Redux

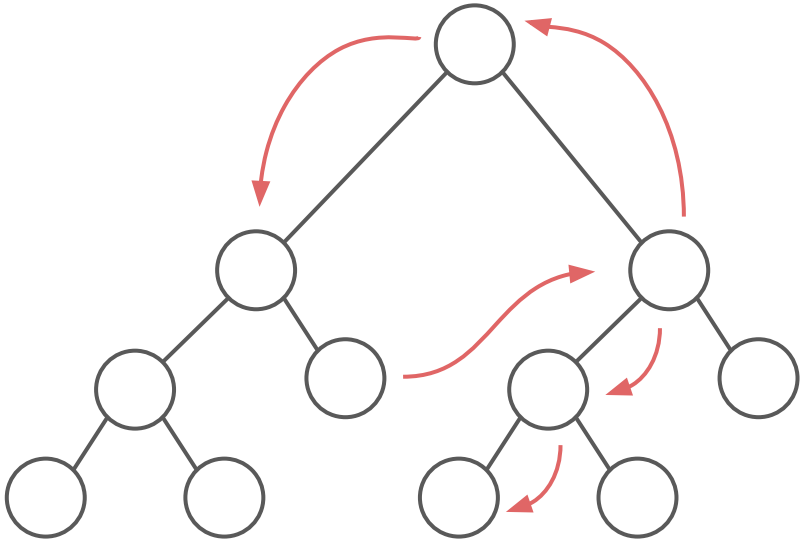
Advanced
state management

Redux is an alternative to state management inside components

Why / What you'll learn

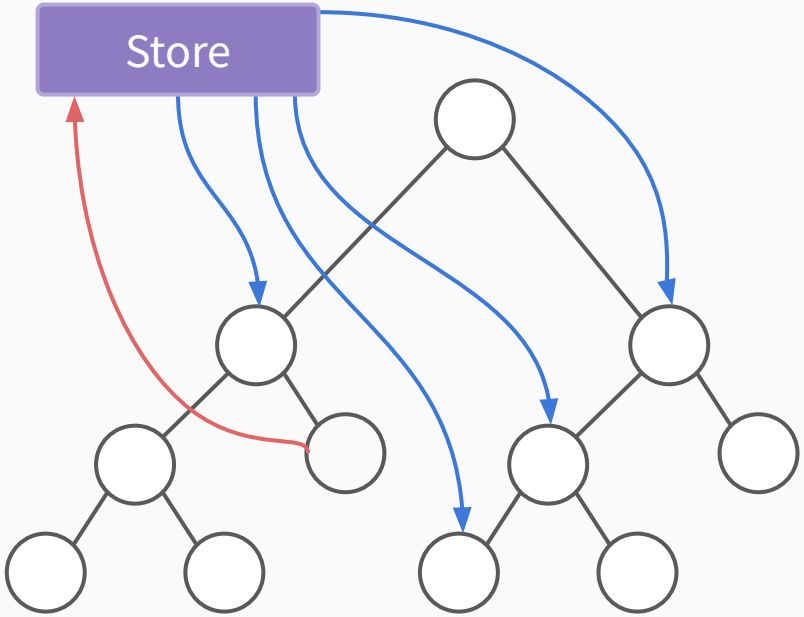
- Alternative UIs while reusing most of the business logic.
- Get undo / redo for free
- Automated bug reports with replay function

Why / What you'll learn



This is how we manage state at the moment

State management with Redux



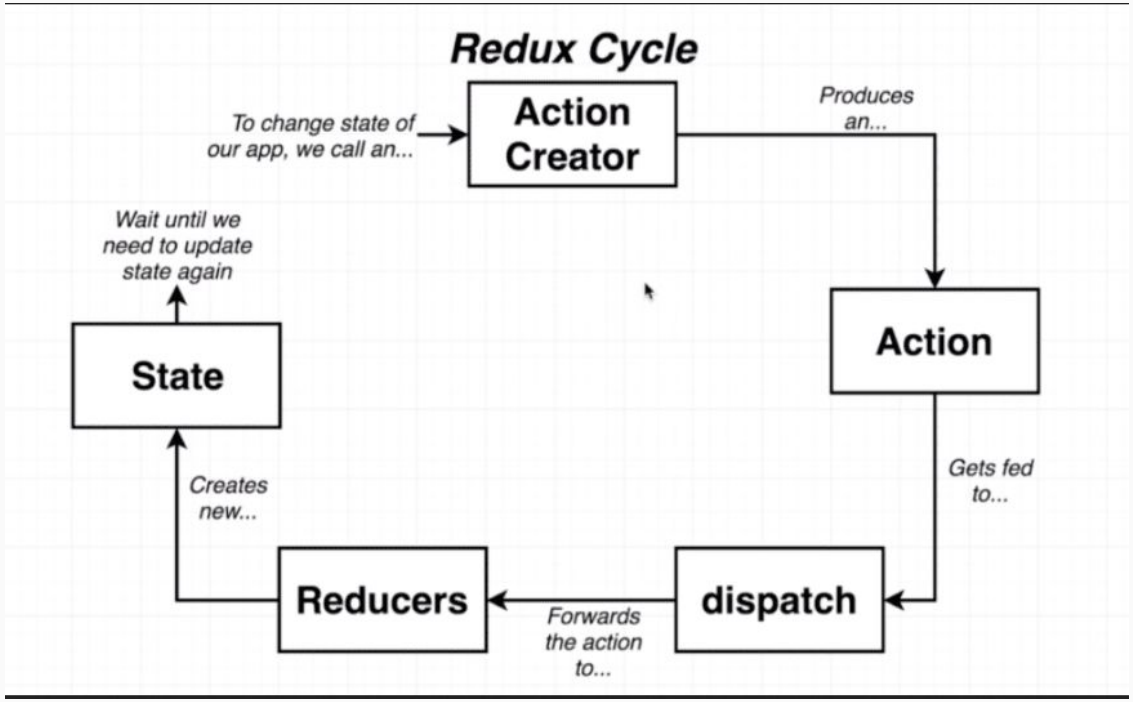
Everything is dispatched from and to **one global store**

State management with Redux

Redux asks you to:

- Describe application state as plain objects and arrays
- Describe changes in the system as plain objects
- Describe the logic for handling changes as *pure functions* *(explanation later)*

State management with Redux



Redux Store

Redux uses a single store to manage everything. The store is just a POJO.

```
{
    dispatch,
    state {
        books,
        colorpicker
    },
    reducer
}
```

*POJO = Plain Old JavaScript Object

Redux Store

Redux uses a single store to manage everything. The store is just a POJO.

```
{
    dispatch,
    state {
        books,
        colorpicker
    },
    reducer
}
```

This is our actual application state.

*POJO = Plain Old JavaScript Object

Actions

Redux Action

<code>

A redux action is an object with a type and an optional payload that describes a state change

```
{
  type: 'ADD_BOOK',
  payload: {
    book: {...}
  }
}
```

Action Creators

Action Creator

<code>

An action creator is a function that returns an action

```
function addBook(book: Book): AddBookAction {
  return {
    type: 'ADD_BOOK',
    payload: {
      book: book
    }
  }
}
```

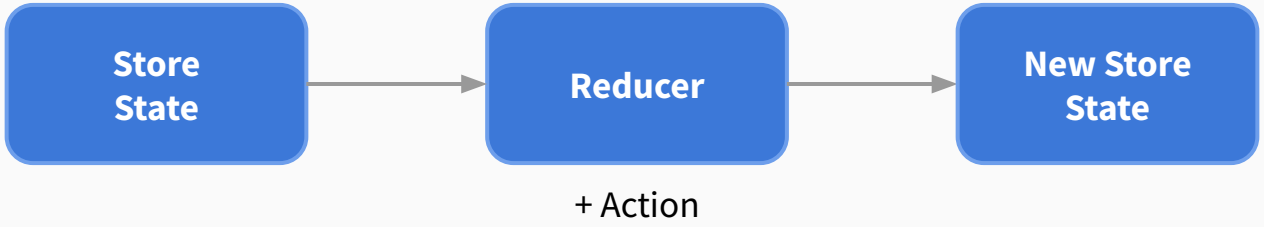
Reducers

A Reducer transfers the

store to another state

Reducers

A reducer takes a state and an action and returns a new state.



Reducer

<code>

A reducer implementing the actual state change for an action type

```
function reducer(state: State = initialState, action: AllPossibleActions): State {
  switch(action.type) {
    case 'ADD_BOOK':
      let newState = { ...state }; // shallow copy of the state
      newState.books = [...state.books, action.book];
      return newState;
    case ...
    default:
      return state;
  }
}
```

Pure functions

A pure function **always** returns

the same output for a given input

Why / What you'll learn

Pure functions

- have no side effects
- are easy to reason about
- are easily testable

Pure function

<code>

This is a pure function

```
function(n) {
  return n * n;
}
```

Pure function

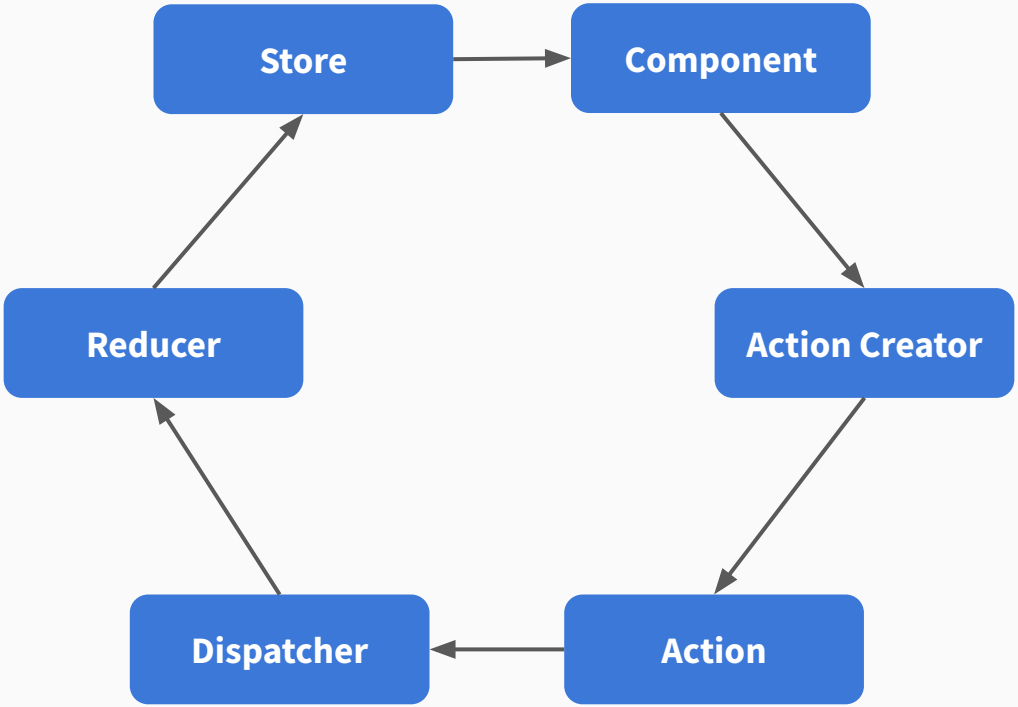
<code>

This is a **NOT** pure function

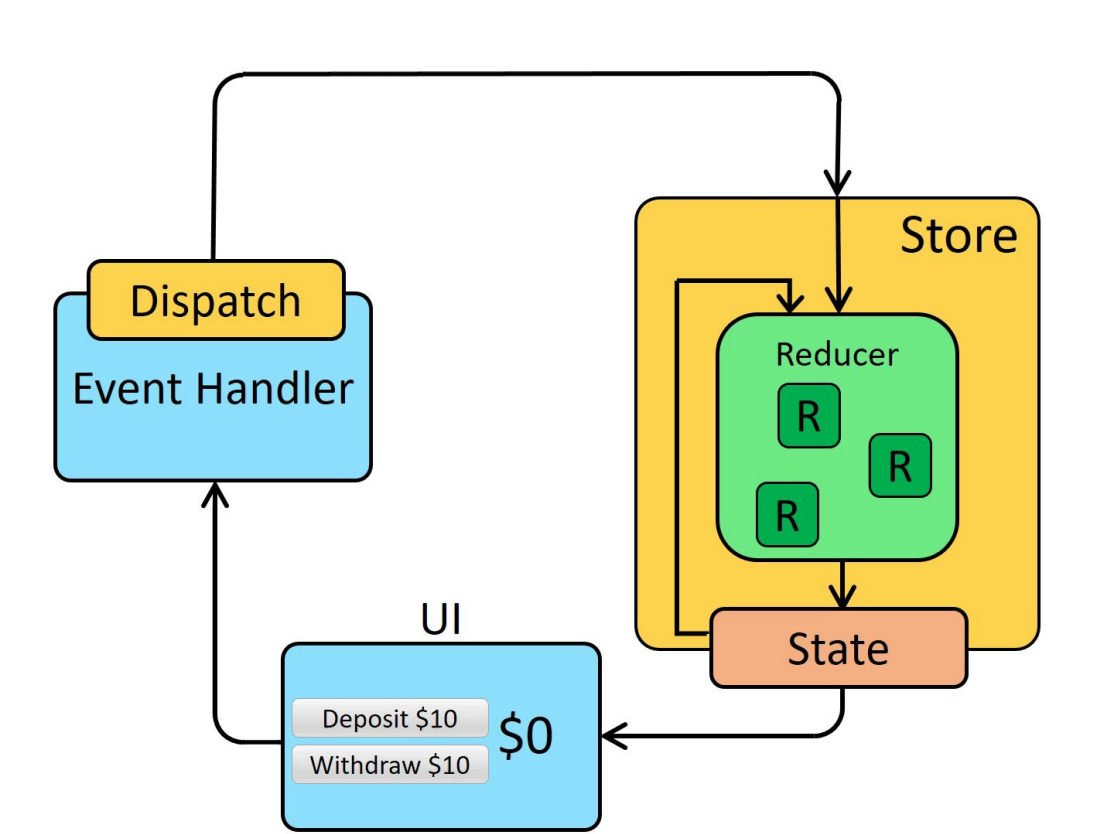
```
function addMinutes(n: number) {
  const now = new Date();
  return now.setMinutes(now.getMinutes() + n);
}
```

Complete Redux cycle

Redux cycle


Store
Component
Action Creator
Action
Dispatcher
Reducer

Redux cycle



GIF under MIT License taken from https://github.com/reduxjs/redux/blob/master/website/static/img/tutorials/essentials/ReduxDataFlowDiagram.gif

Detour

Reducer in React

A reducer-function transfers

one state to another state.

React has a **`useReducer-hook`** to implement

such a pattern

useReducer-hook in React

<code>

The `useReducer`-hook enables more complex state management.

```
interface EmailState { email: string }
const initialState: EmailState = { email: "" }

type ChangeEmailAction = { type: "changeEmail"; payload: string };
type AllActions = ChangeEmailAction | ResetAction ;

const emailReducer = (state: EmailState, action: AllActions): EmailState => {
  switch (action.type) {
    case "changeEmail":
      return { ...state, email: action.payload }
    case "reset":
      return initialState;
  }
};
```

useReducer-hook in React

<code>

The `useReducer`-hook enables more complex state management.

```
interface EmailState { email: string }
const initialState: EmailState = { email: "" }

type ChangeEmailAction = { type: "changeEmail"; payload: string };
type AllActions = ChangeEmailAction | ResetAction ;

const emailReducer = (state: EmailState, action: AllActions): EmailState => {
  switch (action.type) {
    case "changeEmail":
      return { ...state, email: action.payload }
    case "reset":
      return initialState;
  }
};
```

Reducer: Function which takes the current state and an action and returns the new state.

useReducer-hook in React

<code>

The `useReducer`-hook enables more complex state management.

```
interface EmailState { email: string }
const initialState: EmailState = { email: "" }

type ChangeEmailAction = { type: "changeEmail"; payload: string };
type AllActions = ChangeEmailAction | ResetAction ;

const emailReducer = (state: EmailState, action: AllActions): EmailState => {
  switch (action.type) {
    case "changeEmail":
      return { ...state, email: action.payload }
    case "reset":
      return initialState;
  }
};
```

For each action, compute the new state. TypeScript knows the shape of the action from the type defs.

useReducer-hook in React

<code>

The `useReducer`-hook enables more complex state management.

```
interface EmailState { email: string }
const initialState: EmailState = { email: "" }

type ChangeEmailAction = { type: "changeEmail"; payload: string };
type AllActions = ChangeEmailAction | ResetAction ;

const emailReducer = (state: EmailState, action: AllActions): EmailState => {
  switch (action.type) {
    case "changeEmail":
      return { ...state, email: action.payload }
    case "reset":
      return initialState;
  }
};
```

Add a `case`-statement for each action. Omit the `default` so TypeScript tells you which cases are missing.

useReducer-hook in React

<code>

In our component we dispatch actions instead directly changing the state.

```
const EmailForm = () => {
  const [state, dispatch] = useReducer(emailReducer, initialState);

  const handleEmailChange = (event) => {
    dispatch({ type: "changeEmail", payload: event.target.value });
  };

  return <input value={state.email} onChange={handleEmailChange} />;
};
```

useReducer-hook in React

<code>

In our component we dispatch actions instead directly changing the state.

```
const EmailForm = () => {
  const [state, dispatch] = useReducer(emailReducer, initialState);

  const handleEmailChange = (event) => {
    dispatch({ type: "changeEmail", payload: event.target.value });
  };

  return <input value={state.email} onChange={handleEmailChange} />;
};
```

Pass in the reducer-function and its initial state.

useReducer-hook in React

<code>

In our component we dispatch actions instead directly changing the state.

```
const EmailForm = () => {
  const [state, dispatch] = useReducer(emailReducer, initialState);

  const handleEmailChange = (event) => {
    dispatch({ type: "changeEmail", payload: event.target.value });
  };

  return <input value={state.email} onChange={handleEmailChange} />;
};
```

The hook returns the state and a function to dispatch actions.

useReducer-hook in React

<code>

In our component we dispatch actions instead directly changing the state.

```
const EmailForm = () => {
  const [state, dispatch] = useReducer(emailReducer, initialState);

  const handleEmailChange = (event) => {
    dispatch({ type: "changeEmail", payload: event.target.value });
  };

  return <input value={state.email} onChange={handleEmailChange} />;
};
```

Read from the state.

useReducer-hook in React

<code>

In our component we dispatch actions instead directly changing the state.

```
const EmailForm = () => {
  const [state, dispatch] = useReducer(emailReducer, initialState);

  const handleEmailChange = (event) => {
    dispatch({ type: "changeEmail", payload: event.target.value });
  };

  return <input value={state.email} onChange={handleEmailChange} />;
};
```

Trigger updates by dispatching a new action.

Install Redux

How to install redux

- Not included in create-react-app
- Install it via npm

```
npm install --save redux react-redux
```

Redux contains the actual implementation of Redux.

(Similar as “React” is the pure React logic.)

React-redux contains the logic to connect Redux with React.

(Similar as “ReactDOM” is the renderer for React to the DOM.)

Use `@reduxjs/toolkit`

- *“The official, opinionated, batteries-included toolset for efficient Redux development”*
- Install it via npm

```
npm install --save @reduxjs/toolkit react-redux
```

The toolkit contains Redux and additional functions to simplify the work with Redux and to reduce boilerplate.

Simplest Store

<code>

The simplest store is just a function without any actions

```
import { configureStore, combineReducers } from "@reduxjs/toolkit";

const initialState = {}
const books = (state =initialState) => state

const rootReducer = combineReducers({ books });

const store = configureStore({ reducer: rootReducer });

export default store;
```

Simplest Store

<code>

The simplest store is just a function without any actions

```
import { configureStore, combineReducers } from "@reduxjs/toolkit";

const initialState = {}
const books = (state =initialState) => state

const rootReducer = combineReducers({ books });

const store = configureStore({ reducer: rootReducer });

export default store;
```

Simplest Store

<code>

The simplest store is just a function without any actions

```
import { configureStore, combineReducers } from "@reduxjs/toolkit";

const initialState = {}
const books = (state =initialState) => state

const rootReducer = combineReducers({ books });

const store = configureStore({ reducer: rootReducer });

export default store;
```

Simplest Store

<code>

The simplest store is just a function without any actions

```
import { configureStore, combineReducers } from "@reduxjs/toolkit";

const initialState = {}
const books = (state = initialState) => state

const rootReducer = combineReducers({ books });

const store = configureStore({ reducer: rootReducer });

export default store;
```

Create and Provide your store

<code>

Use <Provider /> to make the Redux store available in your app

```
import {Provider} from 'react-redux'
import store from './store'

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

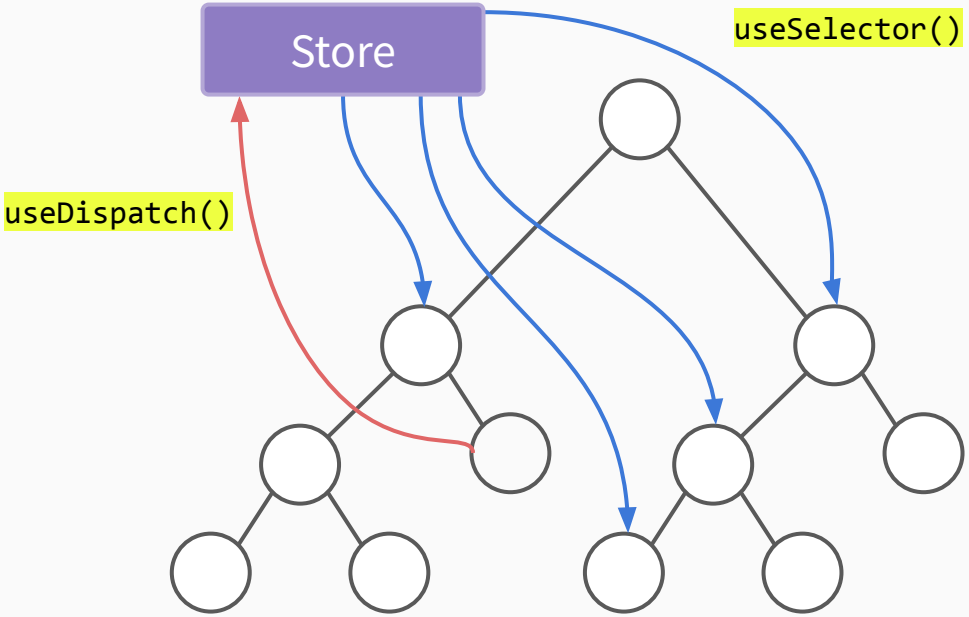
Task

Install Redux and create store

Connect a component to your store

Modern Pattern: `useDispatch()` and `useSelector()`

Redux in your Components



1. Changes to the store are made via dispatched actions

2. State changes are mapped to props of a component

Redux in your Components

<code>

You can use the new hooks to create a connection to the Redux store.

```
import { useDispatch, useSelector } from "react-redux";

// usage in your component
const dispatch = useDispatch();
const books = useSelector(state => state.books);

return (
 <>
    { books.map(book => <BookDetails book={book} key={book.isbn} />)}
    <button onClick={() => dispatch(resetBooks())}>Reset books</button>
  </>
);
```

Redux in your Components

<code>

`useDispatch` replaces `mapDispatchToProps` of the `connect-HOC`.

```
import { useDispatch, useSelector } from "react-redux";

// usage in your component
const dispatch = useDispatch();
const books = useSelector(state => state.books);

return (
 <>
    { books.map(book => <BookDetails book={book} key={book.isbn} />)}
    <button onClick={() => dispatch(resetBooks())}>Reset books</button>
  </>
);
```

Get access to the dispatch function and use it to dispatch an action to the Redux-store (created with an action creator).

Redux in your Components

<code>

`useSelector` replaces `mapStateToProps` of the `connect-HOC`.

```
import { useDispatch, useSelector } from "react-redux";

// usage in your component
const dispatch = useDispatch();
const books = useSelector(state => state.books);

return (
 <>
    { books.map(book => <BookDetails book={book} key={book.isbn} />)}
    <button onClick={() => dispatch(resetBooks())}>Reset books</button>
  </>
);
```

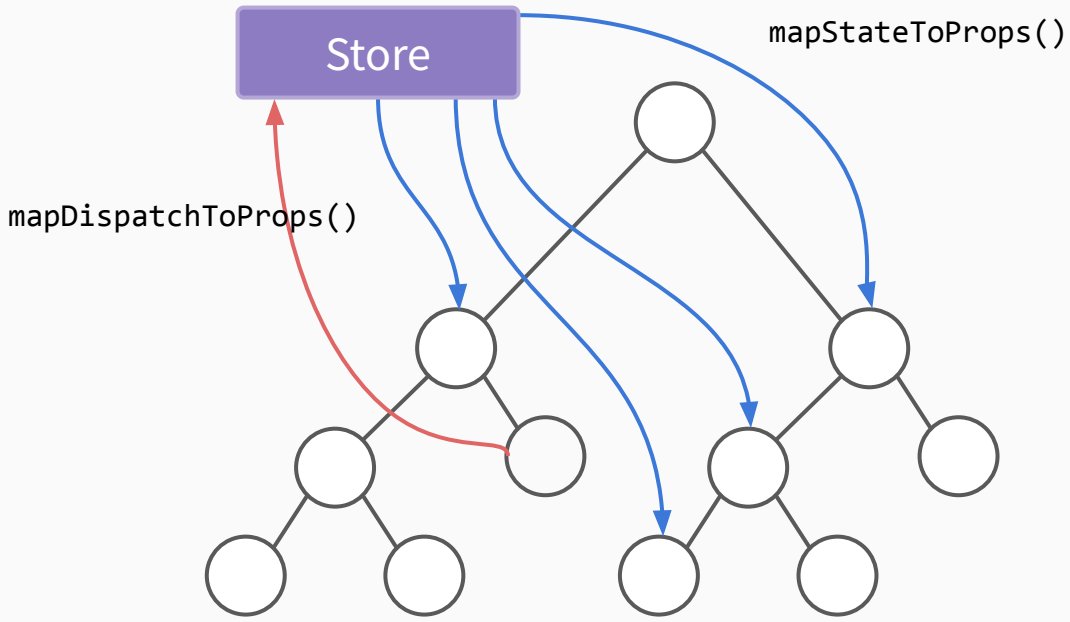
Retrieve data from your state inside your component.

Alternative Pattern: The `connect()` decorator

Why / What you'll learn

- How to use connect()
- What are the mapStateToProps and mapDispatchToProps functions
- What is a decorator function

Repeat: State management with Redux



1. Changes to the store are made via dispatched actions

2. State changes are mapped to props of a component

What is a decorator?

- Design pattern
- Also known as wrapper
- Adds behavior to an individual object

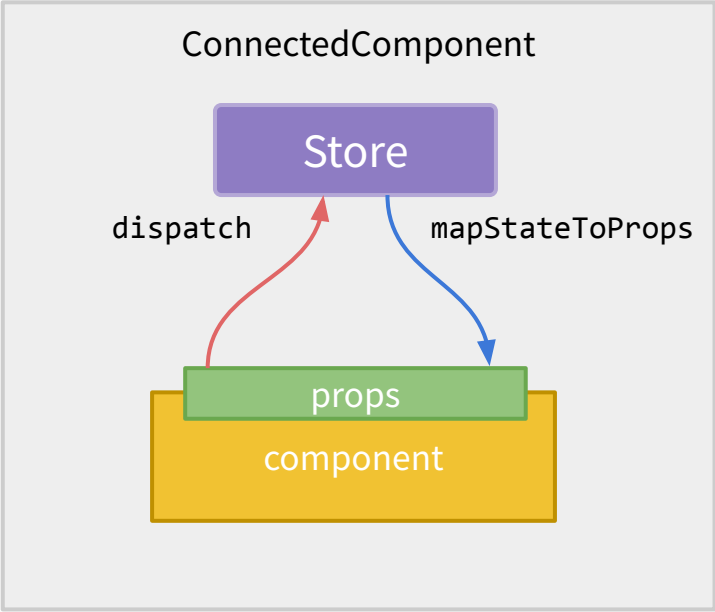
What is the connect() decorator?

- Connects a React component to a Redux store
- Providing a convenient API for the most common use cases
- Wrapping component is also called a *Higher Order Component*

How to connect a component to your store

Use the connect decorator

```
import {connect} from 'react-redux'
const ConnectedComponent = connect(
  (state) => {},
  (dispatch) => {}
)(Component);
```



Write your mapStateToProps function

<code>

Define how to **map the state to** your component **props**

```
connect(
  (state) => { return {books : state.books} },
  (dispatch) => {
    return {
      onBookSelected : (book) => {
        return dispatch(selectBook(book))
      }
    }
  }
)(BookList);
```

Write your dispatch function

<code>

Define how to **dispatch actions** from your component

```
connect(
  (state) => { return {books : state.books} },
  (dispatch) => {
    return {
      onBookSelected : (book) => {
        return dispatch(selectBook(book))
      }
    }
  }
)(BookList);
```

Write your dispatch function

<code>

You can write explicit functions for easier understanding.

```
const mapStateToProps = (state) => {
    return { books: state.books };
}

const mapDispatchToProps = (dispatch) => {
    return { onBookSelected: (book) => { dispatch(bookSelected(book))} };
}

BookList = connect(mapStateToProps, mapDispatchToProps)(BookList);
```

Write your dispatch function with ease

<code>

You can provide mapDispatchToProps as object.

```
// Default
const mapDispatchToProps = (dispatch) => {
    return { onBookSelected: (book) => { dispatch(bookSelected(book))} };
}

// Easier and does the same thing
const mapDispatchToProps = {
    onBookSelected: bookSelected
};
```

Don’t mix patterns.

- It’s advised to not mix patterns. Either use hooks or the connect-function in one component to make it easier to understand where data is coming from.
- Using the connect-function might make your component easier to test or reuse, as the component itself receives all data as props.
- Using hooks follows the generally preferred pattern of composition.

Recap: Action Creator

<code>

We can use an action creator to create our action to dispatch to Redux.

```
function addBooks(books: Book[]): AddBooksAction {
  return {
    type: 'addBooks',
    payload: books
  }
}
```

Typesafe action creator with Redux Toolkit <code>

Redux Toolkit provides utilities to create typesafe actions for us.

```
import { createAction } from "@reduxjs/toolkit";

const addBooks = createAction<Book[]>("addBooks");
```

Typesafe action creator with Redux Toolkit <code>

Redux Toolkit provides utilities to create typesafe actions for us.

```
import { createAction } from "@reduxjs/toolkit";

const addBooks = createAction<Book[]>("addBooks");
```

The type of our payload.

The type of our action.

Redux Slice

Use createSlice to create a reducer and action

Redux Toolkit provides utilities to create typesafe actions and reducers in combination.

```
import { PayloadAction, createSlice } from "@reduxjs/toolkit";

type AddBooksAction = PayloadAction<Book[]>;

const initialState: { books: Book[] } = { books };

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: {
    addBooks(state, action: AddBooksAction) {
      state.books = action.payload
    },
  },
});

export const { addBooks } = booksSlice.actions;
export default booksSlice.reducer;
```

Use createSlice to create a reducer and action

Redux Toolkit provides utilities to create typesafe actions and reducers in combination.

```
import { PayloadAction, createSlice } from "@reduxjs/toolkit";

type AddBooksAction = PayloadAction<Book[]>;

const initialState: { books: Book[] } = { books };

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: {
    addBooks(state, action: AddBooksAction) {
      return { ...state, books: action.payload }
    },
  },
});

export const { addBooks } = booksSlice.actions;
export default booksSlice.reducer;
```

This is the namespace for all our actions. Our actions will later have a type "`books/<ACTION_NAME>`"

Use createSlice to create a reducer and action

Redux Toolkit provides utilities to create typesafe actions and reducers in combination.

```
import { PayloadAction, createSlice } from "@reduxjs/toolkit";

type AddBooksAction = PayloadAction<Book[]>;

const initialState: { books: Book[] } = { books: [] };

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: {
    addBooks(state, action: AddBooksAction) {
      return { ...state, books: action.payload }
    },
  },
});

export const { addBooks } = booksSlice.actions;
export default booksSlice.reducer;
```

We define all our actions directly under the reducers-key. The name of the function is the name of the action creator. The argument type describes how we can later dispatch it.

Use createSlice to create a reducer and action

Redux Toolkit provides utilities to create typesafe actions and reducers in combination.

```
import { PayloadAction, createSlice } from "@reduxjs/toolkit";

type AddBooksAction = PayloadAction<Book[]>;

const initialState: { books: Book[] } = { books: [] };

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: {
    addBooks(state, action: AddBooksAction) {
      return { ...state, books: action.payload }
    },
  },
});

export const { addBooks } = booksSlice.actions;
export default booksSlice.reducer;
```

Every slice exposes all actions and a reducer. The reducer has to be wired up when creating our store, the actions can be used in our application.

Use action creator from our slice

<code>

Use the action creator to dispatch an action to the store

```
// in src/store/books.ts
type AddBooksAction = PayloadAction<Book[]>;
export const { addBooks } = booksSlice.actions;
```

```
// in our component we import the action
import { addBooks } from "./store/books.ts"

// in our component we can use the action
const dispatch = useDispatch()

useEffect(() => {
  fetchBooks().then((books: Book[]) => {
    dispatch(addBooks(books));
  })
}, [])
```

Use action creator from our slice

<code>

Use the action creator to dispatch an action to the store

```
// in src/store/books.ts
type AddBooksAction = PayloadAction<Book[]>;
export const { addBooks } = booksSlice.actions;
```

```
// in our component we import the action
import { addBooks } from "./store/books.ts"

// in our component we can use the action
const dispatch = useDispatch()

useEffect(() => {
  fetchBooks().then((books: Book[]) => {
    dispatch(addBooks(books));
  })
}, [])
```

The type of the payload we pass into our action creator depends on how we typed it initially.

Task

Redux count slice

Selectors allow us to read data from the state.

They are simple functions taking the current state and returning a specific property of interest.

Simple selector

<code>

Create a simple function to extract data from the root state

```
// in src/store/index.ts
export type RootState = ReturnType<typeof rootReducer>;

// in src/store/selectors.ts
import { RootState } from "./index"

export const getBookIsbns = (state: RootState) =>
  state.books.map(book => book.isbn);

export const getBooksFromFowler = (state: RootState) =>
  state.books.filter(book => book.author === "Martin Fowler");
```

```
{
  books: [
    { title: "Martin Fowler" },
    // … other books ...
  ]
}
```

Simple selector

<code>

Create a simple function to extract data from the root state

```
// in src/store/index.ts
export type RootState = ReturnType<typeof rootReducer>;

// in src/store/selectors.ts
import { RootState } from "./index"

export const getBookIsbns = (state: RootState) =>
  state.books.map(book => book.isbn);

export const getBooksFromFowler = (state: RootState)=>
  state.books.filter(book => book.author === "Martin Fowler");
```

We need the shape of our state.

Simple selector

<code>

Create a simple function to extract data from the root state

```
// in src/store/index.ts
export type RootState = ReturnType<typeof rootReducer>;

// in src/store/selectors.ts
import { RootState } from "./index"

export const getBookIsbns = (state: RootState) =>
  state.books.map(book => book.isbn);

export const getBooksFromFowler = (state: RootState) =>
  state.books.filter(book => book.author === "Martin Fowler");
```

Take the root state and create a specific view on it, here the list of all ISBNs or a list of all books by a specific author.

Using a selector with hooks

<code>

Use the `useSelector` hook to execute the selector and use the data

```
import { useSelector } from "react-redux";
import { getBookIsbns } from "./store/selectors";

function IsbnList() {
  const isbns = useSelector<RootState, string[]>(getBookIsbns);

  return (
    <ul>{isbns.map(isbn => <li key={isbn}>{isbn}</li>)}</ul>
  );
}

export default Books;
```

Using a selector with hooks

<code>

Use the `useSelector` hook to execute the selector and use the data

```
import { useSelector } from "react-redux";
import { getBookIsbns } from "./store/selectors";

function IsbnList() {
  const isbns = useSelector<RootState, string[]>(getBookIsbns);

  return (
    <ul>{isbns.map(isbn => <li key={isbn}>{isbn}</li>)}</ul>
  );
}

export default Books;
```

Import the `useSelector`-hook from react-redux.

Using a selector with hooks

<code>

Use the `useSelector` hook to execute the selector and use the data

```
import { useSelector } from "react-redux";
import { getBookIsbns } from "./store/selectors";

function IsbnList() {
  const isbns = useSelector<RootState, string[]>(getBookIsbns);

  return (
    <ul>{isbns.map(isbn => <li key={isbn}>{isbn}</li>)}</ul>
  );
}

export default Books;
```

Import your selector-function and pass it as first argument to the `useSelector`-hook.

Using a selector with hooks

<code>

Use the `useSelector` hook to execute the selector and use the data

```
import { useSelector } from "react-redux";
import { getBookIsbns } from "./store/selectors";

function IsbnList() {
  const isbns = useSelector<RootState, string[]>(getBookIsbns);

  return (
    <ul>{isbns.map(isbn => <li key={isbn}>{isbn}</li>)}</ul>
  );
}

export default Books;
```

Use the return value like any other value in your component.

Using a selector with hooks

<code>

Use the `useSelector` hook to execute the selector and use the data

```
import { useSelector } from "react-redux";
import { getBookIsbns } from "./store/selectors";

function IsbnList() {
  const isbns = useSelector<RootState, string[]>(getBookIsbns);

  return (
    <ul>{isbns.map(isbn => <li key={isbn}>{isbn}</li>)}</ul>
  );
}

export default Books;
```

Type of the root state

Type of the return value of our selector and type of the variable

We need to help the TypeScript compiler and type the selector.

We should treat our “state” as database and keep data in a **normalized shape**.

Selectors are like queries, allowing to retrieve data in the shape we need them in our app.

Task

Redux books slice

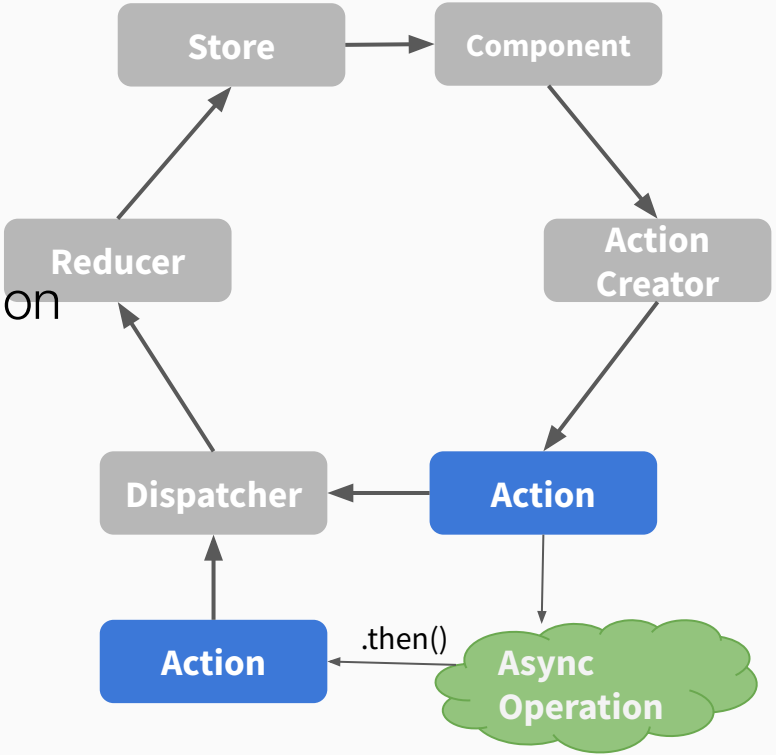
Redux with Async Actions

To handle asynchronous tasks, we either have to orchestrate our actions from our components or use a middleware like Redux Thunk.

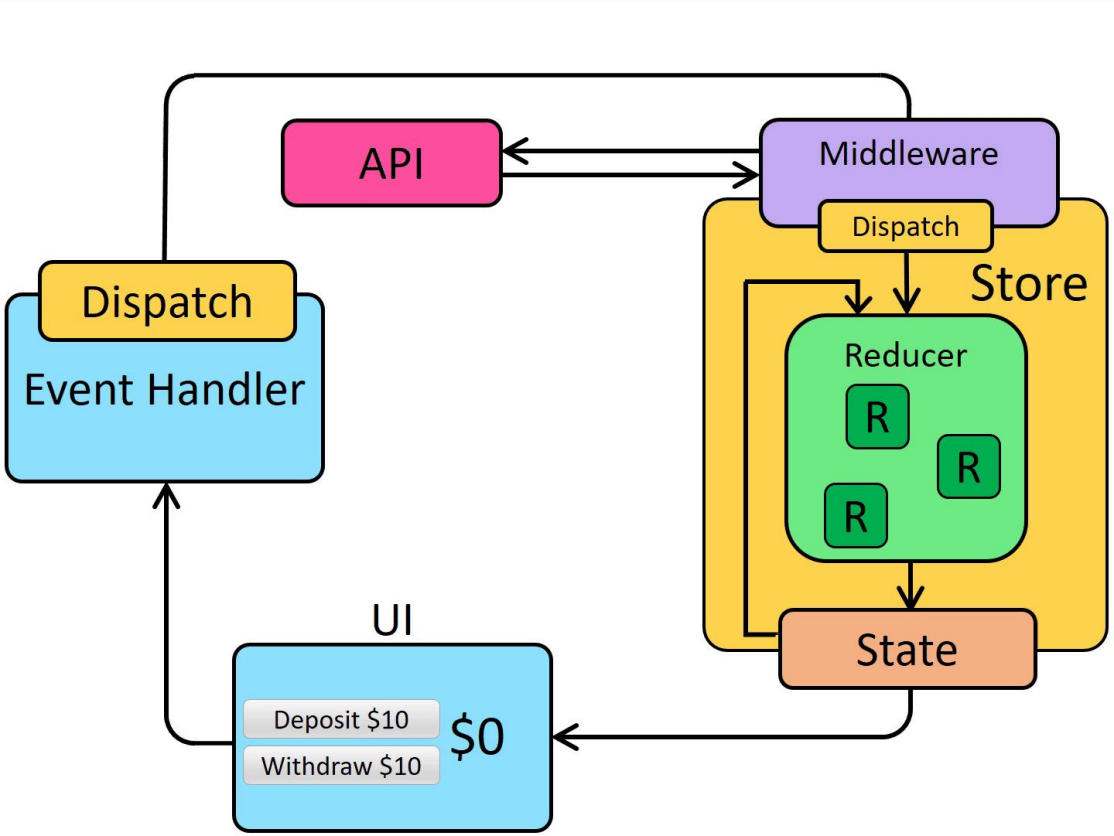
`@reduxjs/toolkit` includes `redux-thunk` by default.

Redux cycle with async actions

- Actions are synchronous
- First actions triggers an async function
- Async function returns a promise
- The promise resolves and triggers another action



Redux cycle with async actions



GIF under MIT License taken from https://github.com/reduxjs/redux/blob/master/website/static/img/tutorials/essentials/ReduxAsyncDataFlowDiagram.gif

Creating an async thunk

<code>

Use the `createAsyncThunk` util to create your thunk.

Type of the return value

Type of the thunk-argument

Type of the thunk-API

```
export const fetchBooks = createAsyncThunk<Book[], void, {}>(
  "books/fetchBooks",
  async () => {
    const response = await fetch("http://localhost:4730/books");
    const books = await response.json();

    return books;
  }
);
```

Creating an async thunk

<code>

Use the `createAsyncThunk` util to create your thunk.

```
export const fetchBooks = createAsyncThunk<Book[], void, {}>(
  "books/fetchBooks",
  async () => {
    const response = await fetch("http://localhost:4730/books");
    const books = await response.json();

    return books;
  }
);
```

The name of our thunk-action. This will be the prefix used for all sub-actions, like `prefix/fulfilled`.

Creating an async thunk

<code>

Use the `createAsyncThunk` util to create your thunk.

```
export const fetchBooks = createAsyncThunk<Book[], void, {}>(
  "books/fetchBooks",
  async () => {
    const response = await fetch("http://localhost:4730/books");
    const books = await response.json();

    return books;
  }
);
```

Our asynchronous logic – we perform all our side effects and async tasks and return our final value. If required, we have access to the thunk-API like `getState` or `dispatch`.

Creating an async thunk

<code>

Use the `createAsyncThunk` util to create your thunk.

```
export const fetchBooks = createAsyncThunk<Book[], void, {}>(
  "books/fetchBooks",
  async (thunkArg, thunkApi) => {
    const response = await fetch("http://localhost:4730/books");
    const books = await response.json();

    return books;
  }
);
```

Our asynchronous logic – we perform all our side effects and async tasks and return our final value. If required, we have access to the thunk-API like `getState` or `dispatch`.

States of an async thunk

<code>

An async thunk dispatches multiple sub-actions to the store depending on its current state. We can react to each of them in our reducer.

```
export const fetchBooks = createAsyncThunk();

// the type of the thunk when it’s dispatched first but hasn’t finished yet
fetchBooks.pending;
// the type of the thunk when it’s finished without an error
fetchBooks.fulfilled;
// the type of the thunk when it’s finished with an error
fetchBooks.rejected;
```

Handling our async thunk in our slice

<code>

We need to handle our async thunk in the slice via the `extraReducers`.

```
export const fetchBooks = createAsyncThunk();

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: { /* our normal reducers */ },
  extraReducers: (builder) => {
    builder.addCase(fetchBooks.fulfilled, (state, action) => {
      // our case-reducer for the fetchBooks-success-action
    })
  },
});
```

In our reducers, we can only handle actions which are defined there as well.

Handling our async thunk in our slice

<code>

We need to handle our async thunk in the slice via the `extraReducers`.

```
export const fetchBooks = createAsyncThunk();

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: { /* our normal reducers */ },
  extraReducers: (builder) => {
    builder.addCase(fetchBooks.fulfilled, (state, action) => {
      // our case-reducer for the fetchBooks-success-action
    })
  },
});
```

We need to use the `extraReducers` for all additional actions we want to handle.

Handling our async thunk in our slice

<code>

We need to handle our async thunk in the slice via the `extraReducers`.

```
export const fetchBooks = createAsyncThunk();

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: { /* our normal reducers */ },
  extraReducers: (builder) => {
    builder.addCase(fetchBooks.fulfilled, (state, action) => {
      // our case-reducer for the fetchBooks-success-action
    })
  },
});
```

Use the `builder` to create a case for every additional action we want to handle.

Handling our async thunk in our slice

<code>

We need to handle our async thunk in the slice via the `extraReducers`.

```
export const fetchBooks = createAsyncThunk();

const booksSlice = createSlice({
  name: "books",
  initialState,
  reducers: { /* our normal reducers */ },
  extraReducers: (builder) => {
    builder.addCase(fetchBooks.fulfilled, (state, action) => {
      // our case-reducer for the fetchBooks-success-action
    })
  },
});
```

Add a case for the success action of our thunk and handle it respectively.

Task

Create an async action

There are many middlewares available for Redux:

- For handling async tasks:
  - Redux Thunk
  - Redux-Saga
  - redux-observable
- …and many many more for many different use cases!

We teach.

workshops.de