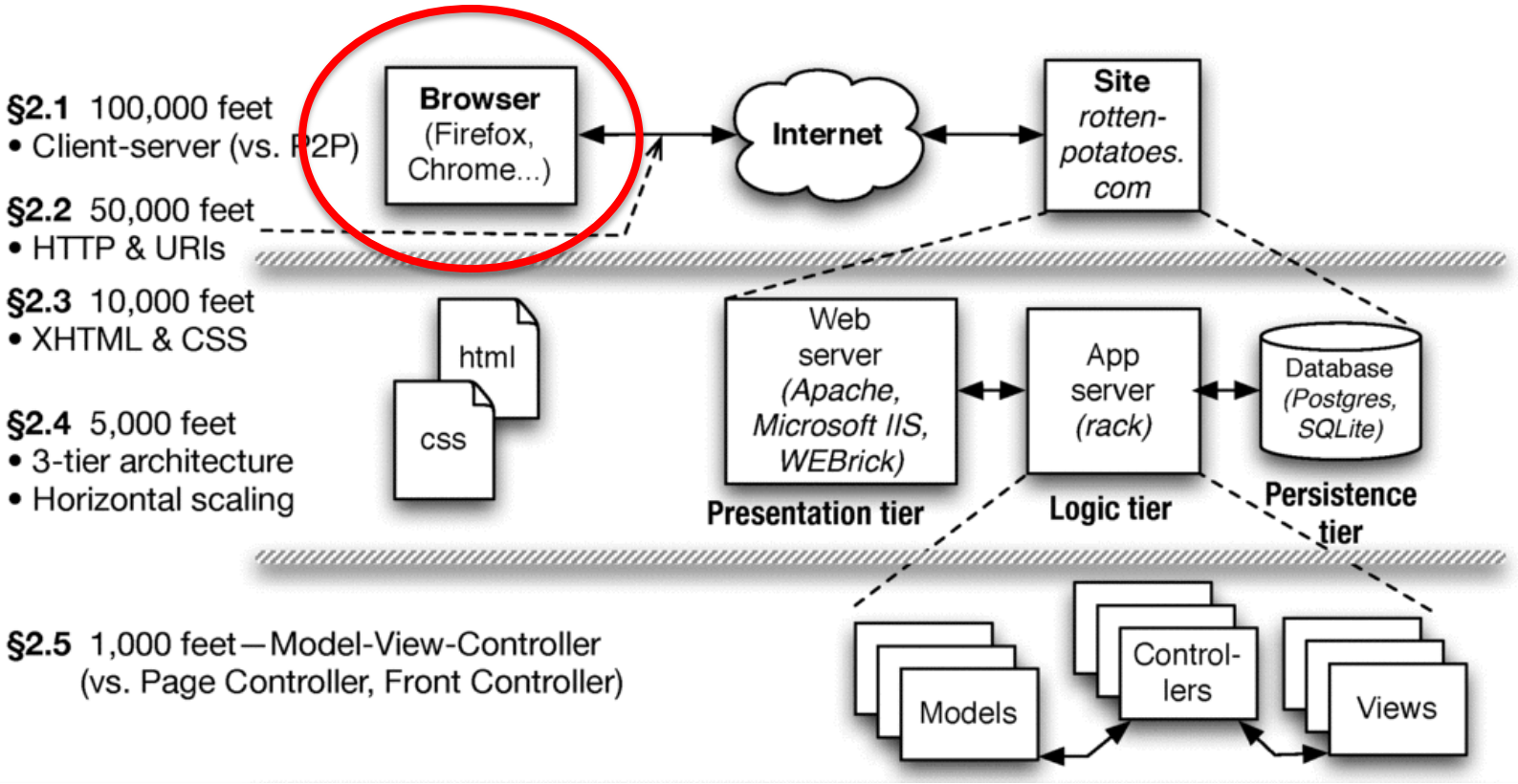
Bird’s-eye View of SaaS Architecture (2)

September 15, 2022

Byung-Gon Chun

(Credits to some slides: Armando Fox, UCB)

Frontend

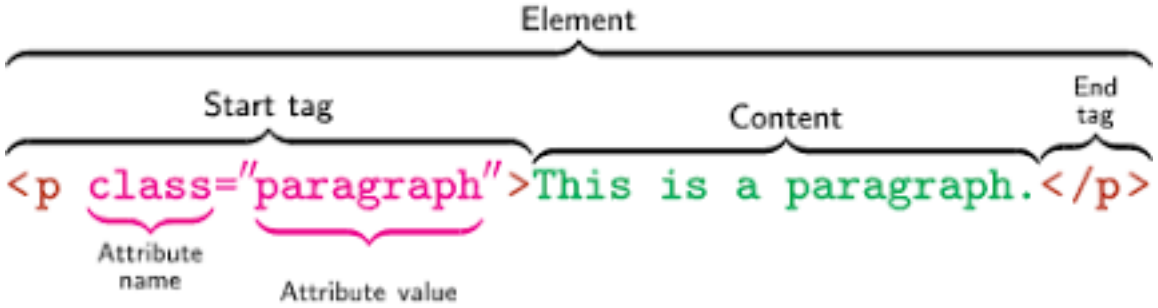

§2.1 100,000 feet
• Client-server (vs. P2P)
§2.2 50,000 feet
• HTTP & URIs
§2.3 10,000 feet
• XHTML & CSS
§2.4 5,000 feet
• 3-tier architecture
• Horizontal scaling
§2.5 1,000 feet—Model-View-Controller
(vs. Page Controller, Front Controller)
Browser
(Firefox, Chrome...)
Internet
Site
rotten-potatoes.com
html
css
Web server
(Apache, Microsoft IIS, WEBrick)
App server
(rack)
Database
(Postgres, SQLite)
Presentation tier
Logic tier
Persistence tier
Models
Control-lers
Views

Concepts You will Learn by Doing HW3 Blogging Service Frontend

- React Components
- Routing, React Router
- Flux design pattern, Redux: implementation of Flux
- Promise design pattern, axios: promise-based HTTP client
- Testing
- Use of Jest and Enzyme

HTML, CSS, JavaScript

- HTML
  - Describes and defines the content of a webpage



Content

- CSS (Cascading Style Sheets)
  - Describes how HTML elements are to be displayed on screen, paper, or in other media.

Presentation

- JavaScript

Functionality

JavaScript

- A programming language that is run by modern browsers
- A high-level, dynamically-typed, object-based, and interpreted programming language
- It can be used to control web pages on the client side of the browser, server-side programs, and even mobile applications.

Progression for the JavaScript Community

- From plain scripts

1. Adding module systems – maintainability, avoiding namespace pollution, reusability
2. Adding compilers (transpilers)
    1. Writing in a language that "thinks" the way you do makes you more productive
    2. Use next generation JavaScript, today
    3. MS TypeScript, FB Babel compiler
3. Adding type systems
    1. MS Typescript - a superset that compiles down to JavaScript — although it feels almost like a new statically-typed language in its own right
    2. FB Flow - an open-source static type checking library, incrementally add types to your JavaScript code

MS TypeScript

Add classical object-oriented semantics

```
interface User {
  name: string;
  id: number;
}

const user: User = {
  username: "Hayes",
Type '{ username: string; id: number; }' is not assignable to type 'User'.
  Object literal may only specify known properties, and 'username' does not exist in type 'User'.
  id: 0,
};
```

FB Flow

```
1 // @flow
2 function square(n: number): number {
3   return n * n;
4 }
5
6 square("2"); // Error!
```

JavaScript ES6

- JavaScript let: block-scoped variable
- JavaScript const
- JavaScript Arrow Functions
- JavaScript Classes: a template for creating objects
- Default parameter values
- …

JavaScript ES6

- JavaScript let: block-scoped variable

```
{
  var x = 2;
}
// x CAN be used here
{
  let x = 2;
}
// x can NOT be used here
```

- JavaScript const: cannot reassign once assigned

JavaScript ES6

- JavaScript Arrow Functions
  - hello = () => “Hello, World!”;
- JavaScript Classes: a template for creating objects

```
- class Car {
    constructor(brand) {
      this.brand = brand;
    }
  }
```

- Default parameter values
  - function multiply(a, b = 1) {

```
        return a * b;
    }
    console.log(multiply(5));
```

TypeScript

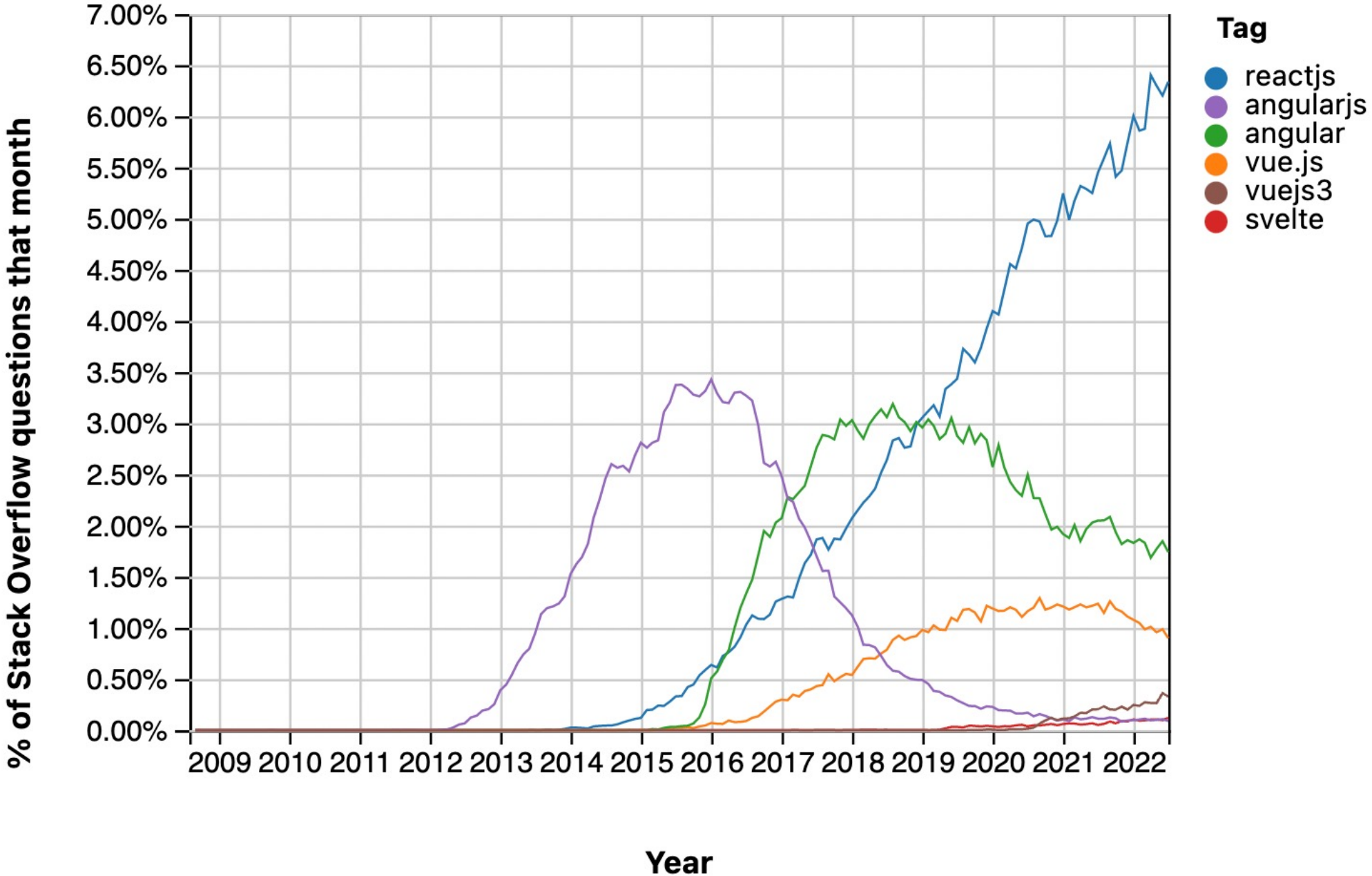
- A super-set of JavaScript
- Add classical object-oriented semantics
- Types
- Classes
- Interfaces
- Inheritance
- Modules
- Generics

Client Application Framework

- Easily implement interactive web applications
- Client-side framework

- Angular: a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript. (from Google)

- React: Javascript Library for Building User Interfaces (from FB)
  - E.g., Facebook, Instagram (basically one large React app), Netflix, …



https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs%2Cvuejs3

JSX

- JSX: a syntax extension to JavaScript to structure component rendering
  - const element = <h1>Hello, world!</h1>;
- React components are typically written in JSX, although they do not have to be
- Embedding expressions in JSX
  - <h1>{10+1}</h1> will render as <h1>11</h1>
- Conditional statements
  - { i === 1 ? 'true' : 'false' }
- After compilation, JSX expressions become regular JavaScript objects.

  E.g., => React.createElement('div');

React element

- React element: the smallest building block of React apps
  - React elements are plain objects. React DOM takes care of updating the DOM to match the React elements.
  - immutable

E.g.,

```
<div id="root"></div>


const element = <h1>Hello, world</h1>;
ReactDOM.render(element, document.getElementById('root'));
// To render a React element into a root DOM node, pass both to ReactDOM.render()
```

React component

- Let you split the UI into independent, reusable pieces
- React component
  - Take in *props* as parameters
  - Can hold *state*
  - Return how a section of the UI (User Interface) should appear via *render* method

  - function Welcome(props) { // functional component
    return <h1>Hello, {props.name}</h1>;
    }

  - class Welcome extends React.Component { // class component <- less popular due to hooks
    render() {
    return <h1>Hello, {this.props.name}</h1>;
    }
    }

React Component

- Elements can be user-defined

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- Extracting components
  - split components into smaller components

All React components must act like pure functions with respect to their props.

- Composing components

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

State

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

the fact that the Clock sets up a timer and updates the UI every second should be an implementation detail of the Clock

State and Lifecycle

Add a local state to a class

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Add lifecycle methods to a class

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

React Hooks

```jsx
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```jsx
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

React Router for Routing

- Navigational components, declarative routing
- React Router: Route, Link, BrowserRouter, Routes
- Route
  - render some *element* when its *path* matches the current URL.
- Link
  - Declarative, accessible navigation around your app
  - <Link to="/about">About</Link>

React Router for Routing

```
1   import { render } from "react-dom";
2   import {
3     BrowserRouter,
4     Routes,
5     Route,
6   } from "react-router-dom";
7   import App from "./App";
8   import Expenses from "./routes/expenses";
9   import Invoices from "./routes/invoices";
10
11  const rootElement = document.getElementById("root");
12  render(
13    <BrowserRouter>
14      <Routes>
15        <Route path="/" element={<App />} />
16        <Route path="expenses" element={<Expenses />} />
17        <Route path="invoices" element={<Invoices />} />
18      </Routes>
19    </BrowserRouter>,
20    rootElement
21  );
```

Ref. https://reactrouter.com/en/v6.3.0/getting-started/tutorial

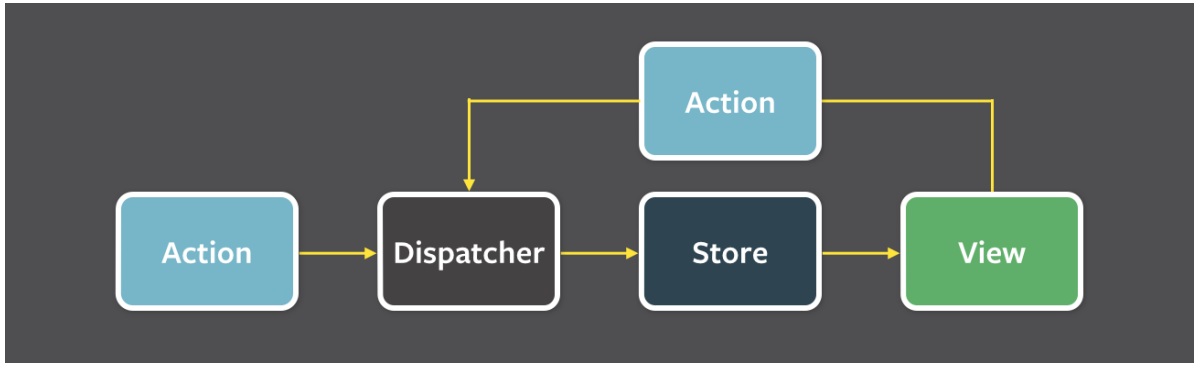
Flux: What Problem Does Flux Solve?

- Why Flux?
  - As apps grow in size and complexity, managing state inside of React components (or the *component-state paradigm*) can become cumbersome.
  - Tight coupling between user interactions and state changes
  - The function in the top-level component must describe all of the state changes that occur.

- The predecessor of Flux Facebook used was MVC.
- But, MVC did not scale well. The flow inside MVC is not well defined.

Flux Design Pattern

Force an unidirectional flow of data between components



Actions are simple objects with a type property and some data

Stores contain the application's state and logic.

The Dispatcher acts as a central hub. The dispatcher processes actions (for example, user interactions) and invokes callbacks that the stores have registered with it.

Views listen for changes from the stores and re-render themselves appropriately.
Views can also add new actions to the dispatcher, for example, on user interactions.
The views are usually coded in React, but it's not necessary to use React with Flux

How Flux is Different From MVC?

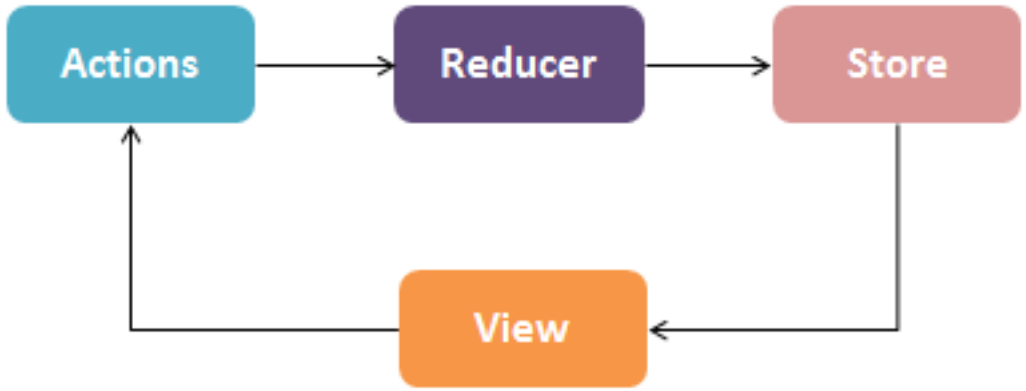
- **The Flow** of the app is essential to Flux and there are very strict rules that are enforced by the Dispatcher. In MVC the flow isn't enforced and most MVC patterns implement it differently

- **Unidirectional flow:** Every change goes through the dispatcher. A store can't change other stores directly. Same applies for views and other actions. Changes must go through the dispatcher via actions. In MVC it's very common to have bidirectional flow.

- **Stores** don't need to model anything and can store any application related state. In MVC models try to model something, usually single objects

Redux: implementation of Flux

- All of your application's data is in a single data structure called the **state** which is held in the **store**
- Your app reads the **state** from this **store**
- The **state** is never mutated directly outside the **store**
- The **views** emit **actions** that describe what happened
- A **new state** is created by combining the **old state** and the **action** by a function called the **reducer**

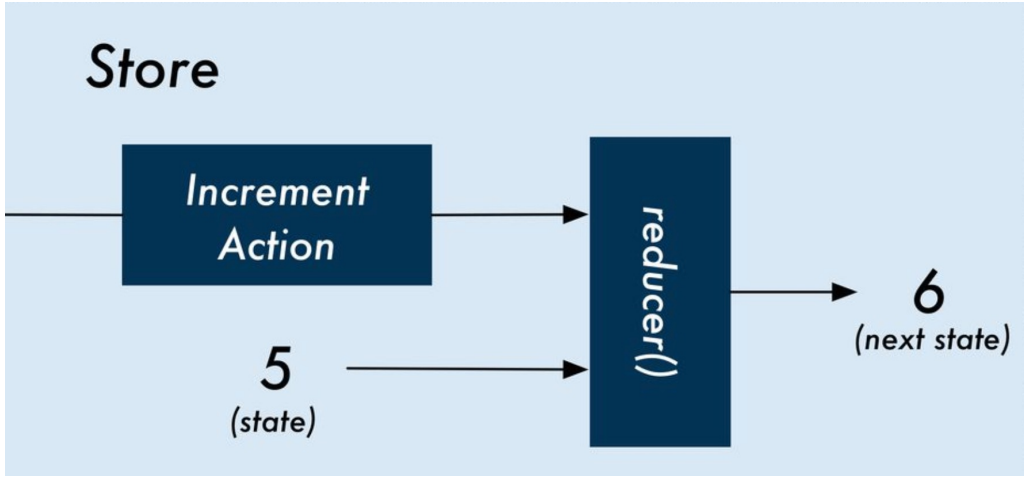


Redux Architecture

Redux Example: Counter

- Our state will be a number.
- Our actions will either be to **increment** or **decrement** the state.
- reducer

```
// Inside the store, receives `action` from the view
state = reducer(state, action);
```



Redux Example: Counter

```
function reducer(state, action) {
  if (action.type === 'INCREMENT') {
    return state + 1;
  // leanpub-start-insert
  } else if (action.type === 'DECREMENT') {
    return state - 1;
  // leanpub-end-insert
  } else {
    return state;
  }
}
```

```
const incrementAction = { type: 'INCREMENT' };

console.log(reducer(0, incrementAction)); // -> 1
console.log(reducer(1, incrementAction)); // -> 2
console.log(reducer(5, incrementAction)); // -> 6
```

* Reducer composition: a fundamental pattern of building Redux apps
- Each reducer is managing its own part of the global state

Promise: What Problem Does Promise Solve?

- JavaScript is single-threaded
- To get around this, use event-based programming (events and listeners)

```
var img1 = document.querySelector('.img-1');

img1.addEventListener('load', function() {
  // woo yey image loaded
});

img1.addEventListener('error', function() {
  // argh everything's broken
});
```

- Exact timing matters - not ideal for async success/failure

Promise Design Pattern

- Promise: async success/failure
  - A promise can only succeed or fail once. It cannot succeed or fail twice, neither can it switch from success to failure or vice versa.
  - If a promise has succeeded or failed and you later add a success/failure callback, the correct callback will be called, even though the event took place earlier.

Promise-based HTTP client: axios

- Performing GET

```
const axios = require('axios');

// Make a request for a user with a given ID
axios.get('/user?ID=12345')
  .then(function (response) {
    // handle success
    console.log(response);
  })
  .catch(function (error) {
    // handle error
    console.log(error);
  })
  .then(function () {
    // always executed
  });
```