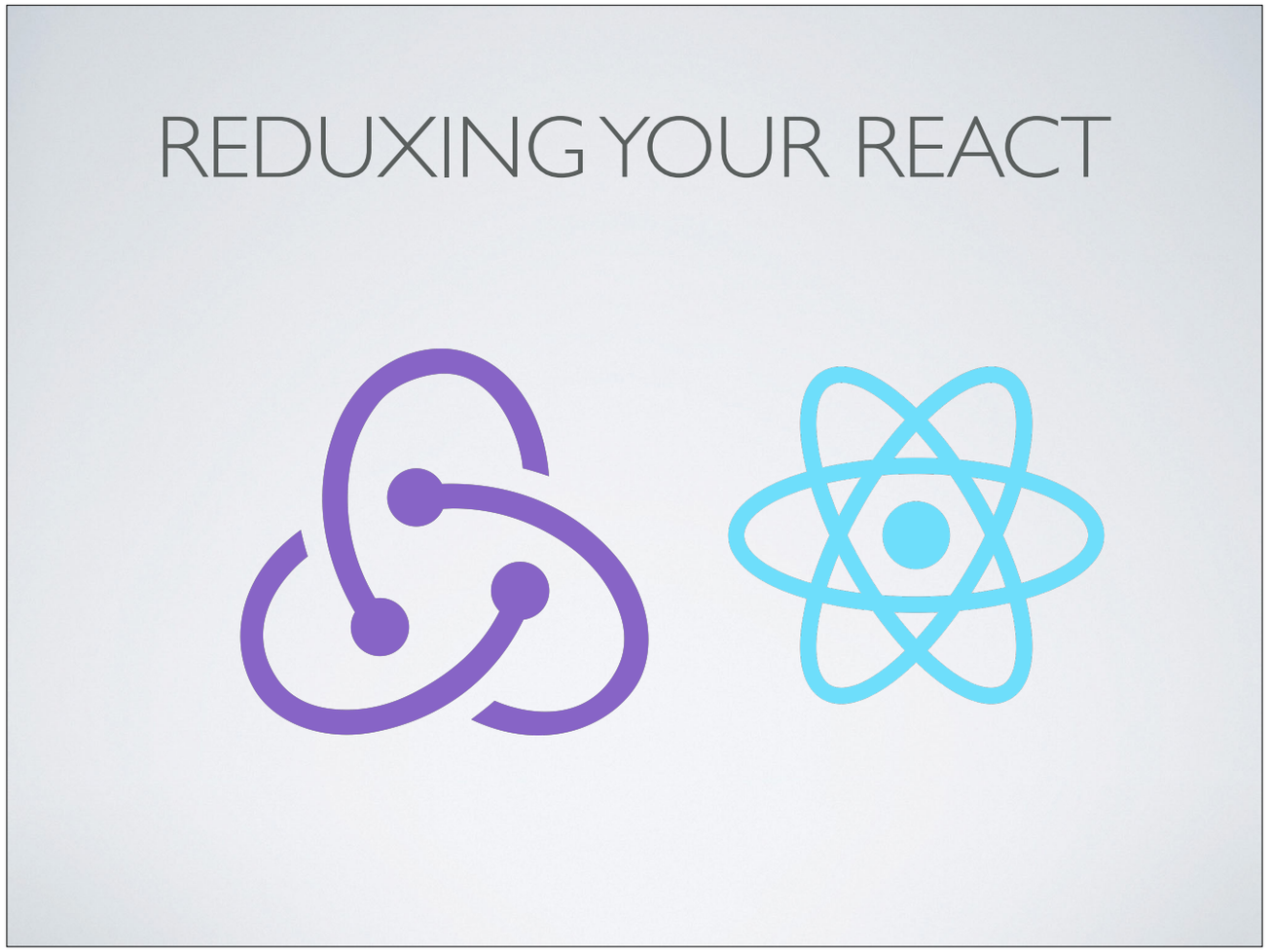


Hello jdays. I’m Alex Lakatos, a Mozilla volunteer which helps other people volunteer. I want to talk to you today about react and redux.

WHAT IS REACT?

- Declarative
- Component-Based
- Learn Once, Write Anywhere

React is a View library, not a framework.
Declarative - React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.
Component-Based - Build encapsulated components that manage their own state, then compose them to make complex UIs.
Learn Once, Write Anywhere - they don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

SMALLEST APP

```
npm install -g create-react-app
create-react-app hello-world
cd hello-world
npm start
```

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

JSX WHAT?

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

EMBEDDING EXPRESSIONS IN JSX

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Alex',
  lastName: 'Lakatos'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

You can embed any JavaScript expression in JSX by wrapping it in curly braces.
For example, 2 + 2, user.firstName, and formatName(user) are all valid expressions:

JSX IS AN EXPRESSION TOO

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

After compilation, JSX expressions become regular JavaScript objects.
This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

SPECIFYING ATTRIBUTES WITH JSX

```jsx
const element = <div tabIndex="0"></div>;
```

```jsx
const element = <img src={user.avatarUrl}></img>;
```

You may use quotes to specify string literals as attributes:
You may also use curly braces to embed a JavaScript expression in an attribute:
Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. Otherwise JSX will treat the attribute as a string literal rather than an expression. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

SPECIFYING CHILDREN WITH JSX

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

If a tag is empty, you may close it immediately with />, like XML:
JSX tags may contain children:
Caveat: Since JSX is closer to JavaScript than HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
For example, class becomes className in JSX, and tabindex becomes tabIndex.

JSX PREVENTS INJECTION ATTACKS

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

It is safe to embed user input in JSX:
By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

JSX REPRESENTS OBJECTS

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Babel compiles JSX down to React.createElement() calls.
These two examples are identical

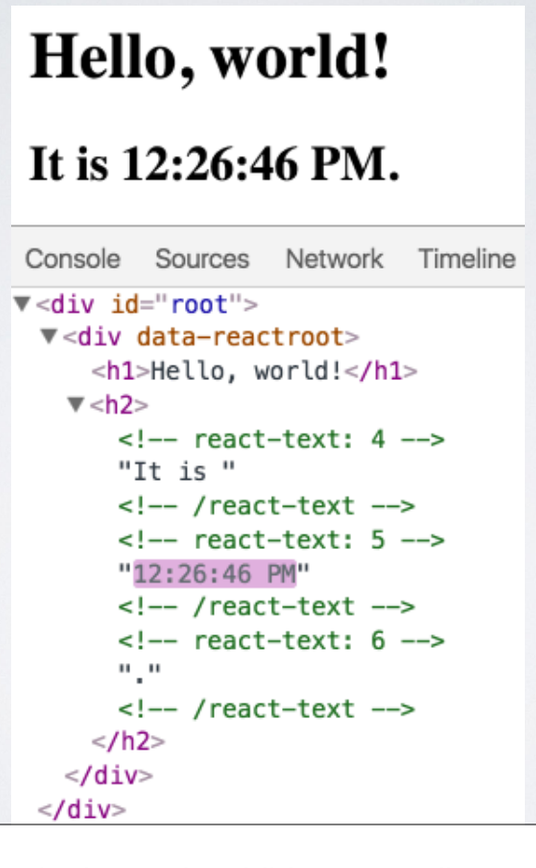
RENDERING ELEMENTS

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Elements are the smallest building blocks of React apps.
An element describes what you want to see on the screen. One might confuse elements with a more widely known concept of "components". We will introduce components in the next slides. Elements are what components are "made of"

RENDERING ELEMENTS



React elements are immutable. Once you create an element, you can't change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

With our knowledge so far, the only way to update the UI is to create a new element, and pass it to ReactDOM.render().

Even though we create an element describing the whole UI tree on every tick, only the text node whose contents has changed gets updated by React DOM.

COMPONENTS AND PROPS

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

FUNCTIONAL AND CLASS COMPONENTS

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

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

The simplest way to define a component is to write a JavaScript function.
This function is a valid React component because it accepts a single "props" object argument with data and returns a React element. We call such components "functional" because they are literally JavaScript functions.
You can also use an ES6 class to define a component.
The above two components are equivalent from React's point of view.

RENDERING A COMPONENT

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

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

Previously, we only encountered React elements that represent DOM tags:
However, elements can also represent user-defined components:
When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

For example, this code renders "Hello, jDays" on the page:

!RENDERING A COMPONENT!

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

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

Caveat: Always start component names with a capital letter.
For example, represents a DOM tag, but <Welcome /> represents a component and requires Welcome to be in scope.

COMPOSING COMPONENTS

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

function App() {
  return (
    <div>
      <Welcome name="Alex" />
      <Welcome name="Tracy" />
      <Welcome name="Chris" />
    </div>
  );
}
```

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.
For example, we can create an App component that renders Welcome many times.
Components must return a single root element. This is why we added a to contain all the <Welcome /> elements.

EXTRACTING COMPONENTS

```jsx
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Don't be afraid to split components into smaller components.
For example, consider this Comment component:
It accepts author (an object), text (a string), and date (a date) as props, and describes a comment on a social media website.
This component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it. Let's extract a few components from it.

EXTRACTING COMPONENTS

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

First, we will extract Avatar:
The Avatar doesn't need to know that it is being rendered inside a Comment. This is why I have given its prop a more generic name: user rather than author.
Docs recommend naming props from the component's own point of view rather than the context in which it is being used.

EXTRACTING COMPONENTS

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

Next, we will extract a UserInfo component that renders an Avatar next to user's name:

EXTRACTING COMPONENTS

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

This lets us simplify Comment even further.

Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps. A good rule of thumb is that if a part of your UI is used several times (Button, Panel, Avatar), or is complex enough on its own (App, FeedStory, Comment), it is a good candidate to be a reusable component.

PROPS ARE READ-ONLY

```
function sum(a, b) {
  return a + b;
}
```

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:
Such functions are called "pure" because they do not attempt to change their inputs, and always return the same result for the same inputs.
In contrast, this function is impure because it changes its own input:

React is pretty flexible but it has a single strict rule:
All React components must act like pure functions with respect to their props.

STATE AND LIFECYCLE

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Consider the ticking clock example from one of the previous slides.
So far we have only learned one way to update the UI.
We call ReactDOM.render() to change the rendered output:
In this part, we will learn how to make the Clock component truly reusable and encapsulated. It will set up its own timer and update itself every second.

STATE AND LIFECYCLE

```
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);
```

We can start by encapsulating how the clock looks:
However, it misses a crucial requirement: 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

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

Ideally we want to write this once and have the Clock update itself:
To implement this, we need to add "state" to the Clock component.
State is similar to props, but it is private and fully controlled by the component.
Components defined as classes have some additional features. Local state is exactly that: a feature available only to classes.

CONVERTING FUNCTION TO CLASS

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

You can convert a functional component like Clock to a class in five steps:

Create an ES6 class with the same name that extends React.Component.

Add a single empty method to it called render().

Move the body of the function into the render() method.

Replace props with this.props in the render() body.

Delete the remaining empty function declaration.

ADDING 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')
);
```

We will move the date from props to state in three steps:

1) Replace this.props.date with this.state.date in the render() method:
2) Add a class constructor that assigns the initial this.state:
3) Remove the date prop from the <Clock /> element

ADDING LIFECYCLE METHODS

```
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()
    });
  }
```

In applications with many components, it's very important to free up resources taken by the components when they are destroyed.
We want to set up a timer whenever the Clock is rendered to the DOM for the first time. This is called "mounting" in React.
We also want to clear that timer whenever the DOM produced by the Clock is removed. This is called "unmounting" in React.
We can declare special methods on the component class to run some code when a component mounts and unmounts:
These methods are called "lifecycle hooks".

RECAP

```
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()
    });
  }
```

Now the clock ticks every second.
Let's quickly recap what's going on and the order in which the methods are called:
1) When <Clock /> is passed to ReactDOM.render(), React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes this.state with an object including the current time. We will later update this state.
2) React then calls the Clock component's render() method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock's render output.
3) When the Clock output is inserted in the DOM, React calls the componentDidMount() lifecycle hook. Inside it, the Clock component asks the browser to set up a timer to call tick() once a second.
4) Every second the browser calls the tick() method. Inside it, the Clock component schedules a UI update by calling setState() with an object containing the current time. Thanks to the setState() call, React knows the state has changed, and calls render() method again to learn what should be on the screen. This time, this.state.date in the render() method will be different, and so the render output will include the updated time. React updates the DOM accordingly.
5) If the Clock component is ever removed from the DOM, React calls the componentWillUnmount() lifecycle hook so the timer is stopped.

USING STATE CORRECTLY

```
// Wrong
this.state.comment = 'Hello';
```

```
// Correct
this.setState({comment: 'Hello'});
```

There are three things you should know about setState().

1.Do Not Modify State Directly

First example, this will not re-render a component. Instead, use setState().

The only place where you can assign this.state is the constructor.

USING STATE CORRECTLY

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

```
// Correct
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

There are three things you should know about setState().

2. State Updates May Be Asynchronous

React may batch multiple setState() calls into a single update for performance.

Because this.props and this.state may be updated asynchronously, you should not rely on their values for calculating the next state.

First example, this code may fail to update the counter:

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

USING STATE CORRECTLY

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

There are three things you should know about setState().

3. State Updates are Merged

When you call setState(), React merges the object you provide into the current state.

For example, your state may contain several independent variables:

Then you can update them independently with separate setState() calls:

The merging is shallow, so this.setState({comments}) leaves this.state.posts intact, but completely replaces this.state.comments.

THE DATA FLOWS DOWN

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

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

This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree.

If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

To show that all components are truly isolated, we can create an App component that renders three <Clock>s:

HANDLING EVENTS

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly.

HANDLING EVENTS

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class. For example, this Toggle component renders a button that lets the user toggle between "ON" and "OFF" states.
You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.
This is not React-specific behavior; it is a part of how functions work in JavaScript.

CONDITIONAL RENDERING

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Try changing to isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application. Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

CONDITIONAL RENDERING

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
```

Inline If with Logical && Operator

CONDITIONAL RENDERING

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Inline If-Else with Conditional Operator

CONDITIONAL RENDERING

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}
```

Preventing Component from Rendering

LISTS & KEYS

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Usually you would render lists inside a component.
a component that accepts an array of numbers and outputs an unordered list of elements.
When you run this code, you'll be given a warning that a key should be provided for list items.

LISTS & KEYS

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

A "key" is a special string attribute you need to include when creating lists of elements. We'll discuss why it's important in the next section.
Let's assign a key to our list items inside numbers.map() and fix the missing key issue.

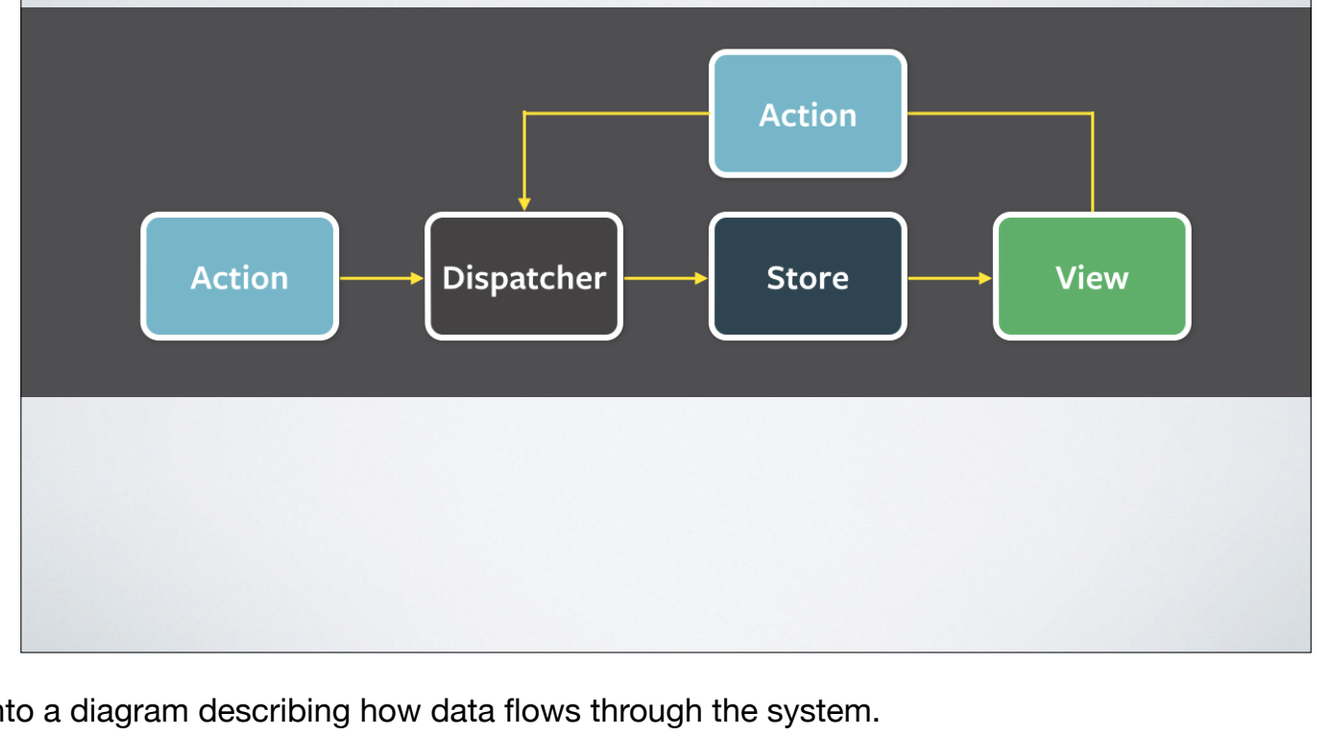
WHAT'S NEXT?

- Composition vs Inheritance
- Thinking in React
- This is just the View

1. React has a powerful composition model, and Facebook recommends using composition instead of inheritance to reuse code between components.
2. Start With A Mock. Break The UI Into A Component Hierarchy. Build A Static Version in React. Identify The Minimal (but complete) Representation Of UI State. Identify Where Your State Should Live. Add Inverse Data Flow. And That's It.
3. The whole idea behind this is that React is just the view layer. You still need routing, and some sort of model strategy. We won't go into routing today, but as far as modelling strategies go, there are 2 worth considering: Flux and Redux.

Flux is a pattern for managing data flow in your application. The most important concept is that data flows in one direction. As we go through this part we'll talk about the different pieces of a Flux application and show how they form unidirectional cycles that data can flow through.

FLUX

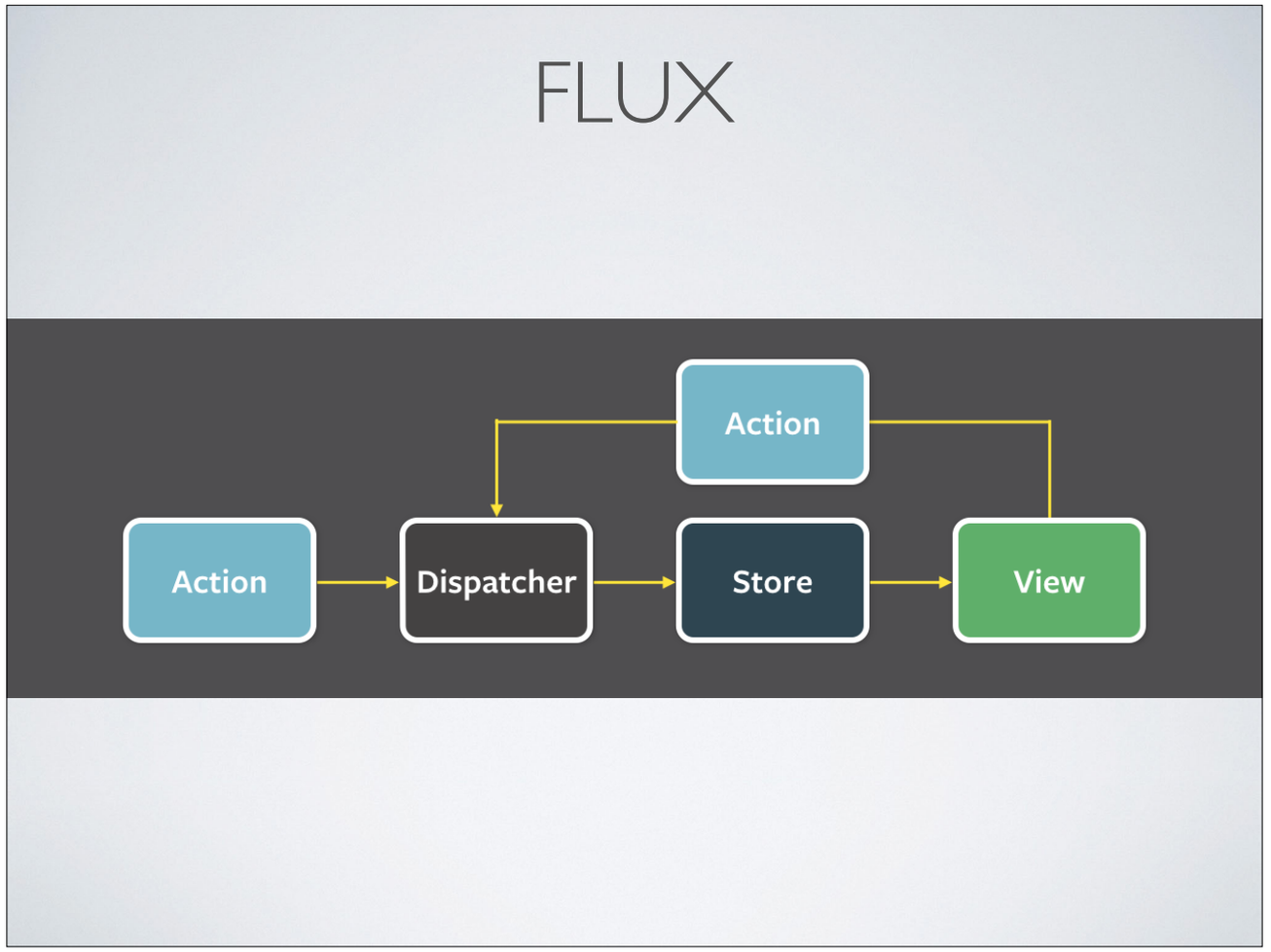


We can piece the parts of Flux above into a diagram describing how data flows through the system.

Views send actions to the dispatcher.

The dispatcher sends actions to every store.

Stores send data to the views.



The **dispatcher** receives actions and dispatches them to stores that have registered with the dispatcher. Every store will receive every action. There should be only one singleton dispatcher in each application.
A **store** is what holds the data of an application. Stores will register with the application's dispatcher so that they can receive actions. The data in a store must only be mutated by responding to an action. There should not be any public setters on a store, only getters. Stores decide what actions they want to respond to. Every time a store's data changes it must emit a "change" event. There should be many stores in each application.
Actions define the internal API of your application. They capture the ways in which anything might interact with your application. They are simple objects that have a "type" field and some data.
Data from stores is displayed in **views**. When a view uses data from a store it must also subscribe to change events from that store. Then when the store emits a change the view can get the new data and re-render. If a component ever uses a store and does not subscribe to it then there is likely a subtle bug waiting to be found. Actions are typically dispatched from views as the user interacts with parts of the application's interface.

Redux is a predictable state container for JavaScript apps.
It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.
You can use Redux together with React, or with any other view library.
It is tiny (2kB, including dependencies).

BUT WHY REDUX

- our code must manage more state than ever before
- new requirements becoming common in front-end product development
- mixing two concepts: mutation and asynchronicity
- Redux attempts to make state mutations predictable

At some point, you no longer understand what happens in your app as you have lost control over the when, why, and how of its state. When a system is opaque and non-deterministic, it's hard to reproduce bugs or add new features.
I find myself trying to manage a complexity that I have never had to deal with before, and I inevitably ask the question: is it time to give up? The answer is no.
I call them Mentos and Coke. Both can be great in separation, but together they create a mess. Libraries like React attempt to solve this problem in the view layer by removing both asynchrony and direct DOM manipulation. However, managing the state of your data is left up to you. This is where Redux enters.

CORE CONCEPTS

- Store / State
- Actions
- Reducers

Redux itself is very simple.

CORE CONCEPTS - STORE

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

Imagine your app’s state is described as a plain object. For example, the state of a todo app might look like this:
This object is like a “model” except that there are no setters. This is so that different parts of the code can’t change the state arbitrarily, causing hard-to-reproduce bugs.

CORE CONCEPTS - ACTIONS

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

To change something in the state, you need to dispatch an action. An action is a plain JavaScript object (notice how we don’t introduce any magic?) that describes what happened. Here are a few example actions:
Enforcing that every change is described as an action lets us have a clear understanding of what’s going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened.

CORE CONCEPTS - REDUCERS

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter;
  } else {
    return state;
  }
}

function todos(state = [], action) {
  switch (action.type) {
  case 'ADD_TODO':
    return state.concat([{ text: action.text, completed: false }]);
  case 'TOGGLE_TODO':
    return state.map((todo, index) =>
      action.index === index ?
        { text: todo.text, completed: !todo.completed } :
        todo
   )
  default:
    return state;
  }
}
```

Finally, to tie state and actions together, we write a function called a reducer. Again, nothing magic about it—it's just a function that takes state and action as arguments, and returns the next state of the app. It would be hard to write such a function for a big app, so we write smaller functions managing parts of the state:

CORE CONCEPTS - REDUCERS

```
function todoApp(state = {}, action) {
  return {
    todos: todos(state.todos, action),
    visibilityFilter: visibilityFilter(state.visibilityFilter, action)
  };
}
```

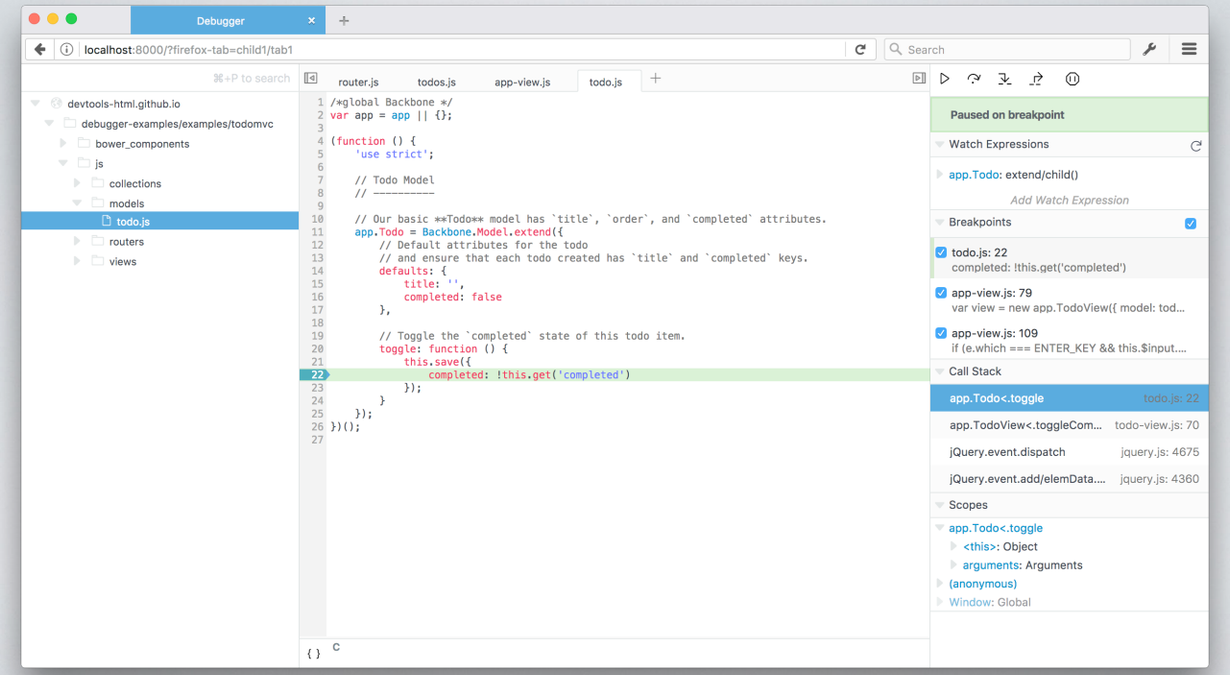
And we write another reducer that manages the complete state of our app by calling those two reducers for the corresponding state keys:
This is basically the whole idea of Redux. Note that we haven’t used any Redux APIs. It comes with a few utilities to facilitate this pattern, but the main idea is that you describe how your state is updated over time in response to action objects, and 90% of the code you write is just plain JavaScript, with no use of Redux itself, its APIs, or any magic.

3 PRINCIPLES

- Single source of truth
- State is read-only
- Changes are made with pure functions

The state of your whole application is stored in an object tree within a single store.
The only way to change the state is to emit an action, an object describing what happened.
To specify how the state tree is transformed by actions, you write pure reducers.
That's it! Now you know what Redux is all about.

DEBUGGER.HTML
(OR MY SHAMELESS SALES PITCH)



https://devtools-html.github.io/debugger.html/

debugger.html is a hackable debugger for modern times, built from the ground up using React and Redux. It is designed to be approachable, yet powerful. And it is engineered to be predictable, understandable, and testable.

Mozilla created this debugger for use in the Firefox Developer Tools. And we've purposely created this project in GitHub, using modern toolchains. We hope to not only to create a great debugger that works with the Firefox and Chrome debugging protocols but develop a broader community that wants to create great tools for the web.

And if you can't introduce React + Redux into your company, here is how you practice it :)

TACK!

@Lakatos88

Alex Lakatos

Thank you & a Mozilla Tech Speaker tradition: on stage selfieee time.
Questions? I'll be around tomorrow as well, feel free to ask me anything React, Redux or Mozilla related. PS, I have some Mozilla swag with me ;)