React & Typescript

Redux, Context, Hooks

Joshua Anthony

Microsoft Certified Trainer

Technical founder - Logwise (startup).

20+ years web development & consulting.

Who are you?

Agenda

Agenda

React

React & State

React & Redux

Reading Redux state

Triggering Actions

Reducing into state

Redux Middleware

Functional programming concepts

Hooks

Component Lifecycle

- Classes vs Hooks

Context

Composition

Building tools & webpack

Principles of React

UI component library

Composable components defined as TS/JS classes or functions

TS/JS classes are actually functions

You are not in control - React calls your code (functions) on it's own schedule

Pragmatic - prefers declarative constructs inspired by functional programming

But designed for real world use and supports imperative escape hatches

Target developers of all skill levels

Architectural Principles of React

Virtual dom

Unidirectional data flow

Lift state up & Push functions down

Principles of React

Virtual dom

In-memory representation of component hierarchy and document structure

Faster performance by only changing Browser DOM when necessary

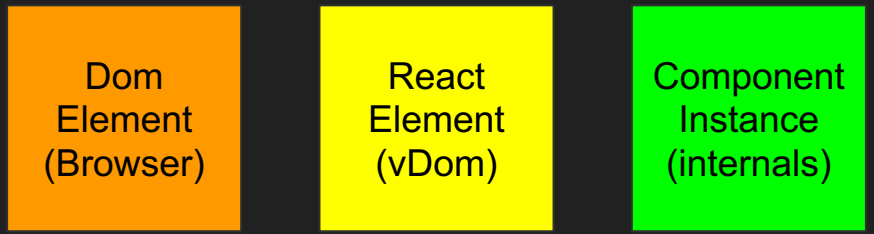
Supports declarative UI without instance references

React maps elements to actual component instances

Principles of React

Unidirectional data flow
Partly an effect of the vDOM

Every TSX/JSX element is turned into a call to createElement with a component name. React joins the behavior of the component with the actual element.



Components communicate through props

Principles of React

Lift state up & Push functions down

Since props are the primary way that components communicate with each other and props only flow downwards - functions as props is required.

Coordinating internal state between components is not possible - requires state elsewhere.

Challenges

React is a UI framework - components and layout always go together

Storing state in components means that state is bound the UI structure.

Since props are the primary way that components communicate with each other and props only flow downwards - every component needs to have all child props also.

Coordinating internal state between components is not possible - requires state elsewhere (lifted up).

Benefits of State & Props

Deeply integrated into React

All layout changes triggered by state or props changes

Easy model to understand.

Demo/Lab shared "state" with State & Props

Quick run through of code

Task: Add a clickable element under "load more" to empty the shopping cart.

React + Redux

Shared SPA state

Redux

State Management Library

State changes can be requested anywhere in an application

All state changes made in a single location

Redux

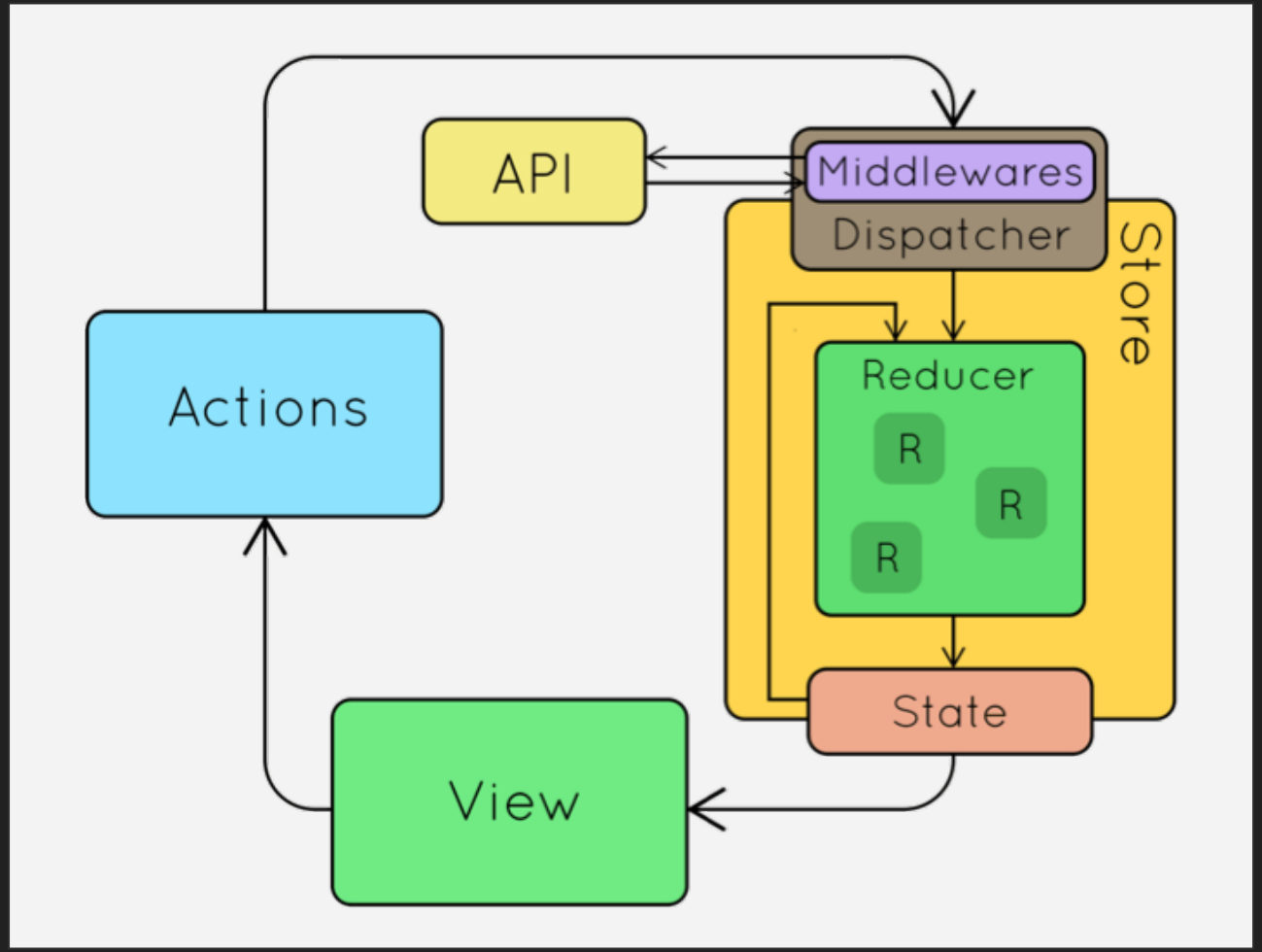
Store – where the application state is kept

Actions – objects that are used to request state changes

Reducers – code that implements changes based on actions

Updates to the store trigger re-render of components
(just like updates to React state)

Redux


API
Middlewares
Dispatcher
Store
Actions
Reducer
R
R
R
State
View

Demo

Store – where the application state is kept

Actions – objects that are used to request state changes

Reducers – code that implements changes based on actions

Updates to the store trigger re-render of components
(just like updates to React state, but using magic)

Redux - store

Redux will manage the store for us…

```
const store = createStore(rootReducer)
```

Redux - store

But we do need to use the store by wrapping our application in a Redux provider

```
<Provider store={store}>
  <App></App>
</Provider>
```

Redux – connecting to Redux

Export the connected component

```
export default connect(
    mapStateToProps,
    mapDispatchToProps
)(ComponentName);
```

Redux – connecting to Redux

Shape your props: `mapStateToProps`

```
const mapStateToProps = (state, ownProps) => {
    return {
        ...ownProps,
        propName1: state.prop1,
        propName2: state.prop2
    }
};
```

Redux – connecting to Redux

Add dispatch function to props: `mapDispatchToProps`

```
const mapDispatchToProps = (dispatch, ownProps) => ({
    action1: () => dispatch( reduxActionCreator1() ),
    action2: (param1, param2) =>
        dispatch( reduxActionCreator1(param1, param2) )
});
```

Redux – connecting to Redux with Typescript

```
const mapState = (state: AppState, ownprops: OwnProps) => ({
    reduxstate: state.cart.totalCost + ownprops.product.price
})

const mapDispatch ={
    addProductToCart: (product: types.ProductItem) => addProduct(product)
}

const connector = connect(
    mapState,
    mapDispatch
)

export default connector(Product);
```

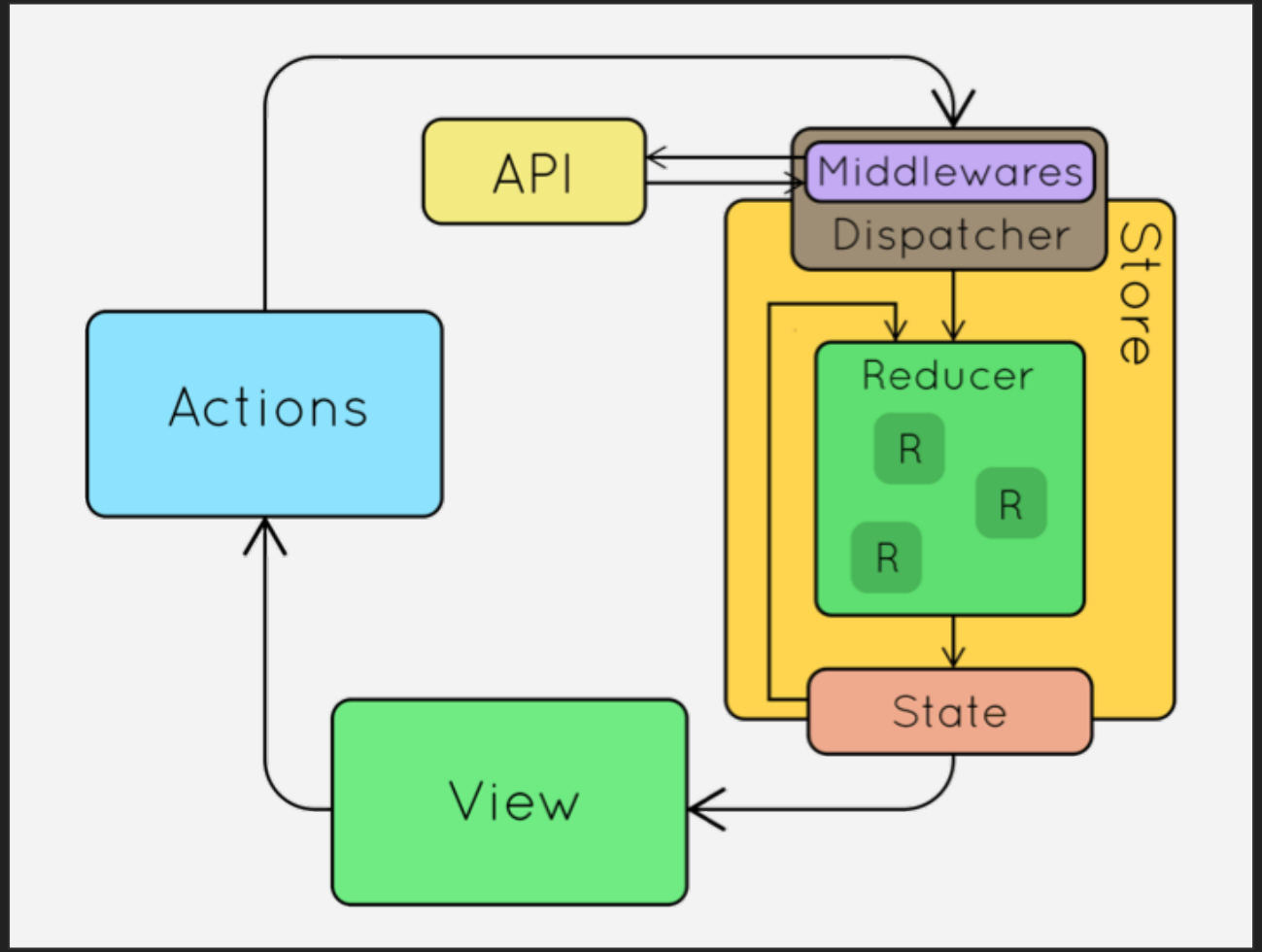
Redux – connecting to Redux with TS

```
type Props = OwnProps & PropsFromRedux

type OwnProps = {
    product: types.ProductItem,
    onClick?: (product: types.ProductItem) => void
}

type PropsFromRedux = ConnectedProps<typeof connector>

class Product extends Component<Props> {
```

Demo / Lab - Receiving State

Task: Read from state into the CartInfo component.

Redux


API
Middlewares
Dispatcher
Store
Actions
Reducer
R
R
R
State
View

Redux - actions

Describe user intent (can include data)

Must have a `type` property

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

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

Redux – action creators

Functions which make creating actions easier.

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

Redux – dispatching actions

Trigger actions using `dispatch(action)`

```
dispatch(addTodo(text))
```

Redux - Actions in TS

```
export const ADD_PRODUCT = 'SHOPPINGCART.ADD_PRODUCT'
export const EMPTY_CART = 'SHOPPINGCART.EMPTY_CART'

interface AddProductAction {
  type: typeof ADD_PRODUCT
  product: CartProduct
}

interface EmptyCartAction {
  type: typeof EMPTY_CART
}

export type CartActionTypes = AddProductAction | EmptyCartAction
```

Redux - Actions in TS

```
export function addProduct(newProduct: CartProduct): CartActionTypes {
    return {
        type: ADD_PRODUCT,
        product: newProduct
    }
}

export function emptyCart(): CartActionTypes {
    return {
        type: EMPTY_CART
    }
}
```

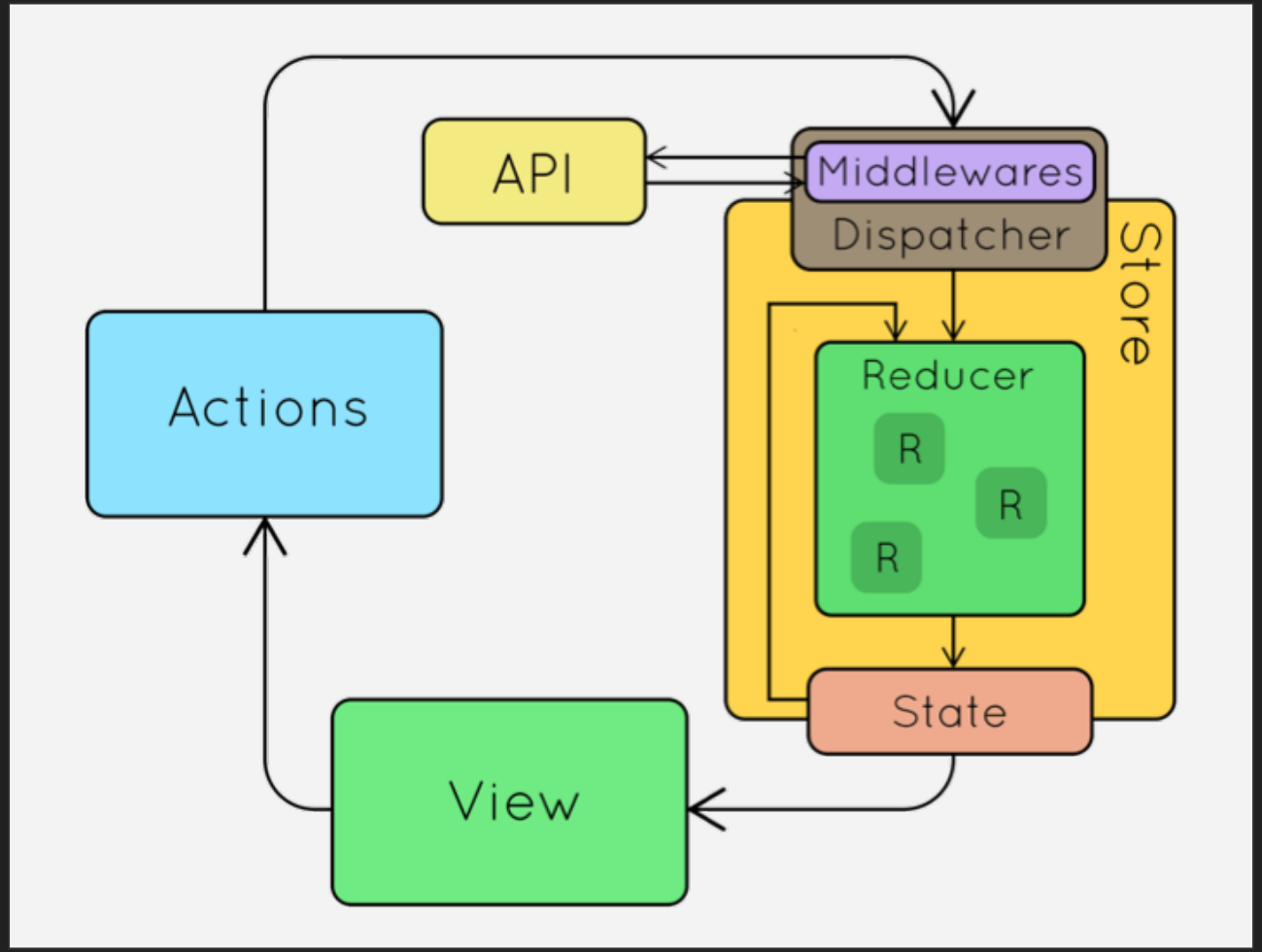
Redux - Actions in TS

```
const mapDispatch ={
    addProductToCart: (product: types.ProductItem) => addProduct(product)
}

const connector = connect(
    mapState,
    mapDispatch
)

export default connector(Product);
```

Demo/labb - dispatch an action

Task: Make the "Buy Now" text add the product to the cart state.

Redux


API
Middlewares
Dispatcher
Store
Actions
Reducer
R
R
R
State
View

Redux - reducers

Pure functions (cannot have side effects) which create a new Redux state

Old state as input parameter - returns the new state

Must create new copies of objects that have been changed…

For example using the spread operator `{…oldObj}`

Redux - reducers

```
let initialState = [];

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

Redux - reducers

Multiple reducers can be used for different aspects of state using `combineReducers`

```
export default combineReducers({
  todos,
  visibilityFilter
})
```

Redux - reducers in TS

```
export function cartReducer(state = initialState, action: CartActionTypes): CartState {
    switch (action.type) {
        case ADD_PRODUCT:
            return {
                items: [...state.items, action.product],
                totalCost: state.totalCost + action.product.price
            };
        case EMPTY_CART:
            return initialState;
        default:
            return state;
    }
}
```

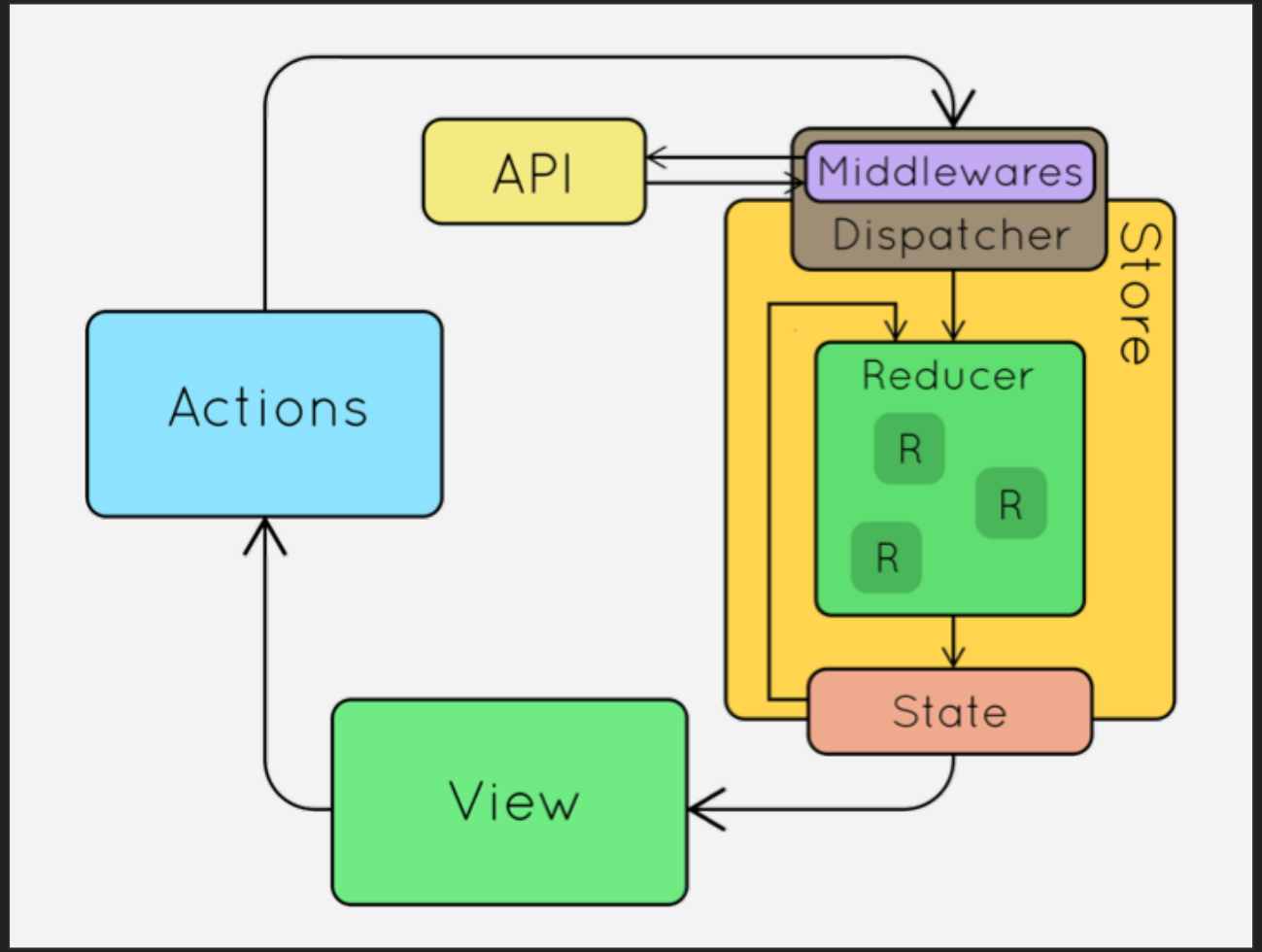
Redux - reducers in TS

```
export const rootReducer = combineReducers({
  cart: cartReducer,
  products: productReducer,
  userPreferences: preferencesReducer
})

export type AppState = ReturnType<typeof rootReducer>

export default function configureStore() {
  const store = createStore(
    rootReducer
  );
  return store;
}
```

Demo / Lab - combine it all

Task: Create a new reducer for tracking the filter state (Veg/Fruit), when the filter changes add an image to the bottom of the page of that category.

Redux


API
Middlewares
Dispatcher
Store
Actions
Reducer
R
R
R
State
View

Redux Middleware

Provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.

Primarily useful for cross-cutting concerns not application logic

i.e: logging, async callbacks, routing, etc

Alternatives to middleware - work - not beautiful

Add the code everywhere - not practical to remember, lots of impact if we need to change it

Wrap a function and call that everywhere - less work but still not automatic

Monkey patch - replace implementation with higher order function

Redux Middleware

Standardised way

Function chaining without needing to know what else is configured

Your middleware:

Function that performs the logic

Function for chaining dispatch

Function for registering on the store

Redux Middleware

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}
```

Redux Middleware in TS

```
export const logger: Middleware =
    (store: MiddlewareAPI<Dispatch<CartActionTypes>, AppState>) =>
        (next: Dispatch<CartActionTypes>) =>
            (action: CartActionTypes) => {
                console.log('dispatching', action)
                let result = next(action)
                console.log('next state', store.getState())
                return result
            }
```

Code along

Add logging to Shop application