REDUX

CORE CONCEPTS

Naveen Pete

Agenda

- Why Redux?
- Core Concepts
- Q & A

Why Redux?

- What is Redux?
  - A predictable state container for JavaScript apps
  - A JavaScript library used to manage an application’s front-end state
  - Created by Dan Abramov
  - Inspired by Flux architecture

- Goal of Redux
  - Make state management in an app more predictable and manageable

- Benefits
  - Shared state – share same piece of state with 2 or more components
  - Caching – API calls or expensive operations, better user experience

Why Redux?

- How Redux Improves Predictability?
  - Data is consolidated to one centralized location: the store
  - Components must request access to data for writing/updating data in the store
  - Strict rules are set on how the store can be updated
  - Redux believes in unidirectional data flow. Data only ever flows one way through the application

- Redux Store vs React Component State
  - React Component State
    - short lived state
    - does not matter to the app globally
    - does not mutate in complex ways
    - localized data, state that does not affect other components
  - Redux Store
    - state that matters globally
    - is mutated in complex ways
    - shared state, accessible through the whole app

Why Redux?

- Pure Functions
  - State in Redux applications is updated using pure functions:
    - Return one and the same result if the same arguments are passed in
    - Depend solely on the arguments passed into them
    - Do not produce side effects

```
// `square()` is a pure function

const square = x => x * x;
```

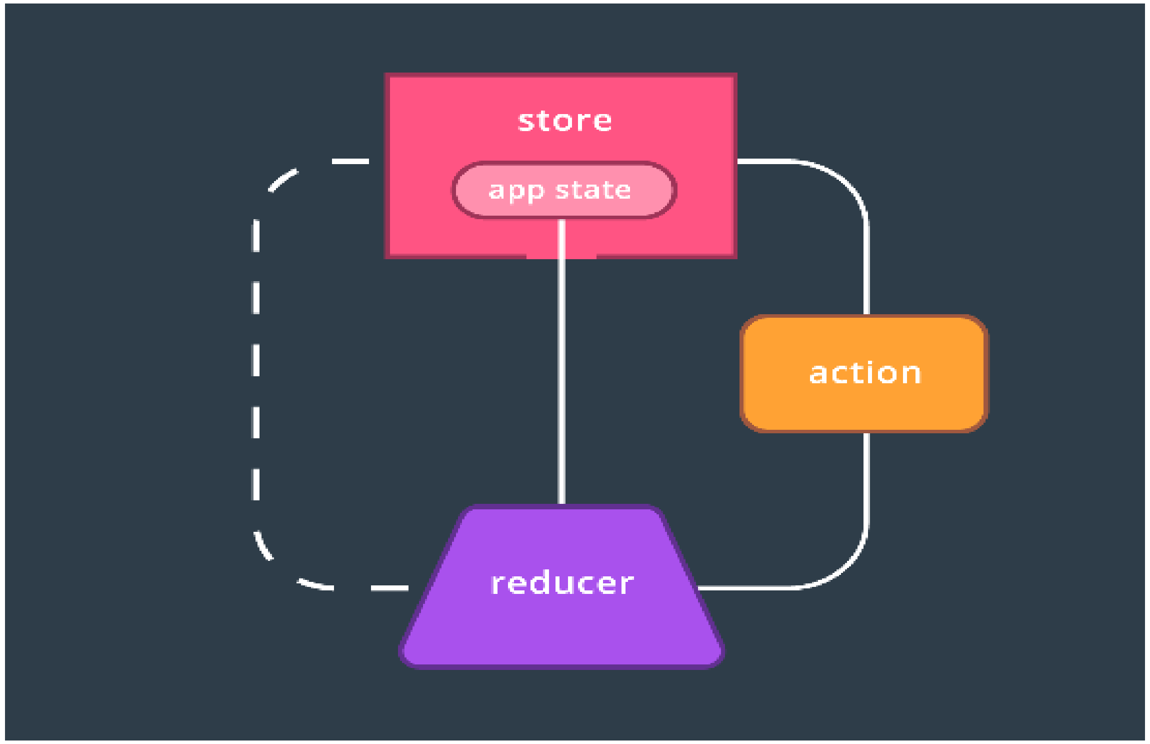
```
// `calculateTax()` is an impure function

const taxPercentage = 0.10;
const calculateTax = amount => amount * taxPercentage;
```

Why Redux?

- Pure Functions
  - What are side effects?
    - Interactions between a function and the world outside of it.
    - Examples:
      - Making HTTP calls
      - Retrieving today's date
      - Math.random()
      - Adding to a database
  - Pure functions
    - are core elements in functional programming
    - are inherently modular, making them easy to test
      - always produce the same result given the same arguments
    - make maintaining code much simpler
      - do not produce side effects
    - lend themselves to better quality code

Core Concepts

- Major parts of Redux
  - The Store
    - is the source of truth for the state in your app
  - Reducers
    - specify the shape of and update the store
  - Actions
    - are payloads of information which tell reducers which type of events have occurred in the application



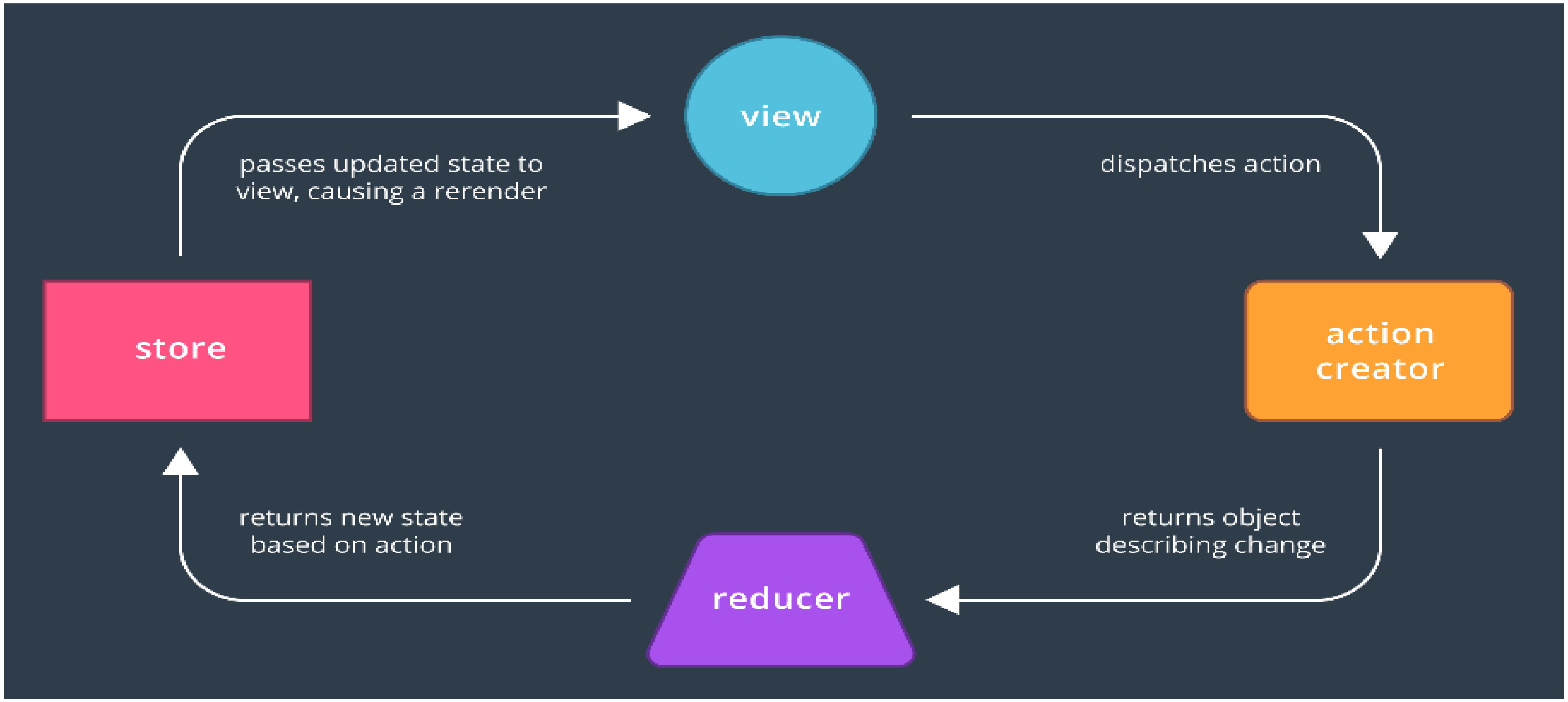
Core Concepts

- Major parts of Redux
  - Most of the application's data or state lives in the store
  - The store's data is populated by reducers. There can be multiple reducers
  - An action is "dispatched" by the store. It is used by reducers to determine what data they should output
  - There can (and will) be more than just one action in a Redux app

- Data flow in a Redux app
  - The store dispatches an action to its reducer
  - The reducer
    - processes the current state with the action
    - returns new state of the app

Core Concepts

- Redux app data flow


view
dispatches action
action creator
returns object describing change
reducer
returns new state based on action
store
passes updated state to view, causing a rerender

Core Concepts

- Actions
  - Similar to browser’s native events
  - Can be thought as custom events within Redux apps
  - Represent various types of events or actions that happen in an app
  - Indicate that some event occurred within the app and state should be updated
  - JavaScript objects that describe any event that should update the app’s state

```
const LOAD_PROFILE = 'LOAD_PROFILE';

const myAction = {
  type: LOAD_PROFILE
};
```

  - ***Type***
    - ‘type’ property distinguishes the specific type of action that occurred within the app
    - prefer constants rather than strings as the value of ‘type’ property
  - ***Payload***
    - can contain any other data needed to represent the type of action that occurred as properties
    - keep the payload as small as possible, include only necessary data

Core Concepts

- Action Creators
  - Are JavaScript functions
  - Wrap actions in functions and make actions more portable and easier to test

```
const ADD_PRODUCT = 'ADD_PRODUCT';

function addProduct(product) {
    return {
      type: ADD_PRODUCT,
      product
    };
}
```

  - whenever we need a ADD_PRODUCT action, we can just call the addProduct() function, pass it a product, and it will generate the action!

Core Concepts

- Reducer
  - is a JavaScript function
  - receives the current state and an action that was dispatched
  - decides how to transform the current state into a brand new state based on the action it received
  - must be a pure function
    - it should take in the current state, an action and return a new state
    - it should not
      - change its arguments
      - have side-effects
      - use other impure functions

```
function appReducer(state, action) {
  switch(action.type) {
    case 'DELETE_FLAVOR':
      let newState = state.filter(iceCream =>
        iceCream.flavor !== action.flavor);
      return newState;
    default:
      return state;
  }
}
```

Core Concepts

- Reducer
  - Specifies the shape of the application's state and decides how the state should change based off specific actions
  - What is returned from the reducer will be the new state of the application
  - You need to make sure you always return either the new state or the previous state
  - The way you decide how to change the state is based on the type of action that was dispatched

  - Store your state in Redux if two components rely on the same piece of state, or if the operation to get that state was expensive.

Core Concepts

- The Store
  - holds the app’s state
  - dispatches actions
  - calls reducers
  - receives/stores new state

- Install Redux package

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

- Redux.createStore(reducer)
  - creates and returns Redux store
  - takes reducer function as first parameter

Core Concepts

- store.getState()
  - will return the current state of the store
  - does not take any arguments

- store.dispatch(action)
  - takes in an action object
  - will call the reducer function, passing it the current state and the action that was dispatched

Reducer Composition

- Process of separating reducers to handle distinct, independent slices of state
- As an application grows, so will the need for multiple reducers to manage different aspects of the Redux store.
- A reducer will receive a section of state and an action, and return a new, modified section of that state

  - Create more than on reducer
  - Use Redux.combineReducers() method

```
{
  users: {},
  posts: {},
  replies: {}
}
```

Normalization

- Do not duplicate data
  - Favor creating references
- Keep your store as shallow as possible
  - Increases performance and minimizes complexity

- Normalizing your Redux store will lead to more efficient and consistent queries

Q & A

- Thank you!