Redux

React Developer Series

Peter Munro

Redux

Redux

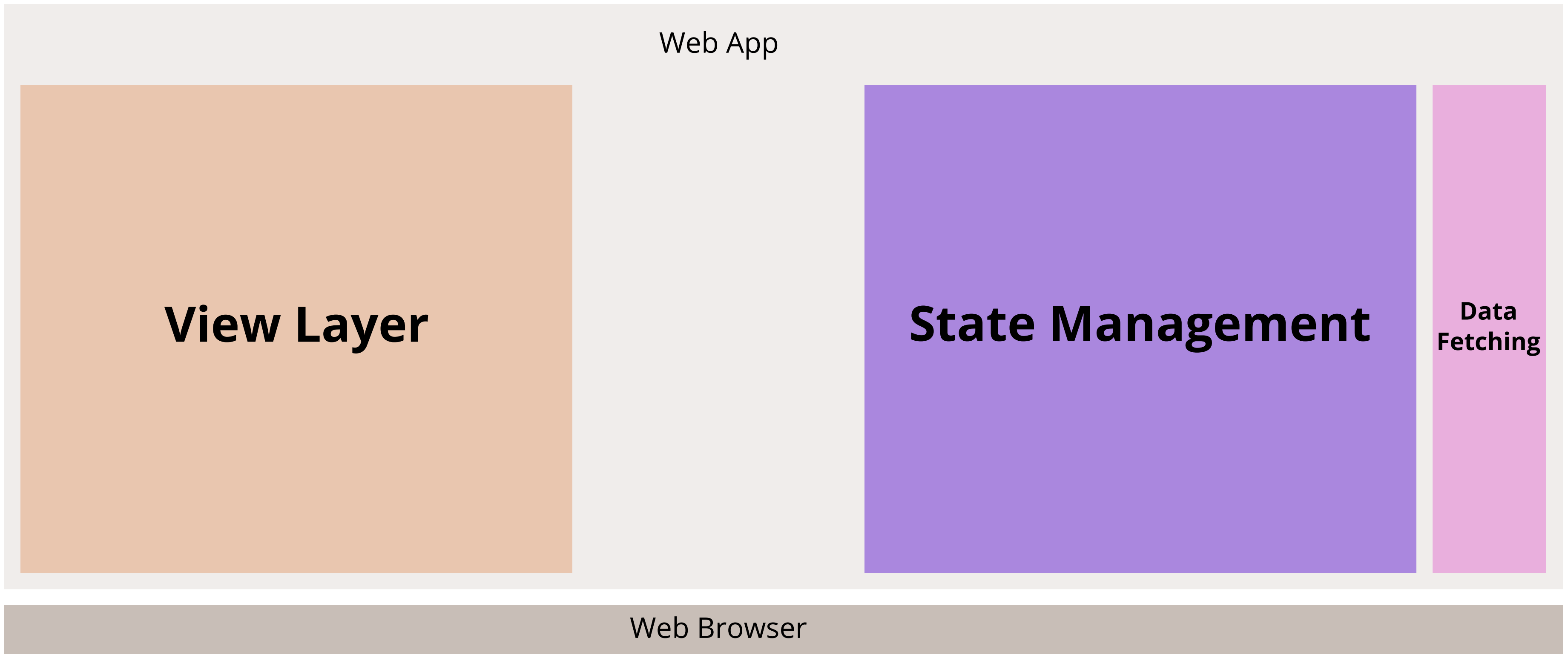
Overview

- Flux & Redux Overview
- Understanding the Redux Architecture
- The State
- Actions
- Action Creators
- Creating Reducer Functions
- Immutability
- Immutable Programming
- Writing Reducers
- The Redux Store
- Integrating with React
- Combining Reducers
- Middleware Overview
- Creating Custom Middleware
- React-Thunk and Asynchronous Actions
- Asynchrony and Server Communications
- Asynchrony Alternative: Redux Saga
- Creating Containers with React-Redux

Flux & Redux Overview

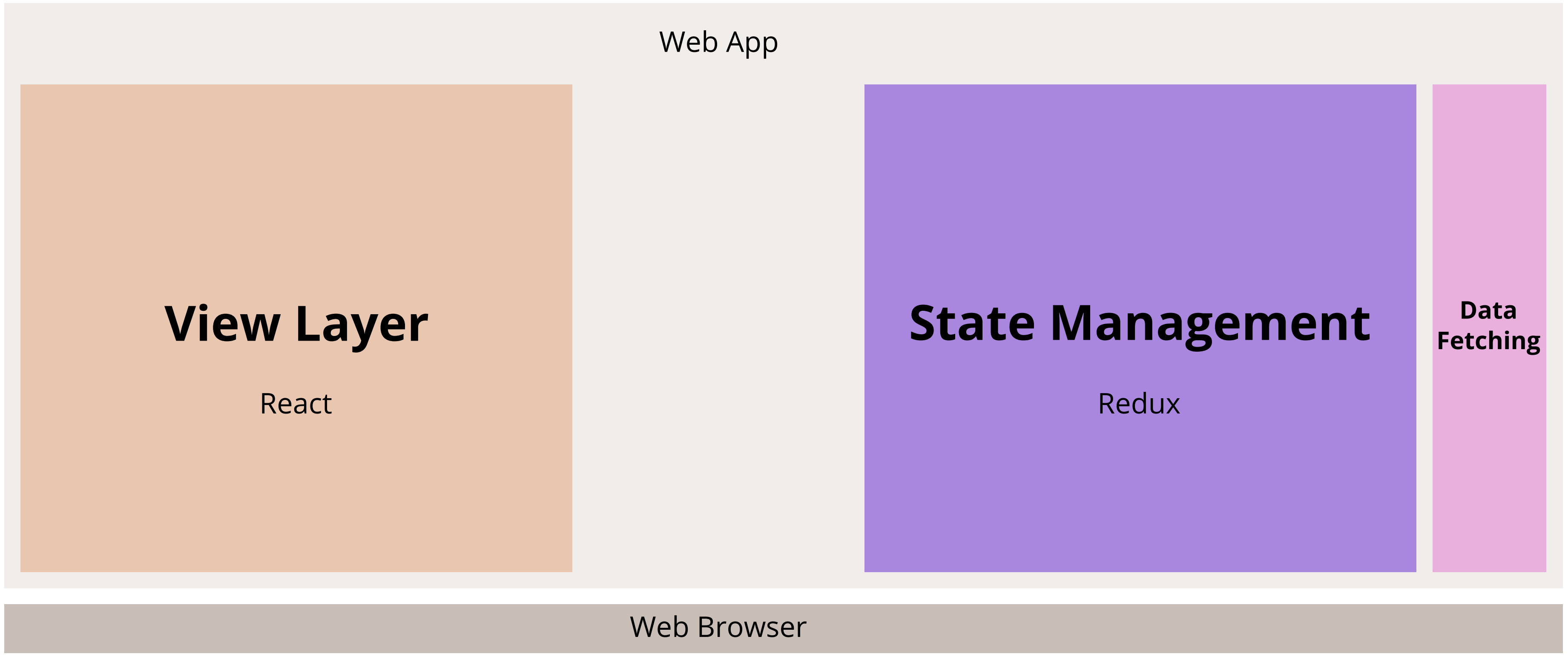
Redux

Single Page Web Apps



- Most web apps have (1) some way to view data in the browser, (2) some state management and (3) a way to fetch data from a server

Where does Redux Fit?



Redux

Redux:

- provides us with *state management* for our app
- can be used with any view layer, including React
- is independent of the view layer
    - it knows **nothing** about React or even about UI, DOM etc

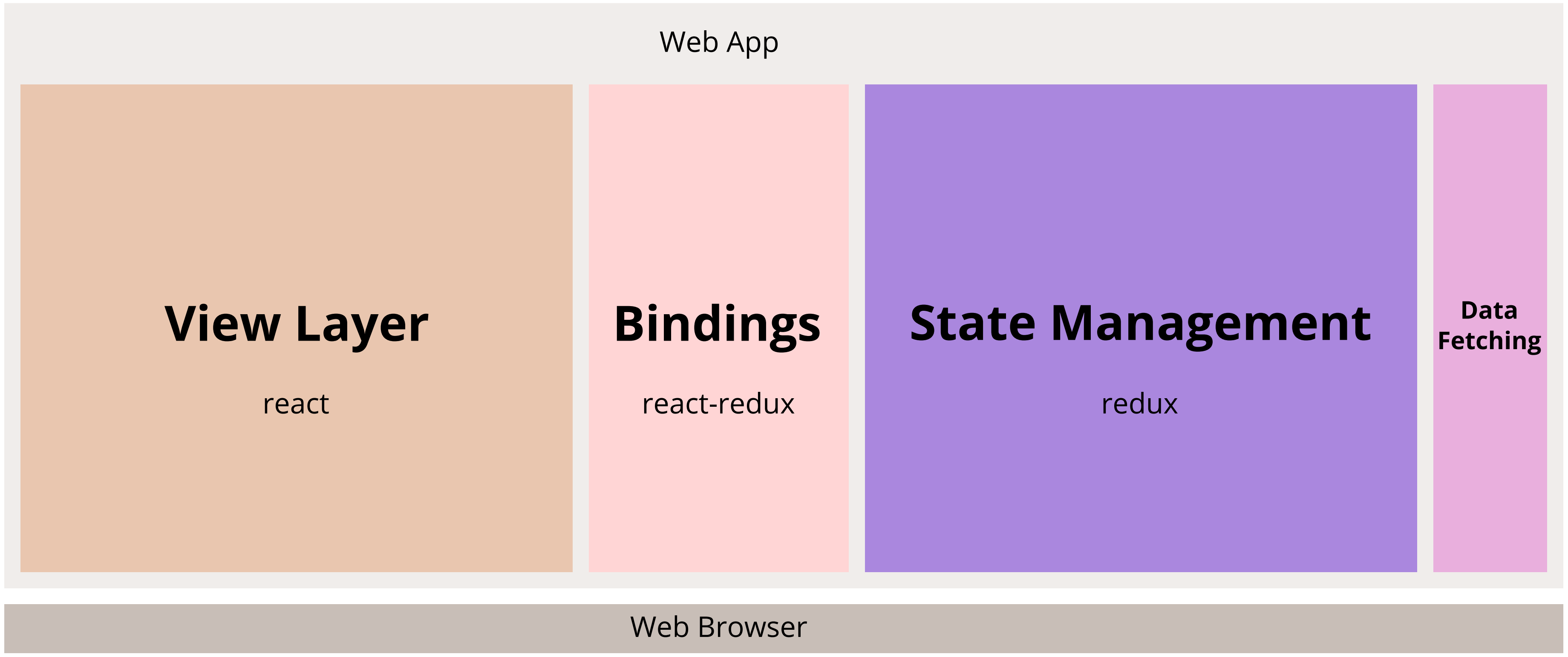
Introducing Redux

- A simple, clear and popular library inspired by Flux
- Developed by Dan Abramov (@gaearon)
    - Written for his talk at ReactEurope 2014
- https://github.com/reactjs/redux

Why Redux?

> *With Redux,*
>
> - *Our app state is predictable, and so is the app.*
> - *Most of the app is pure functions, including the UI. So, it’s easy to test the code.*
> - *Redux is a small library. So, there is no magic.*
> - *This is a pattern used by many developers. So, you are following a good ecosystem. That’s why I think Redux matters.*
>
> *-- Arunoda Susiripala*

Binding Redux to React

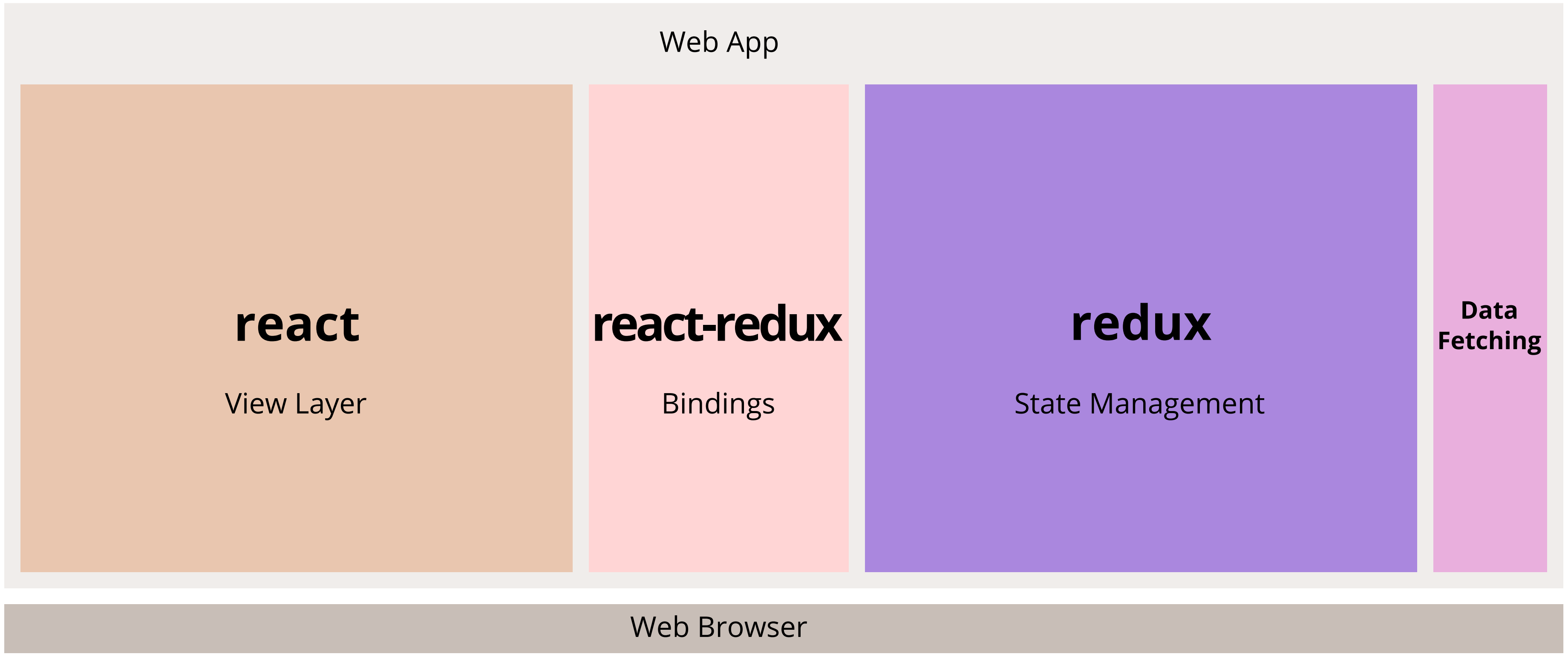


- To use Redux with React, a helper library is available called `react-redux`

Understanding the Redux Architecture

Redux

Redux Architecture



Redux Concepts

- **A Single Immutable State Tree**
    - The whole state of your app is stored in **a single object** inside a store
- **Actions to Change State**
    - Whenever you want to change the state, you need to dispatch an **action**, an object describing what happened
- **Actions as Reducers**
    - To specify how the actions transform the state tree, you write **pure reducers**

"That’s it!" [Dan Abramov]

Redux compared to Flux

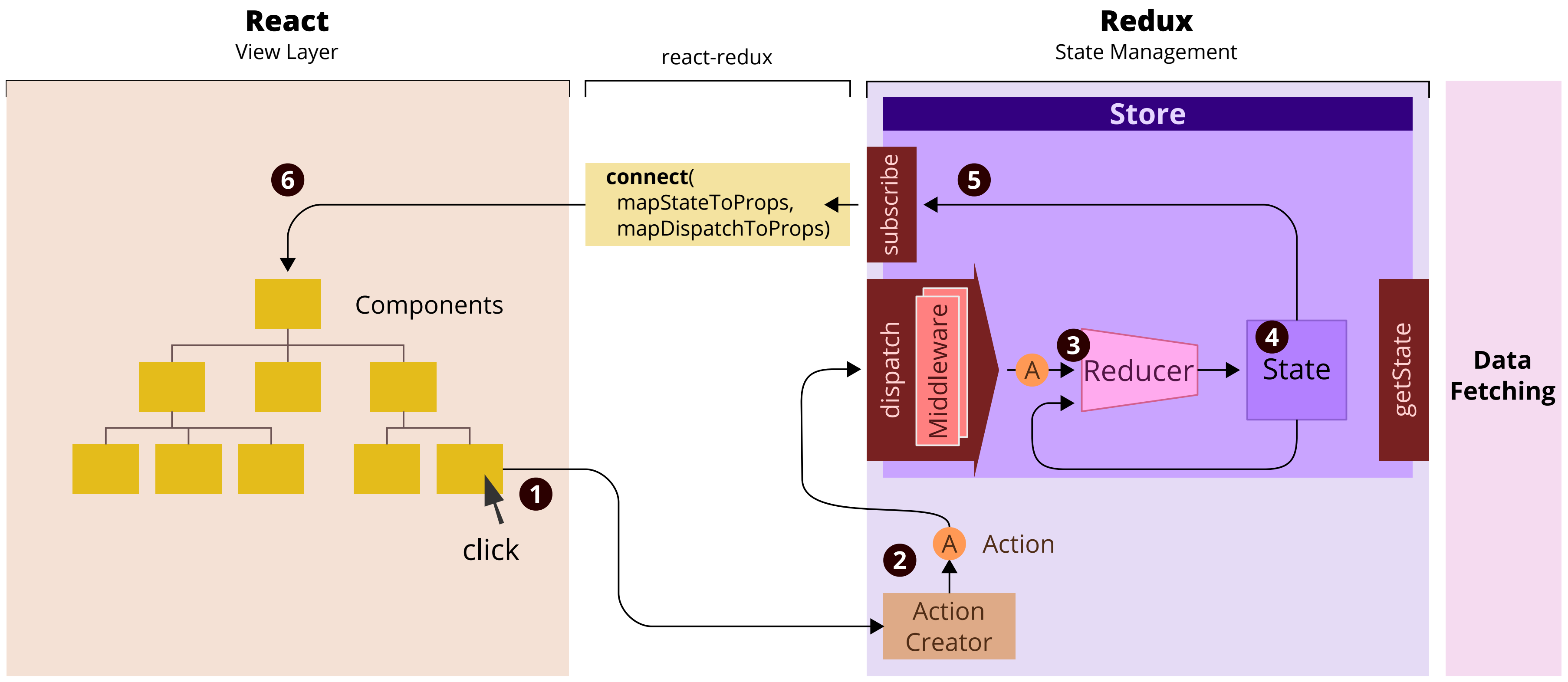
- There's no separate Dispatcher - just a `dispatch()` function on the store
- Action Functions (reducers) simplify actions:
  - They merge Flux's *Dispatcher* with action objects and action creators

A Single Immutable State Tree

With Redux, we keep all our app's state in a single object

- **You** define what it looks like
  - Examples: a single primitive value, an array, an object
- You *never*, *ever* change it!
  - You only ever return a new one

The Flow of Data in a Redux App



The State

Redux

What do we put in state?

Which of these would be likely contenders for storing in Redux's state object?

- The score and blocks in a Tetris app?
- A username entered in a login dialog?
- A password entered in a login dialog?
- The state of a "Mr/Mrs/Miss/Ms/etc" radio button?
- The list of contacts displayed in a contacts app?
- The number of items in a shopping cart?

The State Object

In Redux, you can model state...

- ...as simply as a variable
- ...or as complex as a tree of objects

The top-level will typically be an object, or sometimes an array

A State Object

Here's an example state object from a Tetris® game:

```
const initialState = {
  board: R.repeat(0, BOARD_WIDTH * BOARD_HEIGHT), // [0, 0, 0, ... 0, 0, 0]
  counter: 0,
  gamespeed: 12,
  score: 0,
  block: emptyBlock,
  paused: false,
  gameover: false,
}
```

A More Complex State Object Example

This one is from an IoT dashboard app:

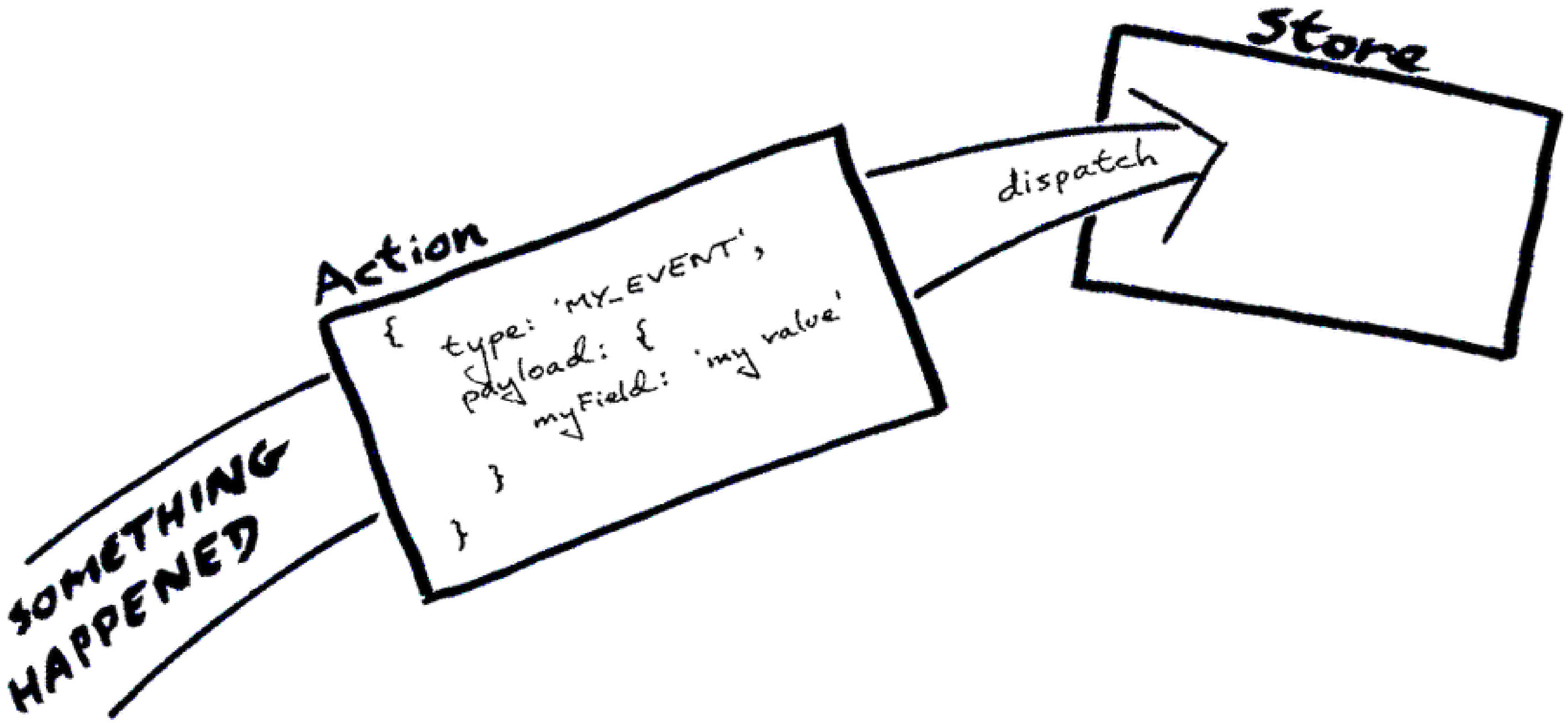
```
export default {
  auth: { },
  app: {
    sites: { payload: [], },
    site: {
      contacts: [], // Contacts for currently selected site
      notifications: [] // Notifications for currently selected site
    }
  },
  routing: {},
  signup: {},
  alert: {
    escalationLadder: { payload: [], },
  },
  ui: {
    sidebarVisible: false,
    notification: null
  }
}
```

Actions

Redux

What's an Action?

Dispatching an Action is the way of saying to Redux *"something happened and I want you to update the state accordingly"*



What should be an Action?

So what might be represented as Actions?

- Mouse events?
- Form submissions?
- Server requests?
- Server responses?

Defining an Action

Think of an action as similar to an 'event':

- it's just a plain JavaScript object
- it's the *only* source of information to update the store
- it has fields:
  - A `type`
  - A payload

Redux doesn't care about the action's data - just that it has a type, conventionally a string, and is serializable

Example

Here's an action which updates a reminder with a new time:

```
{
  type: 'UPDATE_REMINDER',
  reminderTime: '1457567746099'
}
```

- `type`: a string (by convention) of your choice
- rest of object: the details of the action

Example 2

These actions start and stop a Tetris game:

```
{
  type: START_GAME,
  currentRandomShape: initialShape,
  nextRandomShape: nextShape,
}
```

```
{
    type: GAME_OVER,
}
```

And this one adds a score:

```
{
    type: ADD_SCORE,
    points: Math.pow(clearedLines, 2) * 100,
    clearedLines,
}
```

Aside: Using Flux Standard Actions

We can rewrite the action using a coding convention called Flux Standard Actions

```
{
  type: 'UPDATE_REMINDER',
  payload: {
    reminderTime: '1457567746099'
  }
}
```

- `payload`: typically an object containing the details of the action
- The redux-actions utility package helps you work with Flux Standard Actions

Action Types: Strings or String Constants?

Notice that we specified `type` as a string:

```
{ type: 'UPDATE_REMINDER', ... }
```

What could be the problems with this from a code maintenance perspective?

Using String Constants

Instead, we can use constants:

```
{ type: UPDATE_REMINDER, ... }
```

where `UPDATE_REMINDER` is defined as:

```
const UPDATE_REMINDER = 'UPDATE_REMINDER';
```

...or inside an object as:

```
const ActionTypes = {
  UPDATE_REMINDER: 'UPDATE_REMINDER',
  ...
};
```

Dispatching Actions

To 'dispatch' an action means to send it to the Redux store:

```
dispatch({
  type: UPDATE_REMINDER,
  reminderTime: '1457567746099'
});
```

- The `dispatch()` function is provided by the Redux Store

Who dispatches Actions?

Typically, actions will be dispatched by:

1. UI components
   - for example on button clicks or form submissions
2. Server responses
   - when a response (or timeout) has been received from a server

The nice thing is...

...that the action sender *doesn't care* how the state is updated, just that they have to send an action

Action Creators

Redux

Actions

- Actions are simple to create
    - They're just object literals!
- As your project grows, you may need to use *Action Creators*

Action Creators

An Action Creator is just a function that helps callers create actions easily:

From this:

```
dispatch({
  type: UPDATE_REMINDER,
  reminderTime: '1457567746099'
});
```

To this:

```
dispatch(updateReminder('1457567746099'));
```

- We've wrapped the action in an "Action Creator" function, `updateReminder()`
- We can pass parameters to this function to tailor the action we want created

Writing Action Creators

Writing action creators is fairly simple:

```
export function changePassword(newPassword) {
  return {
    type: ActionTypes.CHANGE_PASSWORD,
    newPassword,
  };
}
```

- The action creator takes a parameter that it uses to populate the action

Why Action Creators again?

First, for simple actions you may not need them.

However, they're useful for a number of reasons:

1. code maintenance
   - it's easier to change the contents of an action (eg a property name) if it's only created in one place
2. a place to put conditional logic
   - we'll see this later, particularly with async code
3. Oh, and they don't have to be pure functions
   - they can initiate server API calls etc

http://blog.isquaredsoftware.com/2016/10/idiomatic-redux-why-use-action-creators/

Creating Reducer Functions

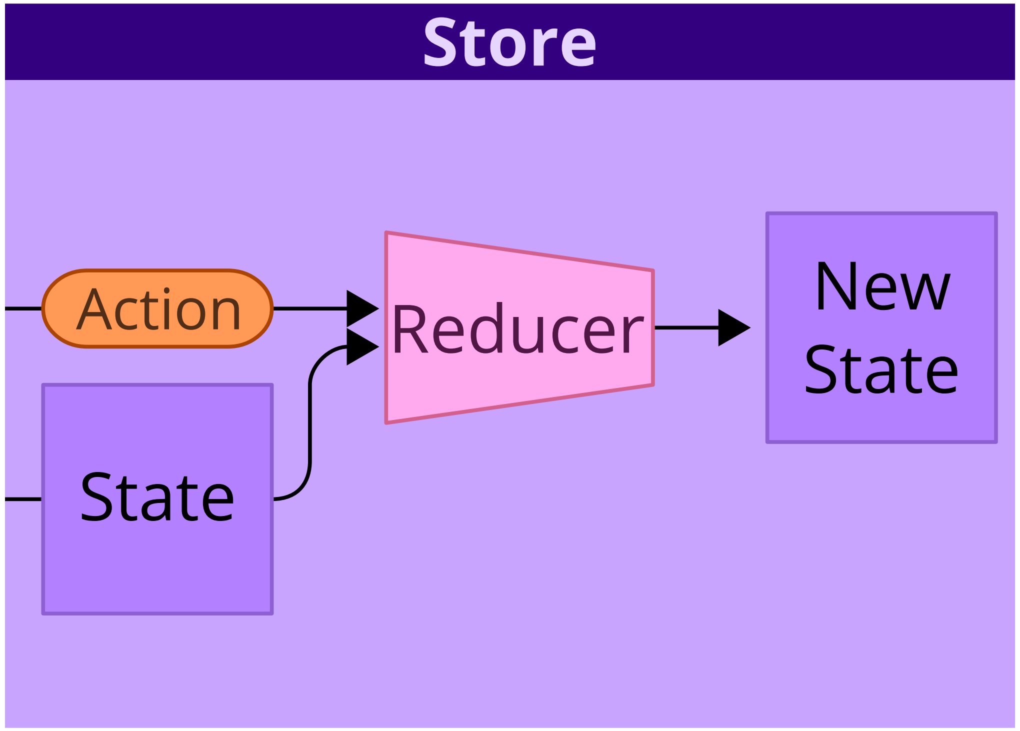
Redux

Reducers

A reducer is just a function

- A reducer can simplify or reduce a data item
    - It's typically an aggregate or summary step
- It takes a collection input and produces a single output
    - Example: `Array.prototype.reduce()`

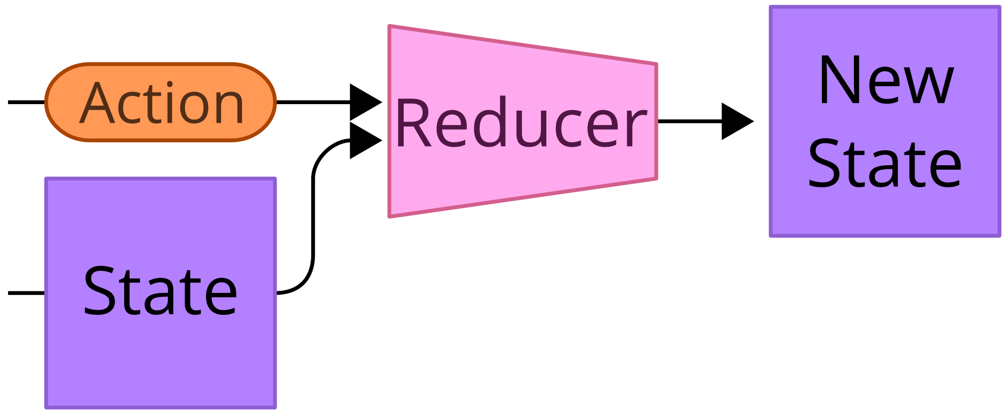
Redux Reducers



In Redux, a reducer is a pure function

- It maps a state + action into a new state:
  - Inputs: (`existingState`, `action`)
  - Output: `newState`

Reducers are Pure Functions



Being a pure function, your reducer *must not*:

- Change its arguments
- Perform side effects
- Call non-pure functions

So, no making calls to APIs in a reducer!

Actions vs Reducers

Actions

- Actions say "something happened"
- Example:

```
{
    type: 'DELETE_ACCOUNT_REQUESTED',
    payload: { id: 'F7092459-A01A-4672-8798-61AE515CDA34' }
}
```

Reducers

- Reducers say "OK, so this is what the new state should look like based on that something that happened"
- Reducers don't update state; they create a *new* state and return it

Immutability

Redux

Immutability

- To 'mutate' an object just means changing its value
- An 'immutable' value is one that can't be changed

You're already using Immutability

Strings are immutable:

```
'hello'.toUpperCase()      // "HELLO"
```

- The string's value isn't changed in-place

Immutable Data in JavaScript

Immutable Types

- `Number`
- `String`
- `Boolean`
- `Undefined`
- `Null`

- If you change an immutable value, you get a new value

Mutable Types

- `Object`
- `Array`
- `Function`

- If you change a mutable value, you change the value in-place
  - and any other users see the changed data

Why Immutability?

Immutability is important for these reasons:

1. It makes code *predictable*
    - and therefore easier to reason about
2. Compilers can reason too
    - They can optimize code that deals with immutable data

Predictability - 1

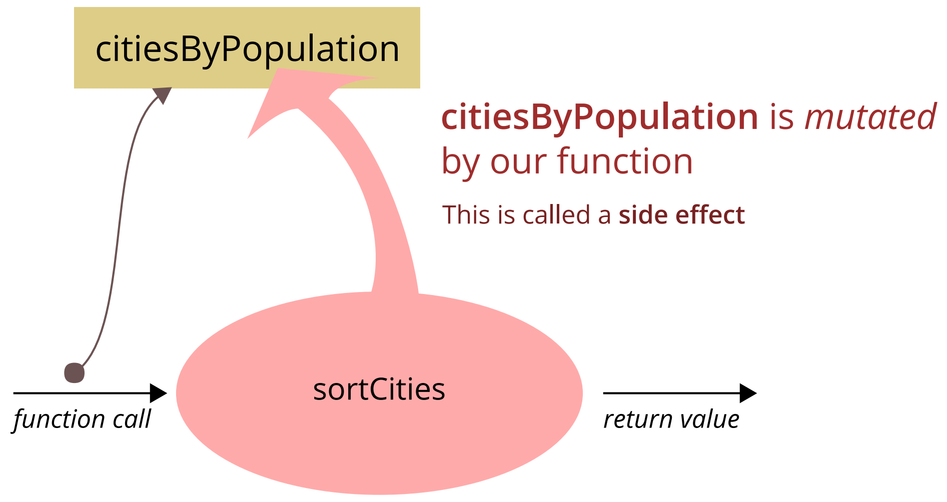
Consider this code:

```
const citiesByPopulation = ['Shanghai', 'Karachi', 'Beijing',
                            'Dhaka', 'Delhi', 'Lagos', 'Istanbul'];

function sortCities(cities) {
  console.log('Cities in alphabetical order:', cities.sort());
}

sortCities(citiesByPopulation);

console.log('Cities in order of population:', citiesByPopulation);
```

What's displayed?

Predictability - 2

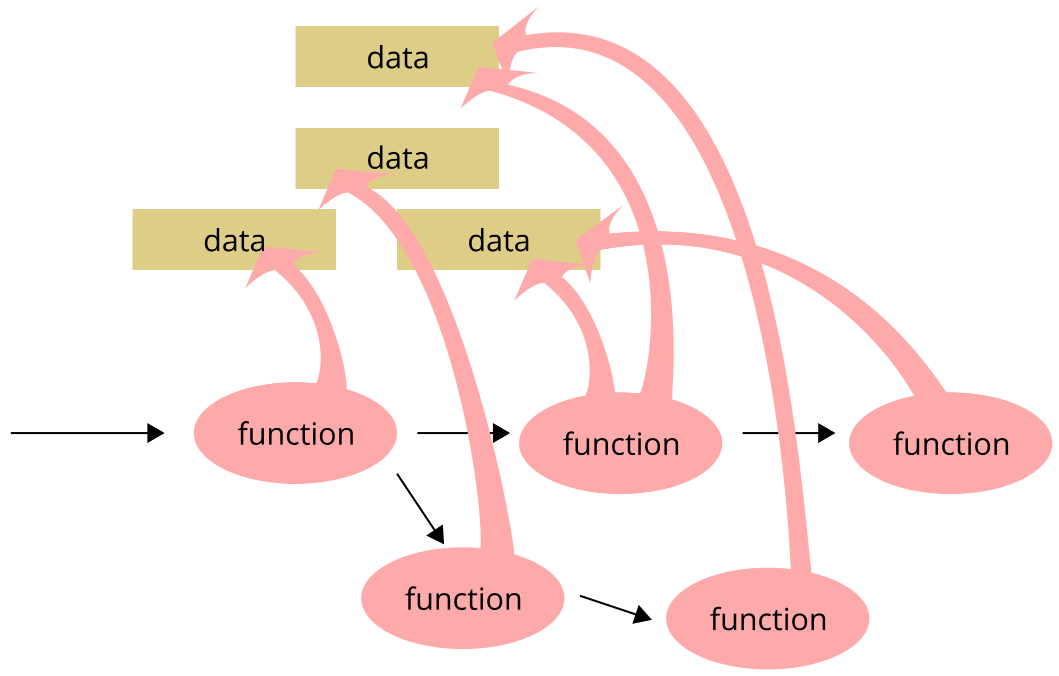
So we have this:



- This function doesn't just generate an output based on its input
  - It introduces a *side effect*: data outside it is changed

Predictability - 3

Suppose our whole application worked like this. If data is changed, which function changed it?

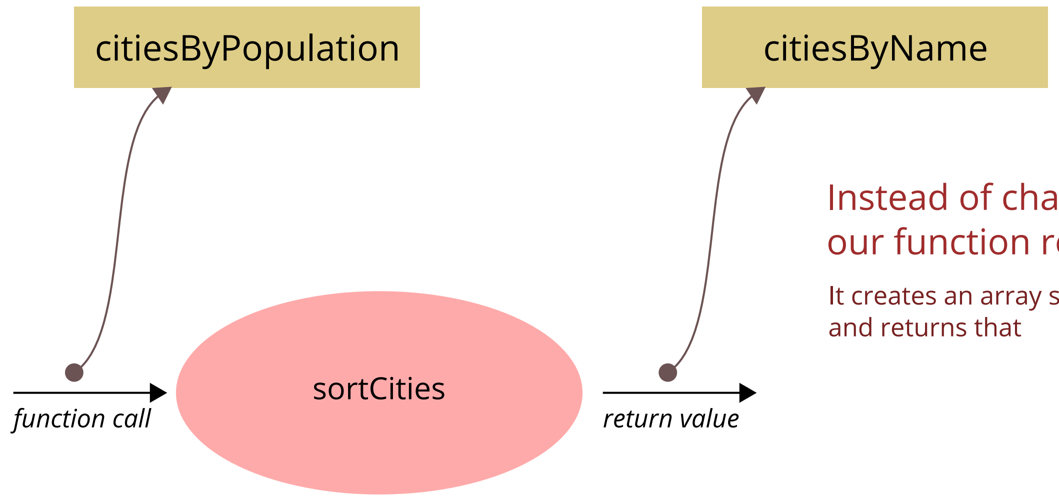


> *A hammer should be able to nail nails regardless of the weather or the kettle boiling earlier. -- ford_beeblebrox on [reddit]*

Solving the Predictability Problem

Let's rewrite our `sortCities` to return a *new array*:

```
function sortCities(cities) {
    return [...cities].sort();
}
```



Instead of changing data in-place, our function returns new data

It creates an array sorted by name and returns that

Pure Function

This function is called a **pure function**

- It doesn't have any side effects
- It doesn't change its inputs
- It has no external dependencies (other than perhaps other pure functions)
- Whenever we call it with the same inputs, it will always produce the same outputs

Benefits

1. Our code is **predictable**
    - We can predict that no adverse changes will occur:
        - we know our function doesn't changing anything outside itself
        - we know our input data has not changed
2. Our code is **easier to write, understand and maintain**
    - Functions are isolated from each other
    - We can test them in isolation

Question

Does this mean we should use immutability everywhere? What about dealing with:

- network connections?
- databases?
- users and their input?

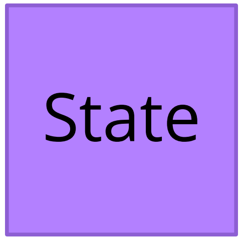
Using Immutability

Rich Hickey suggests:

> *42 doesn’t change. June 29th 2008 doesn’t change. Points don’t move, dates don’t change, no matter what some bad class libraries may cause you to believe. Even aggregates are values. The set of my favorite foods doesn’t change, i.e. if I prefer different foods in the future, that will be a different set.*

Immutability in Redux

- You can think of Redux as *state management* for your app
  - It keeps all state in a single object
  - This object is immutable - we only ever generate a new one



Immutable Programming

Redux

Writing Pure Functions to Update State

With Redux, we write 'reducers' that must be pure functions

- Instead of modifying state:
    - we create a *new* state object, containing the updates
    - we return that

Appending to an Array

It's tempting to do this:

```
function addElementToArray(array, element) {
  array.push(element);
}
```

However, this *modifies* the original array

'Appending to' an Array

- We don't want to modify the original array
    - We return a new one instead:

ES5

```
function addElementToArray(array, element) {
  return array.concat(element);
}
```

Or in ES6:

ES6

```
function addElementToArray(array, element) {
  return [...array, element];
}
```

Other Array Operations

As well as appending to an array, we may need to:

- remove an element
- modify an element

'Updating' an Object

- The other main data structure is a JavaScript object
- To generate a new object, we can use `Object.assign()` (ES6)

```
let original = { first: 1, second: 1, third: 1 };
let updates = { second: 2, third: 2 };

let newObj = Object.assign({}, original, updates);
console.log(newObj);      // {first: 1, second: 2, third: 2}
```

- We have now 'updated' our state to reflect the changed properties `second` and `third`

Other Object Operations

We may also need to perform these operations on our state:

- Adding a property
- Removing a property
- Updating a nested property

Writing Reducers

Redux

Writing Reducers

Here's the general form of a reducer:

```
const initialState = {};
const myReducer = (state = initialState, action) => {
  switch (action.type) {
    case ACTION_A:
      return state_X;
    case ACTION_B:
      return state_Y;
    // ...
    default:
      return state;
  }
};
```

Another Example

This reducer updates the score of a Tetris game:

```
function gameScore(state = {}, action) {
  switch (action.type) {
    case actions.START_GAME:
      return {
        points: 0,
        clearedLines: 0
      };
    case actions.ADD_SCORE:
      return Object.assign({}, state, {
        points: action.points + state.points,
        clearedLines: action.clearedLines + state.clearedLines
      });
    default:
      return state;
  }
}
```

Conventions when Writing Reducers

1. If an unknown action is passed in, the reducer should return the current state
2. The first time a reducer is called, it is called with `undefined` as the state. It must return the app's initial state in this case.

Remember - Pure Functions!

- Reducers must be *pure functions*
  - No mutating of state
  - Always return a *new* state object

The Redux Store

Redux

The Store

The store:

1. holds the current application **state object**
2. lets you dispatch **actions**
3. is where you provide the **reducer** (when you create it)

Notice that the store encompasses the three principles of Redux listed earlier

Creating the Store

To create the store:

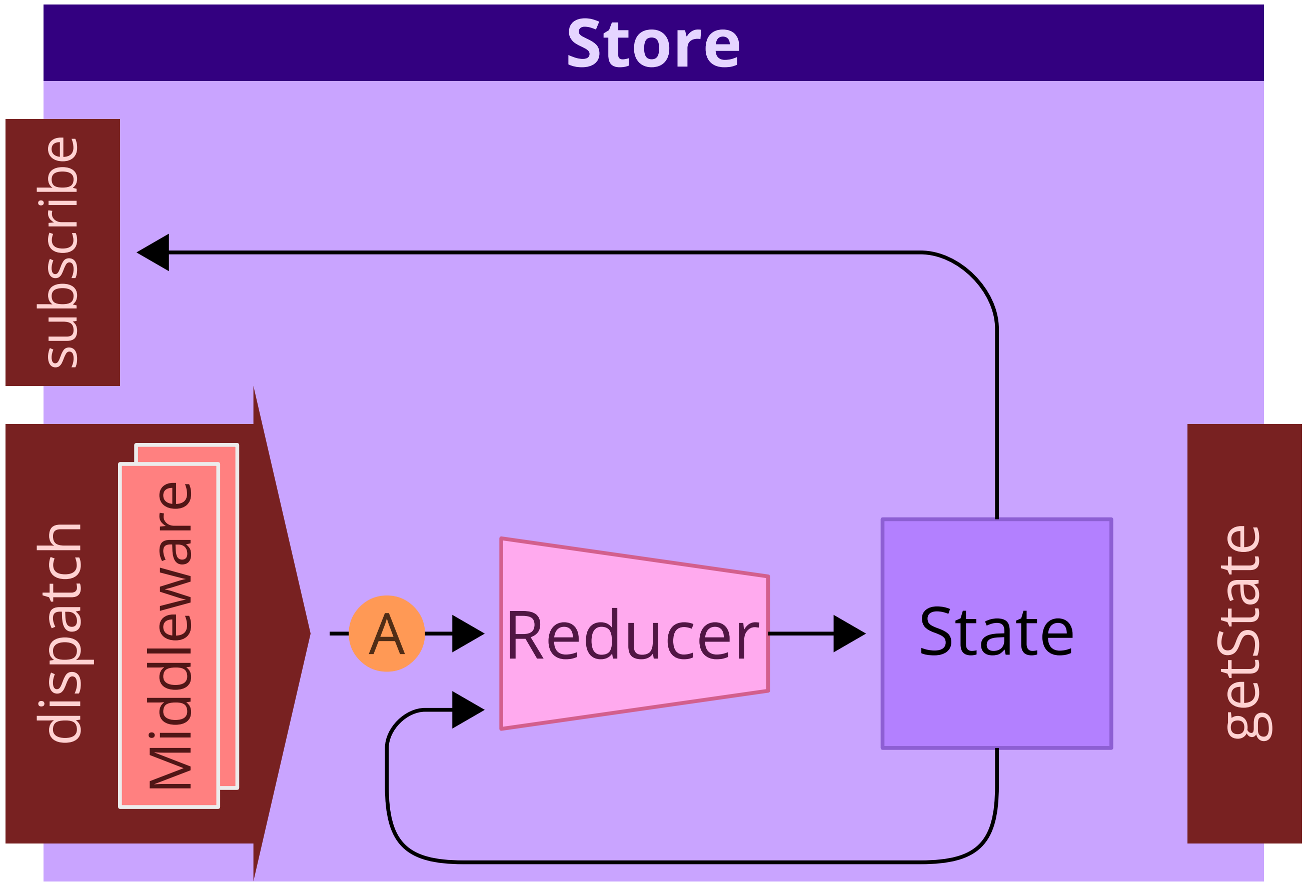
```
import { createStore } from 'redux';

...

const store = createStore(myReducer);
```

Store Methods

The store has these methods:

1. mystore`.dispatch()`
    - Lets you dispatch actions to change your application's state
2. mystore`.getState()`
    - Retrieves current state of Redux store
3. mystore`.subscribe()`
    - Lets you register a callback that the Redux store will call any time an action is dispatched, so you can update the UI to reflect current application state

Redux Store


Store
subscribe
dispatch
Middleware
A
Reducer
State
getState

Integrating with React

Redux

React's Needs

React needs a way for its components:

- to be updated when the state changes
- to retrieve the new state from the store
    - ideally passed in via `props`
- to call the store's `dispatch()` method to dispatch actions

Recap: Container and Presentational Components

Dan Abramov discusses this design pattern. Briefly:

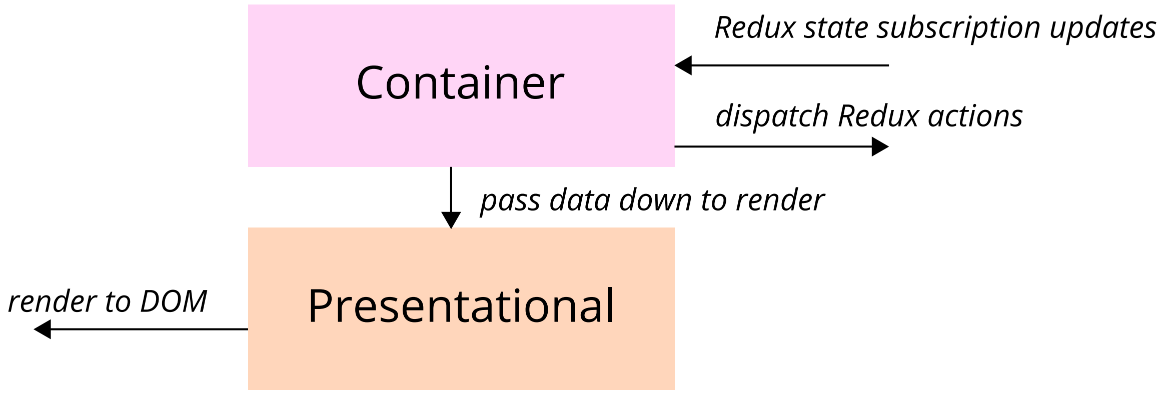
PRESENTATIONAL	CONTAINER
Concerned with how things look	Concerned with how things work
Has DOM markup and styles	Minimal DOM markup; no styling
Uses data passed in via `props` to render	Retrieves or generates data to pass to presentational component
No data retrieval	May talk to server to retrieve data
Minimal or no state - just UI state	Often stateful

PRESENTATIONAL CONTAINER

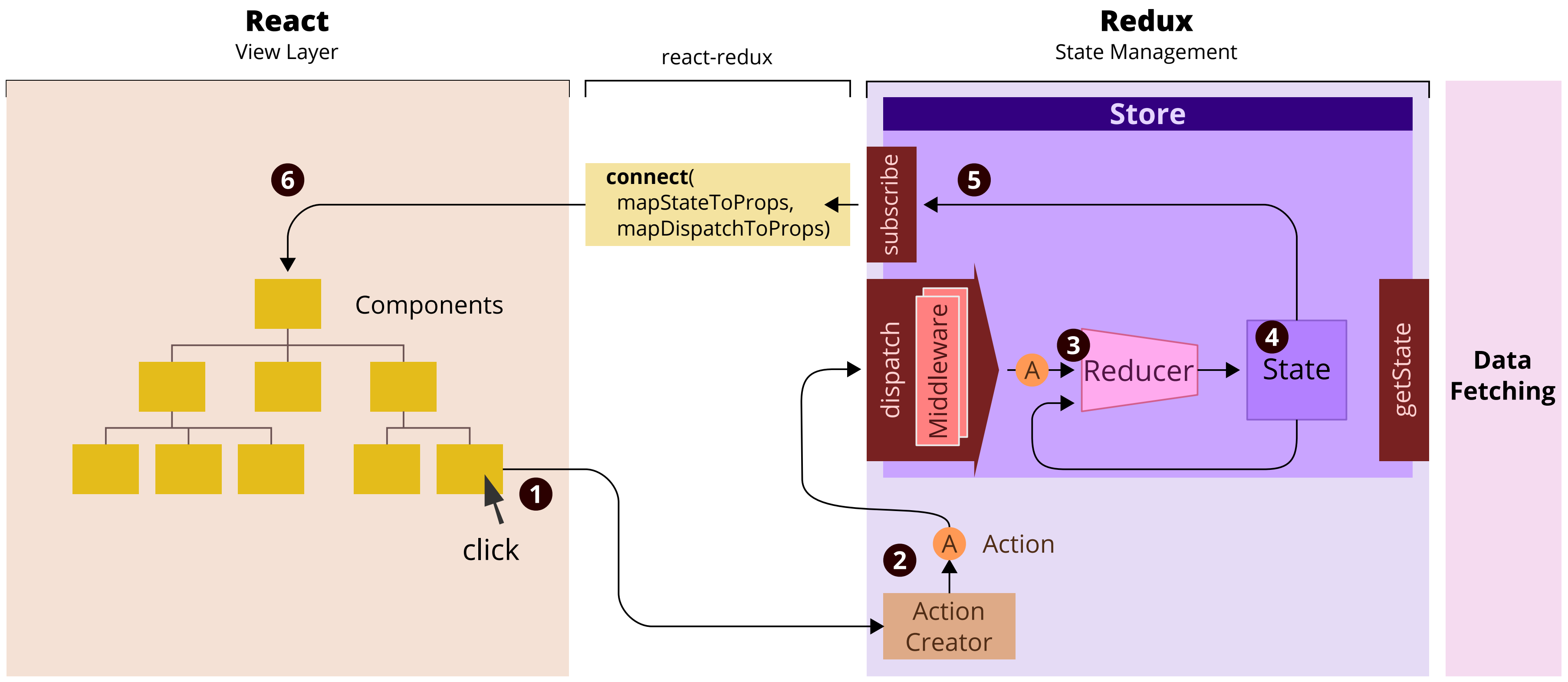
Container Components with Redux

When using Redux, a container component will typically:

- be aware of Redux
  - the presentational component isn't!
- subscribe to the Redux store
- dispatch Redux actions



Data Flow using react-redux



Installing React Redux

To install:

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

The react-redux API

The react-redux API meets these needs by:

- providing a `connect()` function which *wraps* a component
- providing `mapStateToProps()`, called whenever the store is updated
- providing `mapDispatchToProps()` to let the component call the store's `dispatch()`

Implementing Container Components

- You could implement by hand
  - subscribe to the store (`store.subscribe()`)
  - on state updates, render children with new data from store
- But it's easier to generate them using `connect()`!

Using `connect()`

Three steps:

1. First, define `mapStateToProps()`
2. Second, define `mapDispatchToProps()`
3. Call `connect()` to generate the new container component

```
export default connect(mapStateToProps, mapDispatchToProps)(ShoppingListComponent)
```

Defining `mapStateToProps()`

Purpose:

`mapStateToProps()` provides the `props` for the presentational component, based on the current state

```
import { connect } from 'react-redux';
import ShoppingListComponent from './ShoppingListComponent';

function mapStateToProps(state) {
   return { shoppingList: state.shoppingList };
}

export default connect(mapStateToProps)(ShoppingListComponent);
```

- `ShoppingListComponent` will be passed the prop `shoppingList`
  - The value of this prop is taken from the Redux state's property of the same name

Defining `mapDispatchToProps()`

`mapDispatchToProps()` provides callback `props` for the presentational component

- these callbacks will dispatch Redux actions
- the presentational component need not understand Redux, so just invokes the callback

```
// action creators
export function notify(notification) {
  return { type: ActionTypes.NOTIFY, notification };
}
export function getSiteContacts(site) {
  // ...
}
```

```
// mapDispatchToProps
const mapDispatchToProps = { getSiteContacts, notify };
```

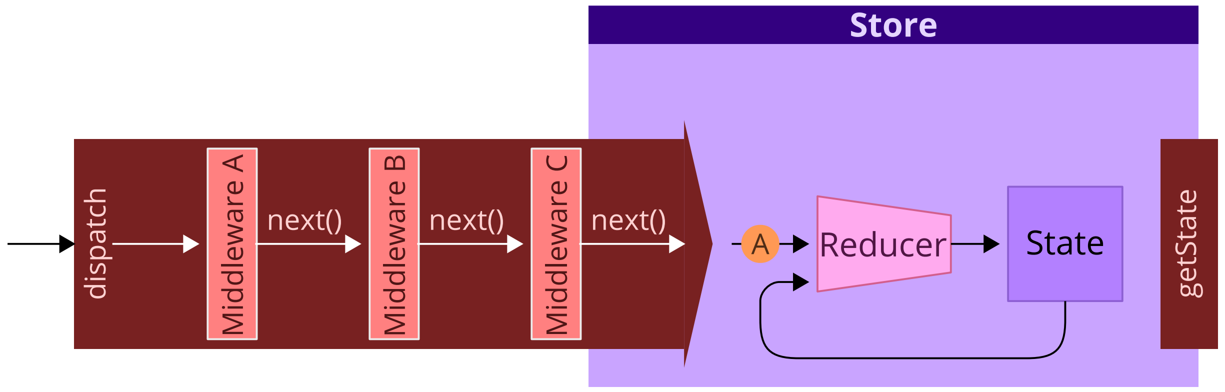
Middleware Overview

Redux

What's Middleware?

- Without middleware:
  - Redux just computes new state from previous state and an action via reducers
  - Reducers are pure functions: no side effects
  - So Redux can't really do anything else!
- Middleware adds "plug-ins" to Redux, helping us to:
  - Perform asynchronous operations which take time
  - Perform impure operations, such as database calls

Middleware is Composable



- Middleware enables you to intercept the `dispatch()` call with your own middleware function, before the action gets sent to the reducers
- Multiple middleware functions may be used ('composed') in turn

Creating Custom Middleware

Redux

The Middleware Function Signature

A middleware function looks like this:

```
const myCustomMiddleware = store => next => action => {
  // ...
}
```

- So `myCustomMiddleware` is a function that receives a `store` and returns a function that takes a `next` parameter
- This function returns a function that takes an `action` and performs our middleware operations

A Simple Middleware

Here's a simple middleware example:

```
const logger = store => next => action => {
  console.log('[logger] Dispatching action', action);
  let result = next(action);
  console.log('[logger] Action returned', result);
  return result;
}
```

- We've intercepted the `dispatch()` function, inserting our own logging code
- The `next()` function invokes the next middleware in the chain

Using Middleware

To introduce middleware into Redux, call `applyMiddleware()`:

```
let middlewares = applyMiddleware(middlewareA, middlewareB, middlewareC);
```

- This returns a store enhancer that you can pass in to `createStore()`

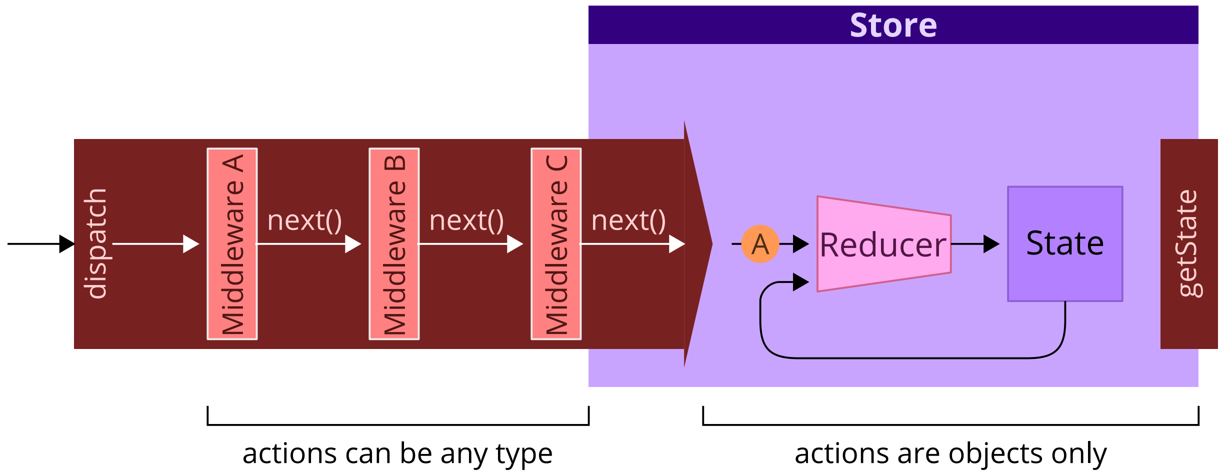
For example:

```
let myApp = combineReducers(reducers)
let store = createStore(
  myApp,
  applyMiddleware(
    thunk,
    logger,
    crashReporter
  )
);
```

Exercise

Exercise: Writing Custom Middleware

Other Kinds of Actions



- Until now, actions have been objects only
  - The root reducer only accepts objects (with a `type` field)
- However, middleware can accept any type of action
  - A middleware could accept functions, promises, arrays or other types

Exercise

Exercise: Middleware for Dispatching Functions

React-Thunk and Asynchronous Actions

Redux

Why Thunk?

thunk *(noun)* : A delayed action

In pure Redux, an Action Creator must return an action

- However:
  - For asynchronous calls such as waiting for a server response...
  - ...it'd be great to deal with asynchronous functions as actions

Redux Thunk gives us exactly that

- These "functions as actions" can be dispatched, and can in turn dispatch further actions

Redux Thunk

Redux Thunk:

- is middleware
- enables you to write action creators that return functions
- once dispatched, these functions are called by the Thunk middleware

So Redux Thunk helps Redux deal with asynchrony

- You don't *need* it
    - ...but you have to deal with the asynchrony yourself otherwise

Installing Redux Thunk

To install:

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

Asynchronous Action Creators

Normally, we dispatch Actions:

```
dispatch({ type: LOAD_CONTACT, contactId: 47 });  // dispatch an action directly
dispatch(loadContact(47));  // dispatch via an action creator
```

Asynchronous Action Creators - 2

Redux Thunk enables us to dispatch functions too:

```
function myAsyncAC(dispatch) {
  dispatch({ type: PENDING });
  setTimeout(() => {
    dispatch({ type: SUCCESS });
  }, 5000);
}

dispatch(myAsyncAC)
```

Redux Thunk Flow

- `redux-thunk` doesn't give us server communications
  - It just gives us the ability to dispatch functions as actions
- Take a look at the source code, which may look familiar

Asynchrony and Server Communications

Redux

Server-related Actions

For each request, we typically have three actions

- a `'LOAD_CONTACTS'` or `'LOAD_CONTACTS_PENDING'` action
- a `'LOAD_CONTACTS_SUCCESS'` action
- a `'LOAD_CONTACTS_FAILURE'` action

Choosing an HTTP API

Options:

- Classic `XMLHttpRequest`
- Fetch API
  - Typically lacks support for cancelable requests
  - You need to call `.json()` to convert objects
  - Server error responses are passed to the `.then()` instead of to `.catch()`
- Axios
  - Fixes these issues
  - However, uses XMLHttpRequest underneath so currently can't be used in ServiceWorkers

Contacting a Server

Here's an example of dispatching an AJAX request:

```
function myThunkActionCreator(someValue) {
  return (dispatch, getState) => {
    dispatch({type : "REQUEST_STARTED"});

    myAjaxLib.post("/someEndpoint", {data : someValue})
      .then(response => dispatch({type : "REQUEST_SUCCEEDED", payload : response})
      .catch(error => dispatch({type : "REQUEST_FAILED", error : error});
  };
}
```

- We dispatch a `REQUEST_STARTED` action to update the state that the request is in progress
- If we later receive a reply, we dispatch `REQUEST_SUCCEEDED` with the payload containing the response
- On an error (or timeout), we dispatch `REQUEST_FAILED` indicating the error, which can be used to update the UI

A fetch API Example

Here's an example using the *fetch* API:

```
function getData(contactId) {
  return dispatch => {
    dispatch(loadContactPending());
    fetch(`/contact/${contactId}`)
    .then(checkHttpStatus)
    .then(response => response.json())
    .then(json => dispatch(success(json)))
    .catch(ex => processErrors(ex, dispatch, failure));
  }
}
```

Asynchrony Alternative: Redux Saga

Redux

Saga

saga (*noun*)

1. a long story of heroic achievement, especially a medieval prose narrative in Old Norse or Old Icelandic
2. a long, involved story, account, or series of incidents

Saga - Computer Science

Saga

- A *long-lived transaction* ... takes a substantial amount of time, possibly on the order of hours or days
- A *saga* is a long-lived transaction that can be broken into a sequence of relatively independent steps that can be interleaved with those of other transactions

See Sagas by Hector Garcia-Molina and Kenneth Salem

Redux Saga

Think of a saga as:

- "a process that receives events, and may emit new events (sync or async), aiming to orchestrate complex workflows inside your application" (@slorber)
- "a piece of code which runs in the background, watches for dispatched actions, may perform some async calls (or synchronous impure calls like browser storage) and can dispatch other actions to the store" (Yassine Elouafi)

Redux Saga - 2

With redux-saga:

- Sagas are like background threads
- You can start, pause and cancel these 'threads' from the main app by dispatching actions as normal
- You use ES6 generators to write your async flows

ES6 Generators

Generators:

- are special functions, defined as `function* myGenerator() { … }`
- are like 'processes': they can be called, paused and resumed at different stages of their execution

Generators and Redux Saga

See 'Generators' course.

Redux Saga Resources

- Redux Saga codebase
  - redux-saga
- Redux Saga book
  - redux-saga.js.org