the state
        filters: filtersReducer(state.filters, action)
    };
}
```

Redux Store

What is a Redux store?

- A store holds the whole state tree of the app.
  - The state tree is read-only.
  - The only way to change the state inside the store is to dispatch an action on it.
- A store is not a class.
  - It's just an object with a few methods on it.

Creating a Redux store

```
import { createStore, combineReducers, applyMiddleware } from 'redux';

const store = createStore(
    combineReducers({
        isLoading: loadingReducer,
        todos: todosReducer,
        searchTerm: searchTermReducer,
        bookmarks: bookMarksReducer,
        userProfile: userProfileReducer,
    }), // required - the root reducer
    INITIAL_STATE, // optional
    applyMiddleware(logger, thunk), // optional
);
```

Listening for store updates

- The `subscribe` method adds a change listener function.
  - It will be called any time an action is dispatched, and some part of the state tree may potentially have changed.

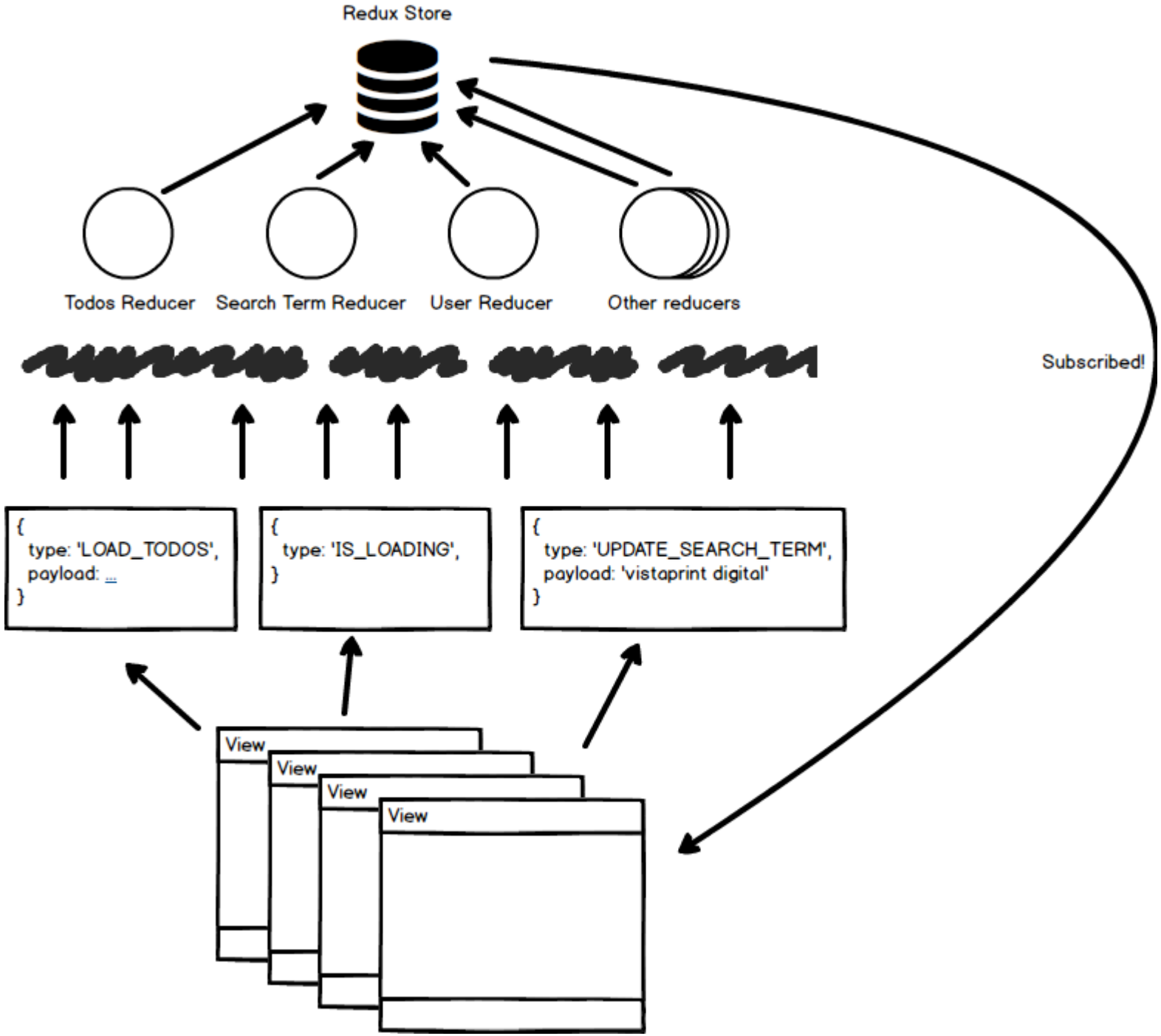
```
let currentValue;
function handleChange() {
    let previousValue = currentValue;
    currentValue = store.getState().some.property;
    if (previousValue !== currentValue) {
        console.log(
            'Some property changed from', previousValue,
            'to', currentValue
        )
    }
}
const unsubscribe = store.subscribe(handleChange);
```

RECAP:

Redux

Data

Flow



Redux and Middleware

What is Redux Middleware?

- Simplified data flow (or workflow):

```
[dispatchedActions] => [middleware] => [reducers] => store
```

- Similar to Express middleware
- Sits between action creators and reducers
- Can block or dispatch additional actions
- Has full visibility into action and the state of your app
- Can handle `async` actions (more on this later)
- Applied to store using `applyMiddleware` function

Redux Logger

Logs to your console...

- The state before the action
- The dispatched action
- The state after the action has passed through the reducers

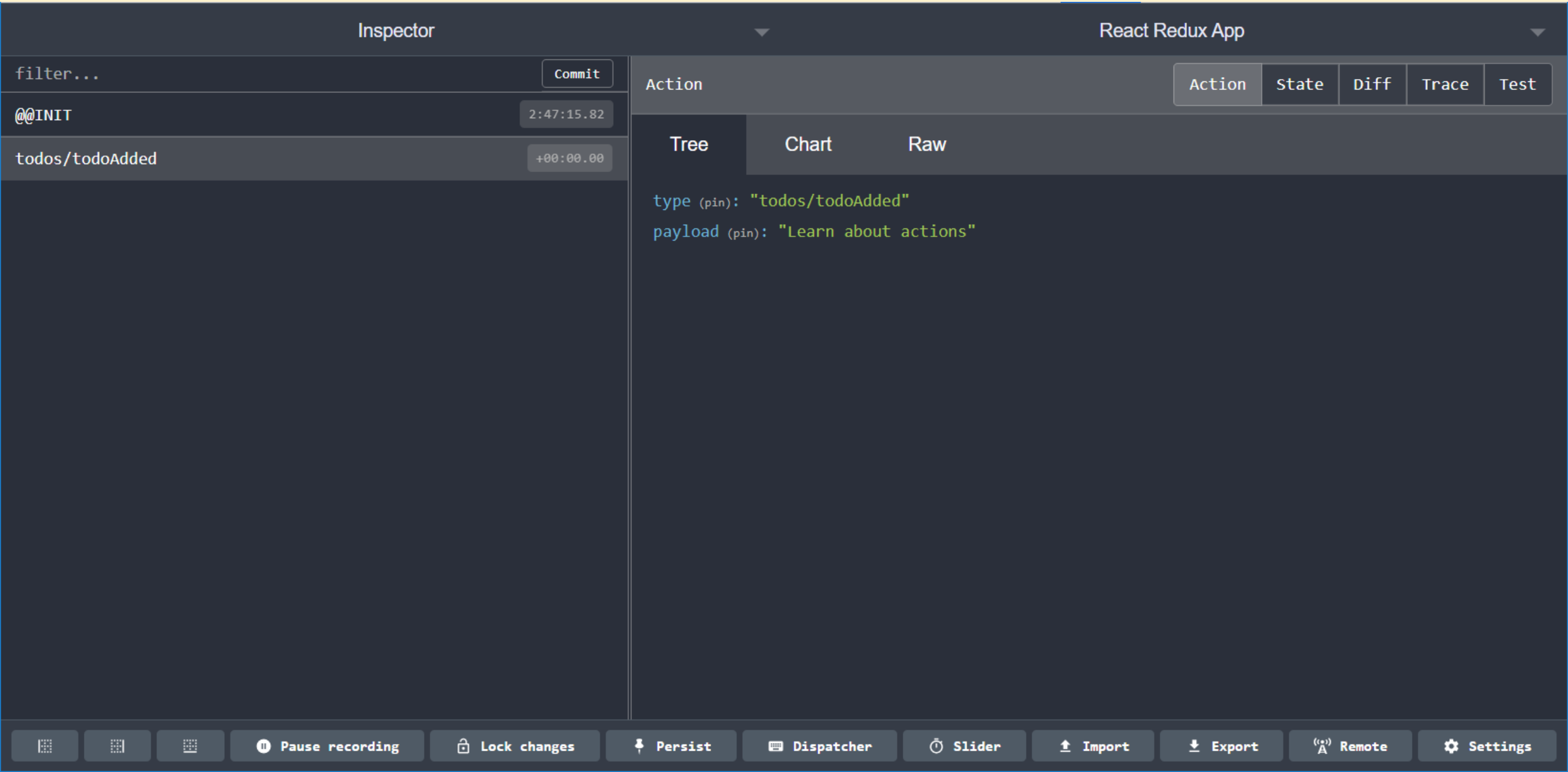
Redux DevTools

https://github.com/reduxjs/redux-devtools

- Exposes all actions and a timeline
- Allows for time travel debugging
- Allows you to dispatch actions from the tool
- Available as a Chrome Extension, Firefox Extension, or Electron app

- **(*)** *You'll likely want to strip out Redux DevTools and Redux Logger in production code.*

Redux DevTools



Container Components

- Concerned with how things work
- Typically don’t contain any presentation
  - Maybe just a View or a Fragment
- Pass down data and callbacks to other components as props
- Callbacks include action creators

Working with Redux store

Using `react-redux`

`useSelector()` hook

- Lets your React components read data from the Redux store.
- `useSelector` accepts a single function called *selector function*.
  - A selector takes the entire Redux store state as its argument, reads some value from the state, and returns that result.

```
const selectTodos = state => state.todos;


const selectAllCompletedTodos = state => {
    const completedTodos = state.todos.filter(
        todo => todo.completed
    );
    return completedTodos.length;
}
```

`useSelector()` hook example

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

const selectTodos = state => state.todos;

export const TodoList = () => {
    const todos = useSelector(selectTodos);
    return (
        <FlatList
            data={todos}
            renderItem={e => <Todo
                key={e.item.id}
                todo={e.item.todo} />}
            style="s.todoList" />
    );
};
```

`useSelector()` hook

- useSelector automatically subscribes to the Redux store for us.
- Any time an action is dispatched and the store is changed, it will call its selector function again right away.
  - So that our state will be updated and our component re-rendered properly.

`useDispatch()` hook

- The `useDispatch` hook gives us the store's `dispatch` method as its result.
  - We can declare `const dispatch = useDispatch()` in any component and later use `dispatch(someAction)` as needed.

useDispatch() hook

```
export default Header = () => {
    const [text, setText] = useState('');
    const dispatch = useDispatch();
    const handleSubmit = txt => {
        // Dispatch the "todo added" action with this text
        dispatch({ type: 'todos/todoAdded', payload: txt });
        // Clear out the text input
        setText('');
    };
    return (
        <TextInput
            placeholder="What needs to be done?"
            value={text}
            onChangeText={val => setText(val)}
            onSubmitEditing={handleSubmit}
        />
    );
};
```

Redux store Provider

- **Problem:** Without a `Provider`, `useSelector` and `useDispatch` **cannot** find the Redux store by themselves!!
  - Redux store Provider is similar to Context Provider.
  - These hooks can't automatically import the Redux store from where it was created.

➢ We have to tell `react-redux` what store we want to use in our components.
  - By wrapping a `<Provider>` component around the entire `<App>`, and passing the Redux store as a prop to `<Provider>`.

Redux store Provider

```
// index.js
import { Provider } from 'react-redux';
import { registerRootComponent } from 'expo';
import App from './App';
import store from './store';

registerRootComponent(
    <Provider store={store}>
        <App />
    </Provider>
);
```

Async actions

With `redux-thunk` middleware

The Basic Middleware Flow

- Redux only works *synchronously* out of the box.

```
[dispatchedActions] => [middleware] => [reducers] => store
```

1. Action creators return actions
2. Actions flow through middleware
3. Actions pass on to reducers
4. Reducers return updated state to the store

What's a thunk?

- A thunk is a function that wraps an expression to delay its evaluation.

```
// calculation of 1 + 2 is immediate
// x === 3
let x = 1 + 2;
```

```
// calculation of 1 + 2 is delayed
// foo can be called later to perform the calculation
// foo is a thunk!
let foo = () => 1 + 2;
```

Using redux-thunk

- First, create a modified version of the store so that it accepts thunk functions:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducer';

// The store now has the ability to accept thunk functions in `dispatch`
const store = createStore(rootReducer, applyMiddleware(thunk));
export default store;
```

Example: Dispatching a Function

- Using `redux-thunk`

```
// thunk function
export async function fetchTodos(dispatch) {
    const response = await fetchData('/fakeApi/todos');
    dispatch({ type: 'todos/todosLoaded', payload:
response.todos })
}


// using the thunk function later on
store.dispatch(fetchTodos);
```