Intro to React, Redux, and TypeScript

Mark Erikson
@acemarke
December 2020

Who Am I?

Answerer of Questions

@acemarke
Twitter, Reactiflux, Reddit, HN

@markerikson
Stack Overflow, Github

Collector of Interesting Links

React/Redux Links

Redux Ecosystem Links

Who Am I?

Writer of Very Long Posts

blog.isquaredsoftware.com

"Idiomatic Redux" opinions series

A (Mostly) Complete Guide to React Rendering Behavior

Redux Maintainer

Redux Docs

- **Redux Essentials tutorial**
- **Redux FAQ**
- **Structuring Reducers**
- **Redux Style Guide**

Redux Libraries

- Wrote React-Redux v7
- Directed React-Redux v7.1 hooks API
- Created Redux Toolkit

Who Am I?

That Guy With The Simpsons Avatar

Notes

- The code samples use the latest modern ES6+ JavaScript syntax, such as arrow functions and destructuring. None of those are required to use React, but most React applications use ES6 and newer features.
- Some of the code samples are shortened or formatted differently in order to fit them into the slides, and aren't representative of production code.
- I am a Redux maintainer, and spend most of my time answering questions about React and Redux. So, I'm *just* a bit biased :)
- Lots of info in these slides:
  - Intended to be an overview, not a complete reference

Prerequisites

These slides assume knowledge of modern JS syntax (ES6+), async JS, HTML, and CSS. See these resources for background on relevant concepts.

How Web Apps Work series

Overviews of key web dev terms, technologies, and concepts:

- HTTP and Servers
- Client Development and Deployment
- Browsers, HTML, and CSS
- JavaScript and the DOM
- AJAX, APIs, and Data Transfer

JavaScript for Java Devs

A cheatsheet to modern JS syntax and concepts, along with key tools in the JS ecosystem:

- JavaScript for Java Devs

Using React to Create User Interfaces

JavaScript Frameworks

Every framework can be viewed as an attempt to say "the hardest part of writing a webapp is $X, so here's some code to make that easier".

Knockout.js is basically what you get when a dev says "the hardest part of writing a webapp is implementing two-way data binding", which you can tell because that's basically 90% of what the framework does.

Similarly, Backbone is the result of feeling like the hard parts are fetching and persisting models to a REST API, and client side routing; that's basically all it does. Figuring out how to turn your models into HTML is easy (apparently), but models are hard, so it helps you.

- *Reddit user /u/Cody_Chaos*

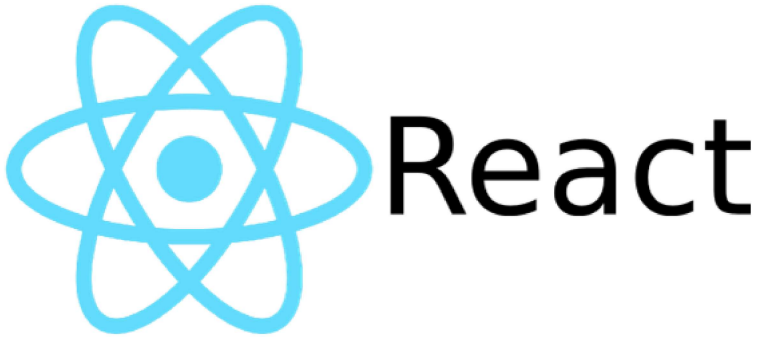
JavaScript Frameworks

> **Angular is what you get if you think the biggest problem with writing webapps is that Javascript isn't Java; Ember that it's not Ruby.** (I kid. But I'm less familiar with those frameworks.) And so on. Everyone has their own ideas of what's hard to solve.
>
> **React + Redux is based on the idea that what's really hard with writing webapps is non-deterministic behaviour and unclear data flow.** And if you've worked on a large Knockout or Backbone project, you're probably inclined to agree.
>
> - *Reddit user /u/Cody_Chaos*

JavaScript Frameworks

- **Why do frameworks exist?**
  - Keep state out of the DOM
  - Higher-level abstractions
  - Code organization
- **Pros**
  - Common concepts that can be shared between apps and developers
  - Large communities, shared knowledge, documentation, bug fixes
  - Better app structure through tools and guidelines
- **Cons**
  - Learning curve
  - Minimum requirements for size
  - Setup and infrastructure

What Is React?



- "A Javascript library for creating user interfaces"
- "The 'V' in 'MVC'"
- "A library, not a framework"

What Is React?

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. Declarative views make your code more predictable and easier to debug

Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

- *React Documentation*

Declarative Code

Imperative

```
$('form').on('submit', function (e) {
  e.preventDefault()
  $.ajax({
    url: '/customers',
    type: 'POST',
    data: $(this).serialize(),
    success: function (data) {
      $('.status').append('<h3>' + data + '</h3>')
      $('#banner').show()
    }
  })
})
```

Declarative

```
function NoteBox() {
  const [data, setData] = useState([]);

  const handlePost = () => {};

  return (
    <div className="NoteBox">
      <h1>Notes</h1>
      <NoteList data={data} />
      <NoteForm onPost={handlePost} />
    </div>
  );
}
```

- React is an abstraction away from the DOM
- Encourages you to think of your application and UI in terms of *state*, rather than explicit UI manipulations
- Allows a simplified mental model for data flow: "re-render the whole app on every update"
- Mix and match reusable components to build UIs

Managing the UI

Declarative Rendering

- Completely recreating the entire UI on every update is not efficient
- Rendering a component returns *descriptions* of what the UI should look like now
- React uses those descriptions to update the UI efficiently

One-Way Data Flow

- Components pass data to their children
- Component rendering is based on internal state plus data from parent
- Predictable top-down data flow makes it easier to understand reason for UI contents

Choosing React

Why Is This Compelling?

> Declarative → Predictable → Confidence → Reliability
>
> *ref: Tom Occhino's ReactConf keynote*

What's the Learning Curve?

- Learning React's basic concepts and API: **EASY**
- Learning to "think in React": **INTERMEDIATE**
- Stuff you can do with React: **ADVANCED**

React Components

Components

Components === State Machines

React thinks of UIs as simple state machines. By thinking of a UI as being in various states and rendering those states, it's easy to keep your UI consistent. In React, you simply update a component's state, and then render a new UI based on this new state. React takes care of updating the DOM for you in the most efficient way

Components === Functions

Just like functions take parameters and return a result, components take in values and return UI output. Given the same input values, a component will return the same UI output. This is often described as `UI = f(state)`

Hello World Component

LIVE PREVIEW

Hello World!

SOURCE CODE

```
function HelloWorld() {
  return <div>Hello World!</div>;
}

render(<HelloWorld />);
```

Declaring Components

Function Components (modern)

```
import React from "react";

// React components are written as functions
function FunctionalComponentSyntax(props) {
  return <div>Hello World!</div>;
}

// Or, use ES6 arrow function syntax to declare the
component
const ArrowFunctionComponent = props => {
  return <div>Hello World!</div>;
};
```

Class Components (legacy)

```
import React from "react";

// Class components use ES6 class syntax
// Function components are now considered standard
class ClassComponent extends React.Component {
  render() {
    return <div> Hello World!</div>;
  }
}

// In React <= 15, this was available as
// React.createClass(). It's now a separate package,
// but deprecated and should not be used
import createReactClass from "create-react-class";

const CreateClassComponent = createReactClass({
  render: function () {
    return <div> Hello World!</div>;
  },
});
```

Basic Usage

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

function HelloWorld() {
  return <div>Hello World!</div>;
}

const parentNode = document.getElementById("root");
ReactDOM.render(<HelloWorld />, parentNode);
```

- A React function component will be called by React. This process is known as **“rendering”**
- When a component renders, it returns a tree of React component descriptions, which will eventually be turned into DOM nodes in the page
- The `ReactDOM.render()` method creates a new component tree, initializes the React library, and appends the generated DOM output from the component and its children to the parent DOM node

JSX Syntax

```
// Before
const MyComponent = (props) => (
    <div>Hello World!</div>
);

ReactDOM.render(
    <MyComponent />,
    document.getElementById("root")
);

//After
const MyComponent = (props) => (
    React.createElement("div", null, "Hello World")
);

ReactDOM.render(
    React.createElement(MyComponent),
    document.getElementById("root")
);
```

- JSX is syntax sugar for nested function calls. Not required to use React, but the standard approach. Requires compilation process, usually with Babel.
- JSX “tags” are turned into `React.createElement()` calls (which could be written by hand without JSX). Those calls return plain JS objects describing the output.
- React “element” objects look like: `{type : thingToRender, props : {}, children : []}`

JSX Syntax

```
// Note use of className for HTML classes
const MyComponent = (props) => {
    const someString = "Something else to display";

    return (
        <div className="class1 class2">
            Hello World!
            {/* <div>Commented out code</div> */}
            <div>{someString}</div>
            <div>{28 + 14}</div>
        </div>
    );
}

// Illegal - can only return one value!
const BrokenComponent = () => (
    <div />
    <div />
);

const WorkingComponent1 = () => (
    [<div />, <div />]
);

const WorkingComponent2 = () => (
    <React.Fragment><div /><div /></React.Fragment>
);
```

- Curly braces “escape” from JSX back into normal JS expressions. They are used to insert values from variables and comment out code
- A few attribute names differ from plain HTML. In particular, use `className` instead of `class` for HTML/CSS class values, and `<label htmlFor="someInput">`.
- Render logic can return a single root element, an array, a string, or a number
- To return multiple items, use `<React.Fragment>` as a pseudo “parent” (which will not add any extra DOM nodes around the content)

JSX Gotchas

```
// Note use of className for HTML classes
const MyComponent = (props) => {
    const someString = "Something else to display";

    return (
        <div className="class1 class2">
            Hello World!
            {/* <div>Commented out code</div> */}
            <div>{someString}</div>
            <div>{28 + 14}</div>
        </div>
    );
}

// Illegal - can only return one value!
const BrokenComponent = () => (
    <div />
    <div />
);

const WorkingComponent1 = () => (
    [<div />, <div />]
);

const WorkingComponent2 = () => (
    <React.Fragment><div /><div /></React.Fragment>
);
```

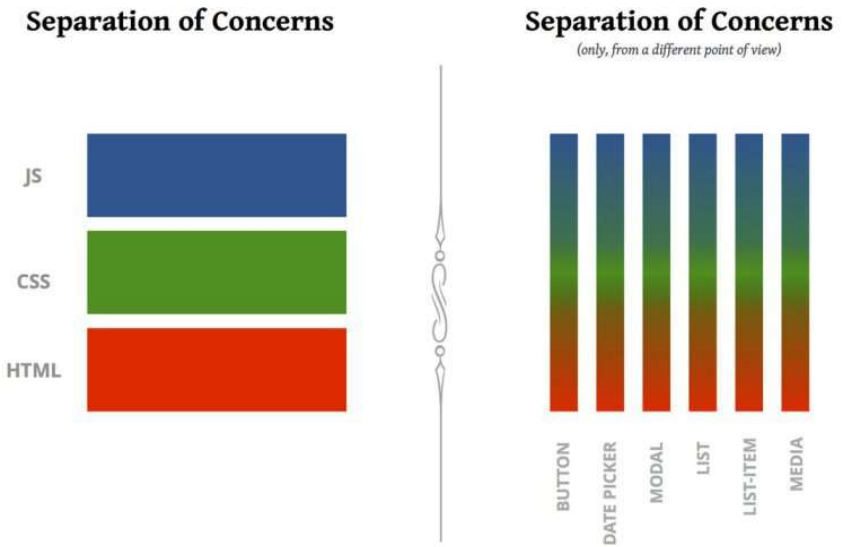
- JSX uses capitalization to differentiate between HTML elements and React components.
- If the first letter is lowercase, the tag is assumed to be an HTML element, and turned into a string. If it's uppercase, the tag is assumed to be a variable name in scope (usually a component).
- A common mistake is to give a variable a `camelCase` name and render it, like `<myComponent>`. React turns that into `{type : "myComponent"}`, which breaks, instead of `{type : MyComponent}`.)

HTML in my JavaScript?!?!?

"Separation of Concerns"?

> Templates encourage a poor separation of concerns. "View Model" tightly couples a template to display logic. Display logic and markup are inevitably tightly coupled. **Templates separate technologies, not concerns**.
>
> React components are **"...a highly cohesive building block for UIs loosely coupled with other components."**
>
> - *Pete Hunt: React: Rethinking Best Practices*

"Separation of Concerns"?



- If code frequently changes together, it should stay together
- React allows you to focus on building components, not templates
- Combining “markup” and JavaScript reduces context switching
- Full power of Javascript for rendering logic

Passing Data As Props

LIVE PREVIEW

Hello Mark

SOURCE CODE

```
function HelloWorld(props) {
  return <div>Hello {props.name}</div>;
}

render(<HelloWorld name="Mark" />);
```

Component Props

```jsx
function HelloWorld(props) {
  return <div>Hello {props.name}</div>;
}

function Parent1() {
  return <HelloWorld name="Mark" />;
}

function KitchenSinkParent() {
  return (
    <KitchenSinkChild
      a={1}
      b="some text"
      c={{ meaningOfLife: 42 }}
      d={[2, 3, 4, 5]}
      e={<SomeOtherComponent />}
      f={AnotherComponent}
      g={() => {
        console.log("Clicked!");
      }}
    />
  );
}
```

- **Props** are values passed from parent to child. Think of them as arguments being passed into a function (which is actually the case!)
- Props are combined into a single object. That `props` object is the only parameter for function components. (For class components, it's available as `this.props` .)
- Props are **read-only** inside of a component, in the same way that a plain function should not modify its arguments.
- Anything can be passed as a prop: primitives, objects, arrays, functions, component types, JSX elements, and more.

React Events - Deja Vu?


```
function MyComponent() {
  const handleClick = e => {
    console.log("Clicked: ", e.target.name);
  };

  return (
    <button onClick={handleClick} name="hello">
      Click Me!
    </button>
  );
}
```

- React manages event handling internally.
  - Event handlers declared in a component are automatically managed by React, using "event delegation" to simplify handling and improve performance.
  - It also normalizes event behavior across browsers.
- React components create event handler callbacks and pass *references* to those callbacks as props to child components.
- Event handler prop names are `camelCased`: `onClick`, `onMouseMove`

Component State: The `useState` Hook

LIVE PREVIEW

Button was clicked:
0 times
Click Me

SOURCE CODE

```
function Counter() {
  const [counter, setCounter] = useState(0);

  const onClick = () => {
    setCounter(counter + 1);
  };

  return (
    <div>
      Button was clicked:
      <div>{counter} times</div>
      <button onClick={onClick}>Click Me</button>
    </div>
  );
}

render(<Counter />);
```

Component State: The `useState` Hook

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

function Counter() {
  // Import the `useState` hook from React
  // Pass in an initial state value, used on first
render
  // Returns a 2-element JS array:
  // - array[0]: current value
  // - array[1]: setter function
  // "Destructuring" creates local variables from array
  const [counter, setCounter] = useState(0);

  const onClick = () => {
    setCounter(counter + 1);
  };

  return (
    <div>
      Button was clicked:
      <div>{counter} times</div>
      <button onClick={onClick}>Click Me</button>
    </div>
  );
}
```

- **Hooks** allow React function components to have internal state and run additional logic after rendering
- The **`useState`** hook allows components to store and update a value internally
  - `useState` accepts an *initial* state value, and returns an array with 2 things inside: the *current* value, and a setter function
  - Calling `setState(newValue)` will queue a re-render with the new value
- Normally use ES6 “array destructuring” to read the two values into local variables, using whatever names you want for them

Component State: The `useState` Hook

```
function Counters() {
  // Multiple calls to useState in one component
  const [timesClicked, setTimesClicked] = useState(0);
  const [counters, setCounters] = useState({ a: 0, b: 0
});

  const handleClick = name => () => {
    setTimesClicked(timesClicked + 1);
    // Must create the new object value, immutably
    setCounters({
      ...counters,
      [name]: counters[name] + 1,
    });
  };

  return (
    <div>
      Button was clicked:
      <div>{timesClicked} times</div>
      <button onClick={handleClick("a")}>Button A
({counters.a})</button>
      <button onClick={handleClick("b")}>Button B
({counters.b})</button>
    </div>
  );
}
```

- Can call `useState` many times in one component
- Calling `setState()` replaces existing value
  - Can have objects instead of primitives, but you must create a new object immutably and pass it in yourself
- `setState()` always requires a new value by reference
  - If the value passed to `setState` hasn't changed, React will skip the re-render

Component State: `useState` Tips

```
function Counter() {
  const [counter, setCounter] = useState(0);

  const onClickBuggy = () => {
    // ❌ Two bugs here!
    // 1) "Closed over" old value of `counter`
    // 2) Updates will be batched together
    setCounter(counter + 1);
    setCounter(counter + 1);
  };

  const onClickFixed = () => {
    // ✅ Can pass an "updater" function to setState,
    // which gets the latest value as its argument
    setCounter(prevCounter => prevCounter + 1);
    setCounter(prevCounter => prevCounter + 1);
  };

  return (
    <div>
      Button was clicked:
      <div>{counter} times</div>
      <button onClick={onClickFixed}>Click Me</button>
    </div>
  );
}
```

- Creating callbacks “closes over” variable values at time of render
  - Can cause bugs due to “stale state” if not careful!
- Be careful with multiple update calls in a row!
- `setState` can accept an “updater” callback instead of a value.
  - Updater will be called with the latest value as its argument, and should return a new value. Safer to use this if multiple updates depend on each other
  - Also can avoid having to capture value from parent scope

Rendering Lists

LIVE PREVIEW

Sort List | Scotts Only | Reset List

- Scott Hanselman
- John Papa
- Scott Guthrie
- Dan Wahlin
- Debora Kurata
- Zoiner Tejada
- Scott Allen
- Elijah Manor
- Ward Bell

SOURCE CODE

```
function ListsExample({speakers}) {
  const [shouldSort, setShouldSort] = useState(false);
  const [filter, setFilter] = useState("");

  const onSortClicked = () => {
    setShouldSort(true);
  };

  const onScottsClicked = () => {
    setFilter("Scott");
  };

  const onResetClicked = () => {
    setShouldSort(false);
    setFilter("");
  };

  let speakersToDisplay = speakers;

  if (filter) {
    speakersToDisplay = speakersToDisplay.filter(name => name.startsWith(filter));
  }

  if (shouldSort) {
    speakersToDisplay = speakersToDisplay.slice().sort()
  }


  const speakerListItems = speakersToDisplay.map(speaker => (
```

Rendering Lists

```
function ListsExample({speakers}) {
  const [shouldSort, setShouldSort] = useState(false);
  const [filter, setFilter] = useState("");

  const onSortClicked = () => {
    setShouldSort(true);
  };

  const onScottsClicked = () => {
    setFilter("Scott");
  };

  const onResetClicked = () => {
    setShouldSort(false);
    setFilter("");
  };

  let speakersToDisplay = speakers;

  // Prefer deriving data while rendering, vs storing derived values
  if (filter) {
    speakersToDisplay = speakersToDisplay.filter(name =>
name.startsWith(filter));
  }

  const speakerListItems = speakersToDisplay.map(speaker => (
    <li key={speaker}>{speaker}</li>
  ));
}
```

- List items must have **keys**, which tell React list item identity
  - Should be unique per list
  - Ideally, use item IDs
  - Fallback, use array indices, but only if data won't reorder
- General tip: prefer "deriving data" while rendering, vs storing derived data in state
  - store description of how to filter or sort, not filtered/sorted values

It's Just JavaScript

- JSX looks like HTML, but is converted to plain JS function calls
- No special “template syntax” for looping or conditions - use normal JS logic and syntax!
  - Conditional rendering: `if/else`, ternary operator
  - Looping: `array.map()`, `for`

Render Logic

```
// My basic render function structure:
function RenderLogicExample({
  someBoolean, // 1) Destructure values from `props`
object
  someList,
}) {
  // 2) Declare state values
  const [a, setA] = useState(0);
  const [b, setB] = useState(0);

  // 3) Render any dependent items into temporary
variables,
  //    such as conditional components or lists
  const conditionalComponent = someBoolean ? (
    <SomeConditionalComponent />
  ) : null;
  const listItems = someList.map(item => (
    <ListItem key={item.id} item={item} />
  ));

  // 4) Use a single JSX structure filled with content
  return (
    <div>
      <div>
        A: {a}, B: {b}
      </div>
      {conditionalComponent}
      {listItems}
    </div>
  );
}
```

- Render methods can contain arbitrary logic. Different people structure their render methods in different ways. Many use inline logic in their JSX structures, including ternary statements and array mappings.
- My preferred approach for rendering is to keep all logic outside of the main JSX structure, for clarity
- Render logic must be “**pure**“, without any “side effects”
  - No AJAX calls, mutating data, random values
  - **Rendering should only be based on current props and state**

Parent/Child Communication

LIVE PREVIEW

Sort List | Scotts Only | Reset List

- Scott Hanselman
- John Papa
- Scott Guthrie
- Dan Wahlin
- Debora Kurata
- Zoiner Tejada
- Scott Allen
- Elijah Manor
- Ward Bell

SOURCE CODE

```
// ES6 arrow function - either function syntax is fine
const SpeakerListItem = ({ speaker, selected, onClick }) => {
  const itemOnClick = () => onClick(speaker);

  let content = speaker;

  if (selected) {
    content = (
      <b>
        <i>{speaker}</i>
      </b>
    );
  }

  return <li onClick={itemOnClick}>{content}</li>;
};

function ListSelectionExample({speakers}) {
  const [shouldSort, setShouldSort] = useState(false);
  const [filter, setFilter] = useState("");
  const [selectedSpeaker, setSelectedSpeaker] = useState(null);

  const onSpeakerClicked = speaker => {
    setSelectedSpeaker(speaker);
  };

  const onSortClicked = () => {
    setShouldSort(true);
  };
```

Parent/Child Communication

```
const SpeakerListItem = ({ speaker, selected, onClick }) => {
  // Child uses callback from parent
  const itemOnClick = () => onClick(speaker);

  // Callback used to notify parent
  return <li onClick={itemOnClick}>{content}</li>;
};

function ListSelectionExample({speakers}) {
  const [selectedSpeaker, setSelectedSpeaker] = useState(null);

  // Parent creates callback that queues state update
  const onSpeakerClicked = speaker => {
    setSelectedSpeaker(speaker);
  };

  // Omit derived filtered/sorted values
  let speakersToDisplay = speakers;

  const speakerListItems = speakersToDisplay.map(speaker => (
    // Values and callbacks passed to child as props
    <SpeakerListItem
      key={speaker}
      speaker={speaker}
      selected={speaker === selectedSpeaker}
      onClick={onSpeakerClicked}
    />
  ));
}
```

- Parents pass data as props to children
- Parents pass callbacks to children as props, children communicate to parent by running `props.somethingHappened(data`

Component Effects: The `useEffect` Hook

```
function EffectExample() {
  const [count, setCount] = useState(0);

  // The useEffect hook takes a callback function,
  // which may contain side effects
  useEffect(() => {
    // Updating the document title is a side effect
    // (updating the mutable contents of the page)
    // Can reference a captured variable inside here
    document.title = `You clicked ${count} times`;
  });

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

- Component render logic is not allowed to contain side effects
- Components can run additional logic that *can* have side effects, *after* the rendering is done
- The **`useEffect`** hook allows components to run logic that has side effects
- By default, `useEffect` runs its callback after *every* render
  - Effects run on brief timeout after render is complete

Component Effects: The `useEffect` Hook

```
function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  const friendId = props.friend.id;
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    FriendsAPI.subscribe(friendId, handleStatusChange);

    // Effect hook callbacks can return a cleanup
function
    return () => {
      FriendsAPI.unsubscribe(friendId,
handleStatusChange);
    };

    // Optional dependencies array:
    // only runs callback when deps values change
    // Think of this as "syncing effects to state"
  }, [friendId]);

  // omit rendering
}
```

- Effect callbacks can return a cleanup function
  - Typical example: subscribe to an event, then unsubscribe in cleanup
  - Cleanup function will run before unmount, but also if effect re-runs
- Effect hook accepts a “dependencies array” argument to limit when the effect callback runs
- Concept: “What state does this effect sync with?”
  - `useEffect(fn)`: “all state” (run callback after every render)
  - `useEffect(fn, [])`: “all state” (only run callback on mount)
  - `useEffect(fn, [value1, value2])`: “these values” (run on mount, and when any value changes)

Rules of Hooks

All React hooks usage **must** follow two important rules!

- **Only call hooks at the top level of a function component**
  - May not call hooks inside conditional logic, loops, or nested functions
  - Reason: React needs to track exact order of hooks used, and that order must match every time a function component is rendered
- **Do not call hooks outside of React function components**
  - Do not call hooks in plain JS (non-component) functions
  - May call hooks inside of "custom hooks", but those must also follow hooks rules when used

React "Rules of Hooks" ESLint plugin warns about these - do what it says!

(Note: some hooks data dependency issues can be tricky to solve)

Custom Hooks

```
function FriendListItem(props) {
  // What if we want to reuse the "is online" logic?
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function statusChanged(status) {
      setIsOnline(status.isOnline);
    }

    FriendsAPI.subscribe(props.friend.id,
statusChanged);

    return () => {
      FriendsAPI.subscribe(props.friend.id,
statusChanged);
    };
  });

  return (
    <li style={{ color: isOnline ? "green" : "black" }}>
{props.friend.name}</li>
  );
}
```

- Need some way to share and reuse logic between components
  - If writing normal JS code, just “extract a function”
- Can do the same thing with hooks-related code

Custom Hooks

```
// "Custom hook": extracted function starting with "use"
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function statusChanged(status) {
      setIsOnline(status.isOnline);
    }

    FriendsAPI.subscribe(props.friend.id,
statusChanged);

    return () => {
      FriendsAPI.subscribe(props.friend.id,
statusChanged);
    };
  });

  return isOnline;
}

function FriendsList(props) {
  const isOnline = useFriendStatus(props.friend.id);
  // etc
}

function FriendsListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);
  // etc
}
```

- Can extract hooks-related code into a new “custom hook” function
- Just a function that calls React hooks
- Name must start with `use`
- Many React libraries now ship their own custom hooks
  - React-Redux: `useSelector`, `useDispatch`
  - React-Router: `useHistory`, `useLocation`, `useParams`
  - Others: `useClickOutside`, `useWindowSize`, `useFormState`, etc

Other Built-In Hooks

```
function OtherHooksExample(props) {
  const divRef = useRef();
  const counterRef = useRefo(0);

  useLayoutEffect(() => {
    console.log("Div width: ", divRef.current.width);
  }, [props.value]);

  useLayoutEffect(() => {
    renderCounterRef.current += 1;
    console.log(`Rendered ${counterRef.current}
times!`);
  });

  const data = useMemo(() => {
    return calculateExpensiveData(props.value);
  }, [props.value]);

  const callback = useCallback(() => {
    console.log("Clicked!");
  }, [props.value]);

  return (
    <div ref={divRef}>
      <ChildComponent onClick={callback} data={data} />
    </div>
  );
}
```

- `useLayoutEffect(fn, deps)`
  - same as `useEffect`, but runs callback synchronously during commit phase
- `useMemo(fn, deps)`
  - recalculates value when deps change
- `useCallback(fn, deps)`
  - updates callback function reference when deps change
- `useRef()`
  - provides a mutable “ref object” that persists between renders
  - Object is `{current: null}`
  - Used as “instance variable”, or access to DOM nodes
  - Plain object, so mutating `.current` doesn’t trigger re-renders!

Access DOM Nodes with Refs

LIVE PREVIEW

Rendered text content

Click Me

SOURCE CODE

```
function RefsExample() {
  const divRef = useRef();

  const onClick = () => {
    // refObj.current field points to the real DOM node
    if (divRef.current) {
      divRef.current.textContent = "Modified text content";
    }
  };

  return (
    <div>
      <div ref={divRef}>Rendered text content</div>

      <button onClick={onClick}>Click Me</button>
    </div>
  );
}

render(<RefsExample />);
```

- Pass a ref object as `<element ref={refObj}>`
- `refObj.current` will be the real DOM node after render is done
- Escape hatch - use sparingly!

Forms and Controlled Inputs

LIVE PREVIEW

First Name:
Last Name:
Is Jedi:

Result:

SOURCE CODE

```
function FormComponent() {
  const [nameFields, setNameFields] = useState({
    firstName: "",
    lastName: "",
  });
  const { firstName, lastName } = nameFields;

  const [isJedi, setIsJedi] = useState(false);

  const onNameFieldChanged = e => {
    setNameFields({
      ...nameFields,
      [e.target.name]: e.target.value,
    });
  };

  const onIsJediChanged = e => {
    setIsJedi(e.target.checked);
  };

  let description = "";
  if (firstName && lastName) {
    description = `(${isJedi ? "Jedi" : "Not a Jedi"})`;
  }

  return (
    <form
      style={{ display: "flex", flexDirection: "column", alignItems: "start" }}
    >
```

Forms and Controlled Inputs

- Browsers normally own and update input state based on user interaction
  - React wants to own updating DOM based on state
- React uses "controlled inputs" as the idiomatic form input technique
  - React forces inputs to have values based on state
  - Useful for validating data immediately
  - Can still use "uncontrolled" inputs if desired
- "Controlled inputs" must have:
  - `value` prop (or equivalent, like `checked` for checkboxes)
  - `onChange` callback that reads event and updates state
- Sequence:
  - React applies `value` to input
  - User edits input
  - Browser triggers change event
  - App code takes *proposed* new value from event, sets state with new value
  - React re-renders and applies `value` to input
- Tips:
  - Can use `input.name` prop -> `e.target.name` field as a generic key to update fields in an object
  - Some inputs like checkboxes have other value field, like `checked`

Styling Components

```
import React, { useState } from "react";
import classnames from "classnames";
import styles from "./StylingExample.module.css";

function StylingExample(props) {
  const [isActive, setIsActive] = useState(false);

  const contentClassname = classnames("fixed-class", {
    active: isActive,
    inactive: !isActive,
  });

  return (
    <div style={{ display: "flex", flexDirection:
"column" }}>
      <div>
        This has <span style={{ color: "red" }}>red
text</span>
      </div>
      <button className="btn btn-primary">Click
Me</button>
      <div className={contentClassname}>Some content
here</div>
      <div className={styles.moreContent}>More content
here</div>
    </div>
  );
}
```

Multiple options available for styling React components, including:

- Inline styles (`style` prop):
  - Applies styles directly to that element, high importance
  - Style objects have `camelCased` properties, unlike CSS
- Normal CSS classnames
  - Write CSS, hand-write class names
  - Common to use the `classnames()` util to conditionally combine multiple classnames into one string
  - Use `className` prop in JSX, not `class`!
- CSS Modules:
  - Build-time conversion of simple CSS file classnames to globally-unique classnames (`.label -> "StylingExample_label-a841b7"`)

Building Apps with React

React Ecosystem

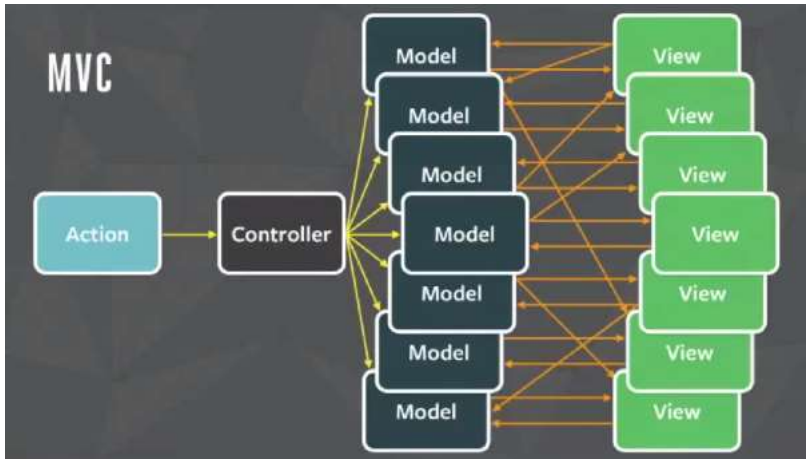
- **Focused on the "view" layer**
  - Entirely possible to build a React app with no other dependencies, but most apps use a variety of additional libraries for specific capabilities.
  - Good news: can pick exactly the libraries you need for your use cases.
  - Bad news: need to pick exactly the libraries you need for your use cases.
- **Commonly used libraries**
  - AJAX requests: Axios, SuperAgent, Fetch
  - Routing: React-Router
  - State management: Redux, MobX
  - Data/utilities: Immer, Lodash
- **Build tools**
  - Babel: compiles ES6/JSX syntax to widely-compatible ES5
  - Webpack: reads multiple JS module formats, loads non-JS formats such as CSS and images, runs Babel, and generates optimized output bundles for deployment
  - TypeScript: static type syntax on top of JS; provides compile-time type checking, and can also do JS compile output similar to Babel

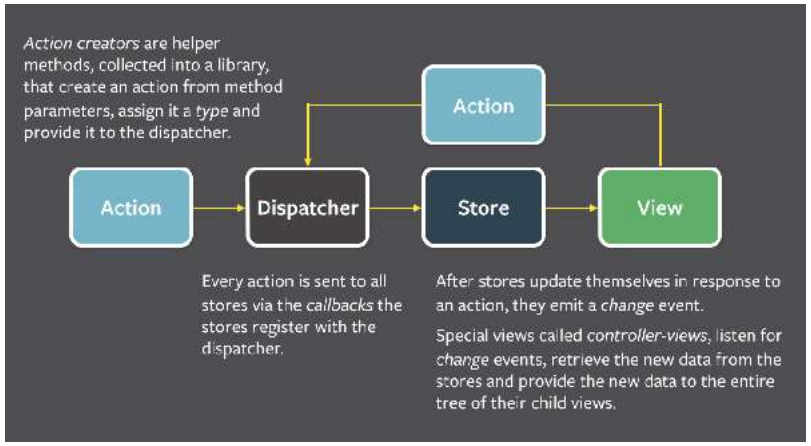
Requirements and Deployment

- **Build tools normally used, but not required**
  - You can start writing React code with two simple script tags for `react.js` and `react-dom.js` in an HTML file
  - JSX syntax is the accepted way to write React components, but not required, and other alternatives exist, although rarely used (util functions, templating tools)
  - Modern JS-based web apps do use many build tools: linter, test runner, compiler, bundler/minifier, etc. These tools are normally written in JS, and so require the Node.js runtime on your dev/build machines to run them.
- **Deployment**
  - A normal React app build is just static JS/HTML/CSS files that can be uploaded to any web server. You do not need Node on a server to run React apps.
  - React apps don't care what language the web app server is written in, as long as it provides an API the client can use to fetch data.
- **Standard React "Distributions"**
  - Create-React-App: opinionated build config for Single Page Applications
  - Next.js: React-based client/server framework, including Server-Side Rendering and Static Site Gen capabilities
  - Gatsby: focused on Static Site Gen, using GraphQL for data sources

State Management with Redux

Redux's Predecessor: Flux





- A year after releasing React, Facebook announced the “Flux Architecture”:
  - “We found that two-way data bindings led to cascading updates, where changing one object led to another object changing, which could also trigger more updates. As applications grew, these cascading updates made it very difficult to predict what would change as the result of one user interaction. When updates can only change data within a single round, the system as a whole becomes more predictable.”
  - **“Flux is more of a pattern than a framework”**
- Basic concepts:
  - Many *data stores* register with a singleton *dispatcher*. To trigger updates, plain object *actions* are dispatched, and stores update themselves in response. UI

What Is Redux?

Created by Dan Abramov for a talk at React Europe 2015 to demonstrate the idea of "time-travel debugging". Now the most widely used state management tool for React apps.

- "A predictable state container for JavaScript apps"
- "Flux taken to its logical conclusion"
- "A platform for developers to build customized state management for their use-cases, while being able to reuse things like the graphical debugger or middleware"

What Is Redux?

Predictable

Redux attempts to make state mutations predictable by imposing certain restrictions on how and when updates can happen. These restrictions are reflected in the three principles of Redux:

- **Single source of truth**: The state of your whole application is stored as a tree of plain objects and arrays within a single **store**. (How much you put in the store is up to you - not all data needs to live there.)
- **State is read-only**: State updates are caused by *dispatching* an action, which is a plain object describing what happened. The rest of the app is not allowed to modify the state tree directly.
- **Changes are made with pure functions**: All state updates are performed by pure functions called **reducers**, which are `(state, action) => newState`

Centralized

Having a single store and single state tree enables many powerful techniques: logging of all updates, API handling, undo/redo, state persistence, "time-travel debugging", error reports with full snapshots of app state, and more.

Core Concepts

```
// App state: a plain object with many keys or "slices"
const todoAppState = {
  todos: [
    { id: 0, text: "Learn React", completed: true },
    { id: 1, text: "Learn Redux", completed: false, color: "purple" },
    { id: 2, text: "Build something!", completed: false, color: "blue"
},
  ],
  filters: {
    status: "Active",
    colors: ["red", "blue"],
  },
};

// Actions: plain objects with a "type" field
const a1 = { type: "todos/todoAdded", payload: "Go to swimming pool" };
const a2 = { type: "todos/todoToggled", payload: 1 };
const a3 = { type: "filters/visibilityUpdated", payload: "SHOW_ALL" };
```

State

App state is stored in plain objects, like this Todo example. There's no "setters", so that different parts of the code can't change the state arbitrarily. That helps avoid hard-to-reproduce bugs.

Actions

To change something in the state, you need to dispatch an action. An action is a plain JS object with a `type` field. Actions are like events - they should describe "what happened in the app".

Core Concepts

```
function todosReducer(state = initialState, action) {
  switch (action.type) {
    case "todos/todoAdded": {
      return [
        ...state,
        {
          id: nextTodoId(state),
          text: action.payload,
          completed: false,
        },
      ];
    }
    case "todos/todoToggled": {
      return state.map(todo => {
        if (todo.id !== action.payload) {
          return todo;
        }

        return {
          ...todo,
          completed: !todo.completed,
        };
      });
    }
    default:
      return state;
  }
}
```

Reducers

- All state update logic lives in functions called **reducers**. Since they're just functions, smaller functions can be composed together into larger functions. Because reducer functions are `(state, action) => newState`, they are very easily testable and straightforward to understand.
- Reducers should be *pure functions*, with no "side effects". That means they should only rely on inputs, and not affect anything external. **Reducers need to update data immutably, by making copies of state and modifying the copies before returning them, rather than directly modifying inputs**.

Core Concepts

```
import { createStore, combineReducers } from "redux";

import todosReducer from "./features/todos/todosSlice";
import filtersReducer from
"./features/filters/filtersSlice";

const rootReducer = combineReducers({
  todos: todosReducer,
  filters: filtersReducer,
});

const store = createStore(rootReducer, preloadedState);
```

Store

A Redux store contains the current state value. Stores are created using the `createStore` method, which takes the root reducer function and an optional preloaded state value.

Stores have three main methods:

- `dispatch`: starts a state update with the provided action object
- `getState`: returns the current stored state value
- `subscribe`: accepts a callback function that will be run every time an action is dispatched

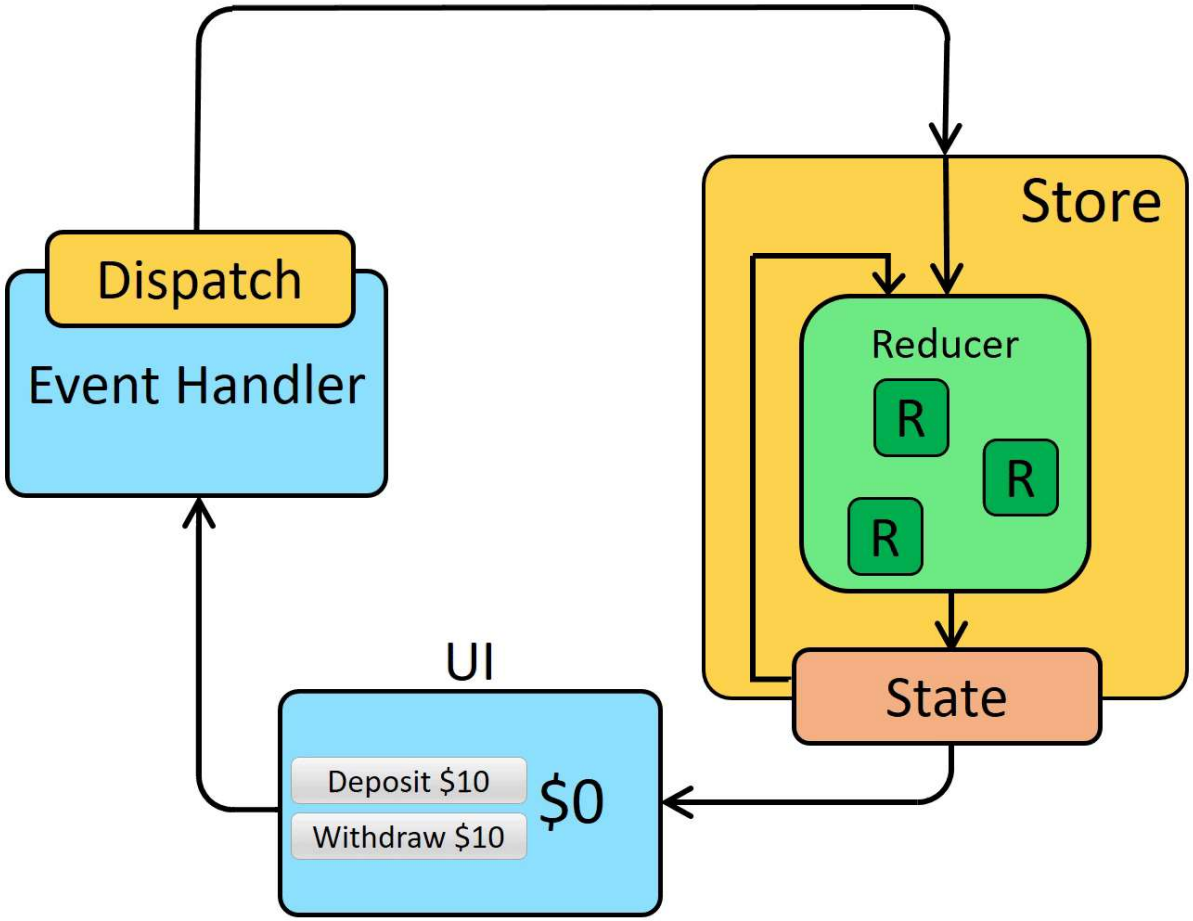
Core Concepts

```
const store = createStore(rootReducer, preloadedState);

console.log(store.getState());
// {todos : [.....], filters : {status:
"SHOW_COMPLETED"}}

store.dispatch({ type: "filters/filterUpdated", payload:
"SHOW_ALL" });
console.log(store.getState());
// {todos : [.....], filters : {status: "SHOW_ALL"}}

const stateBefore = store.getState();
console.log("Number of todos: ",
stateBefore.todos.length);
// "Number of todos: 2"

store.subscribe(() => {
  console.log("An action was dispatched");
  const stateAfter = store.getState();
  console.log("Number of todos: ",
stateAfter.todos.length);
});

store.dispatch({ type: "todos/todoAdded", payload: "Buy
milk" });
// "An action was dispatched"
// "Number of todos: 3"
```

Store

- To trigger a state update, call `store.dispatch(action)`
  - The store will call `reducer(state, action)`, and save the result
- Add subscription callbacks with `store.subscribe(listener)`
  - All subscribers will run after every dispatch

Redux Data Flow: Synchronous



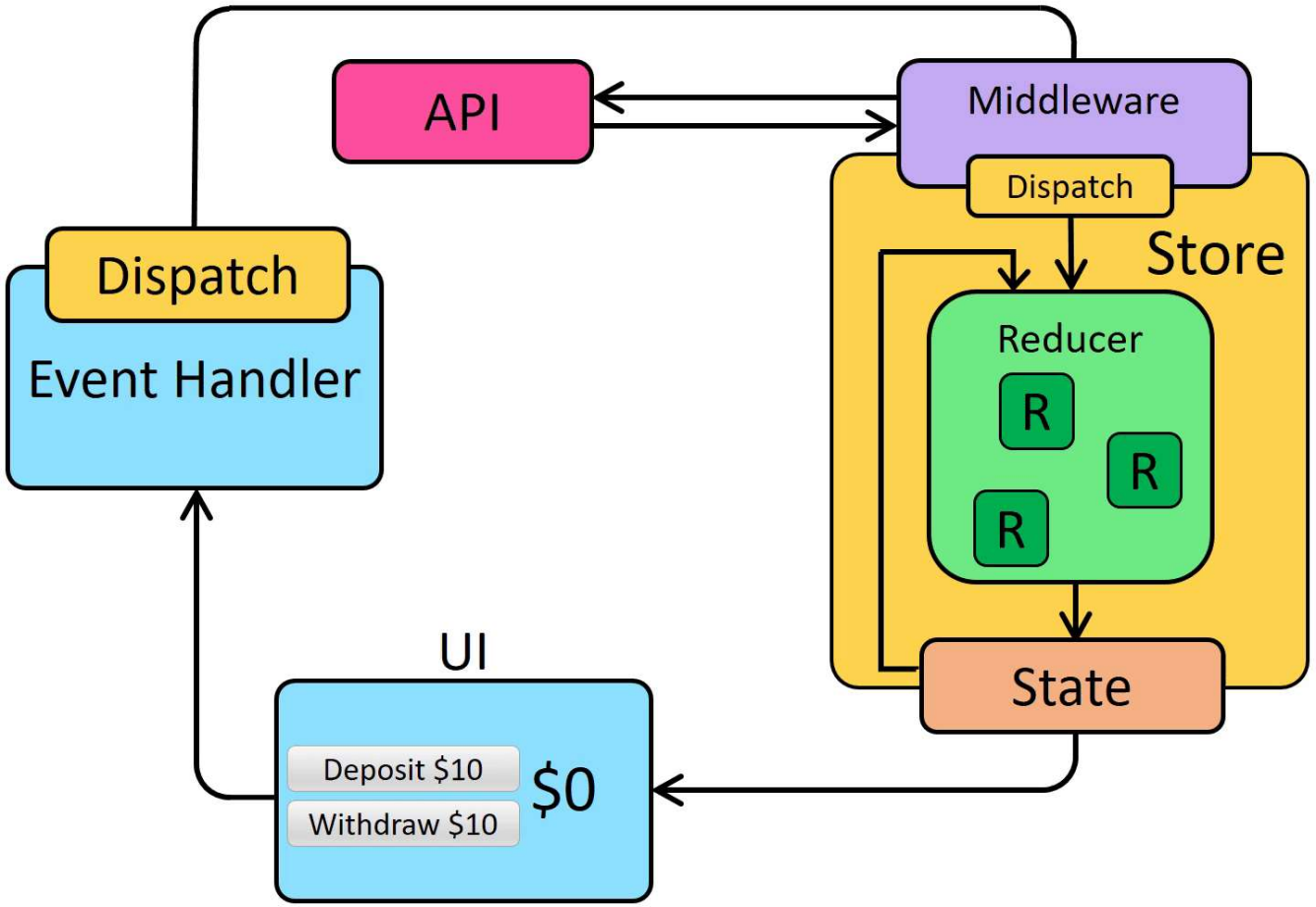
Redux and Async Logic

```
const thunkMiddleware = storeAPI => next => action => {
  // If the "action" is actually a function instead...
  if (typeof action === "function") {
    // then call the function
    return action(storeAPI.dispatch, storeAPI.getState);
  }

  // Otherwise, it's a normal action - send it onwards
  return next(action);
};

const middlewareEnhancer = applyMiddleware(thunkMiddleware);
const store = createStore(rootReducer, middlewareEnhancer);

// Write a function that has `dispatch` and `getState` as args
const fetchSomeData = (dispatch, getState) => {
  // Make an async HTTP request
  client.get("todos").then(todos => {
    // Dispatch an action with the todos we received
    dispatch({ type: "todos/todosLoaded", payload: todos });
    // Check the updated store state after dispatching
    const allTodos = getState().todos;
    console.log("Todos after loading: ", allTodos.length);
  });
};

// Pass the _function_ we wrote to `dispatch`
store.dispatch(fetchSomeData);
// logs: 'Todos after loading: ###'
```

Middleware

- Middleware are store plugins that wrap `dispatch`
- Can be used for many use cases, but primarily for async logic
- Redux “thunk” middleware is the standard async middleware
  - Allows passing *functions* to `dispatch` instead of actions
  - Functions receive (dispatch, getState)` as args
  - Can do any sync or async logic inside

Redux Data Flow: Asynchronous



Redux UI Integration

```
// 1) Create a new Redux store
const store = createStore(counterReducer);

// 2) Subscribe to redraw whenever the data changes
store.subscribe(render);

// Our "UI" is some text in a single HTML element
const valueEl = document.getElementById("value");

// 3) When the subscription callback runs:
function render() {
  // 3.1) Get the current store state
  const state = store.getState();
  // 3.2) Extract the data you want
  const newValue = state.value.toString();

  // 3.3) Update the UI with the new value
  valueEl.innerHTML = newValue;
}

// 4) Display the UI with the initial store state
render();

// 5) Dispatch actions based on UI inputs
document.getElementById("increment").addEventListener("click",
function () {
  store.dispatch({ type: "counter/incremented" });
});
```

Using Redux with a UI requires a few consistent steps:

1. Create a Redux store
2. Subscribe to updates
3. Inside the subscription callback:
   1. Get the current store state
   2. Extract the data needed by this piece of UI
   3. Update the UI with the data
4. If necessary, render the UI with initial state
5. Respond to UI inputs by dispatching Redux actions

React-Redux

React-Redux

- Since Redux is UI-framework agnostic, need a "bindings" library to integrate with a given UI layer
- React-Redux provides bindings to let React components interact with the Redux store
- Only UI bindings library maintained by the Redux team

Mark's Dev Blog:
The History and Implementation of React-Redux

- Integrating Redux with a UI
  - What does React-Redux even do for you, and why do we need "UI bindings" in the first place?
  - How does it work internally?
- Development History of the React-Redux API
  - Initial design constraints
  - API and implementation changes over time

React-Redux v7

- Goals:
  - v6 architecture tried to use context for state propagation, which had performance problems
  - Switch back to per-component Redux store subscriptions for better performance
  - Enable future implementation of a React-Redux hooks API
- Behavior and implementation:
  - Rewrote `connect` using React hooks API + internal Redux store subscriptions
  - Used React renderer batching from ReactDOM and React Native
  - Released in spring 2019

React-Redux v7.1: Hooks!

- Goal: eventually design and ship a public `useRedux()`-type hooks API
  - Bikeshedding issue thread: Issue #1179: Discussion: Potential Hooks API Design
  - Extensive discussions around potential API designs and implementation constraints
- Behavior and implementation:
  - Settled on two primary hooks: `useSelector` and `useDispatch`
  - Unlike `connect`, cannot enforce top-down nested subscriptions
  - Reference (`===`) equality checks for selectors instead of shallow equality
  - No action creator binding
  - No automatic wrapping of components to avoid re-renders
- **We now recommend using the hooks API as the default approach**
  - `connect` is supported indefinitely, but no plans for further changes
- The hooks API has a different set of tradeoffs than `connect` does:
  - Thoughts on React Hooks, Redux, and Separation of Concerns
  - ReactBoston 2019: Hooks, HOCs, and Tradeoffs

React-Redux: `useSelector()`

```
import React from 'react'
import { useSelector } from 'react-redux'

export const PostsList = () => {
  const posts = useSelector(state => state.posts)

  const renderedPosts = posts.map(post => (
    <article className="post-excerpt" key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content.substring(0, 100)}</p>
    </article>
  ))

  return (
    <section>
      <h2>Posts</h2>
      {renderedPosts}
    </section>
  )
}
```

- Extracts a value from the Redux state for use in this component
  - Accepts a “selector” function as its argument
  - Subscribes to the store and re-runs the selector whenever the store state changes
- Differences from `connect` and `mapState`:
  - Uses reference equality by default (also takes an optional equality comparison argument)
  - Can be called multiple times in one component
  - Don’t have to return an object (and should prefer returning single values)

React-Redux: `useDispatch()`

```
import React, { useState } from 'react'
import { useDispatch } from 'react-redux'
import { postAdded } from './postsSlice'

export const AddPostForm = () => {
  const [title, setTitle] = useState('')
  const [content, setContent] = useState('')

  const dispatch = useDispatch()

  const onSavePostClicked = () => {
    dispatch( postAdded({title, content }) )
  }

  return (
    <section>
      <h2>Add a New Post</h2>
      <form>
        {/* omit form inputs */}
        <button type="button" onClick=
{onSavePostClicked}>
          Save Post
        </button>
      </form>
    </section>
  )
}
```

- Returns the store's `dispatch` method
- Differences from `connect` and `mapDispatch`:
  - Equivalent to calling `connect` with no `mapDispatch` argument
  - No "binding action creators" any more - up to you to call `dispatch()` in your own handlers

React-Redux: `<Provider>`

```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";

import App from "./App";
import store from "./store";

ReactDOM.render(
  // Render a `<Provider>` around the entire `<App>`,
  // and pass the Redux store to as a prop
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

- Makes the Redux store accessible to all components in the app
- Should be set up in app entry point file, and wrap entire app component

Redux Toolkit

Redux Toolkit

- Common complaints about Redux:
  - Configuring a Redux store is too complicated
  - Have to add a lot of packages to do anything useful (`redux-thunk`, `reselect`, etc)
  - Too easy to accidentally make a mistake like mutating state
  - Amount of "boilerplate" you need to write (see Issue #2295 for discussion)
    - Action types
    - Action creators
    - Immutable update logic
    - "Have" to use multiple files
    - Complex store setup process

Redux Toolkit

- Created a new official package called **Redux Toolkit**, inspired by `create-react-app` and `apollo-boost`.
- Goals:
  - Simplify common Redux use cases
  - Provide good opinionated defaults out of the box
  - Minimize the amount of code you have to write by hand
  - Doesn't "hide" that you're using Redux, just makes it easier
  - Opt-in - can add incrementally to an existing app, or use day 1 on a new project
  - Provide a great developer experience for TypeScript users
- Originally named "Redux Starter Kit", but renamed after 1.0 release
  - Confusion over "starter kit" naming: boilerplate? only good for beginners?
  - Renamed to "Redux Toolkit" (package: `@reduxjs/toolkit`) for 1.0.4 release
- Written in TypeScript, designed to simplify TS usage patterns
- Most recent TS and API dev work thanks to co-maintainer Lenz Weber (`@phryneas`)

Useful for *all* Redux users, both new and experienced!

Redux Toolkit: `configureStore()`

```
import { configureStore } from "@reduxjs/toolkit";

import todosReducer from "./todos/todosReducer";
import visibilityReducer from
"./visibility/visibilityReducer";

const store = configureStore({
  reducer: {
    todos: todosReducer,
    visibility: visibilityReducer
  }
});

/*
The store has been created with these options:
- The slice reducers automatically passed to
combineReducers()
- Added redux-thunk and mutation detection middleware
- DevTools Extension is enabled (w/ "action stack
traces")
- Middleware and devtools enhancers were composed
*/
```

- A small wraper around the Redux `createStore` function:
  - Automatically sets up the Redux DevTools extension by default
  - Automatically adds `redux-thunk` by default, plus middleware to check for accidental mutations and non-serializable values
  - Accepts either a root reducer function, or an object of slice reducers and will automatically call `combineReducers` for you
  - Accepts middlewares and store enhancers as arrays, and composes them properly

Redux Toolkit: `configureStore()`

```
import {
  configureStore,
  getDefaultMiddleware,
  combineReducers
} from "@reduxjs/toolkit";

// Examples of adding a middleware and a store enhancer
import logger from "redux-logger";
import { reduxBatch } from "@manaflair/redux-batch";

import todosReducer from "./todos/todosReducer";
import visibilityReducer from
"./visibility/visibilityReducer";

const rootReducer = combineReducers({
  todos: todosReducer,
  visibility: visibilityReducer
});

const store = configureStore({
  reducer: rootReducer,
  middleware: [...getDefaultMiddleware(), logger],
  devTools: NODE_ENV !== "production",
  preloadedState: {},
  enhancers: [reduxBatch]
});
```

- Options are passed as a "named arguments" object:
  - Override middleware list
  - Use an initial state
  - Enable/disable DevTools Extension
  - Provide additional store enhancers

Immer

```
import produce from "immer";

// Plain JS with object spread and map
return {
  ...state,
  first: {
    ...state.first,
    second: state.first.second.map((item, i) => {
      if (i !== index) return item;

      return {
        ...item,
        value: 123
      };
    })
  }
};

// Immer
return produce(state, draft => {
  // "Mutating" the draft here is safe - it's a Proxy
wrapper!
  draft.first.second[index].value = 123;
});
```

- An immutable update library from Michel Westrate (author of MobX)
  - Uses ES6 Proxies to let you “mutate” your data, but applies the changes immutably
  - Can drastically simplify your immutable update logic

Redux Toolkit: `createReducer()`

```
import { createReducer } from "@reduxjs/toolkit";

function addTodo(state, action) {
  // Can safely call state.push() here
  state.push({ text: action.payload, completed: false
});
}

function toggleTodo(state, action) {
  const { index } = action.payload;

  const todo = state[index];
  // Can directly modify the todo object
  todo.completed = !todo.completed;
}

const todosReducer = createReducer([], {
  ADD_TODO: addTodo,
  TOGGLE_TODO: toggleTodo
});
```

- Accepts a lookup table of action types to reducer functions
- It uses Immer internally, so your reducers can “mutate” the state!
- **Warning:** this only works _if_ you are using the “magic” `createReducer` with Immer inside. Otherwise, these functions `are` mutating the state!

Redux Toolkit: `createAction()`

```
import { createAction, createReducer } from
"@reduxjs/toolkit";

const addTodo = createAction("ADD_TODO");

console.log(addTodo("Buy milk"));
// {type : "ADD_TODO", payload : "Buy milk"}

console.log(addTodo.toString());
// "ADD_TODO"

console.log(addTodo.type);
// "ADD_TODO"

const todoReducer = createReducer([], {
  // Use the action creator function as the object key!
  // ES6 computed properties will coerce to string
  [addTodo]: (state, action) => {
    state.push({ text: action.payload, completed: false
});
  }
});
```

- Inspired by `redux-actions`
- Generates an action creator that uses the given type and accepts `payload` as an argument
- Action creator function overrides `toString()`, so it can be used as the “action type” itself where needed (also exposed as `actionCreator.type`)

Redux Toolkit: `createSlice()`

```
import { createSlice } from "@reduxjs/toolkit";

const userSlice = createSlice({
  name: "user",
  initialState: { name: "", age: 20 },
  reducers: {
    // mutate the state all you want with immer
    updateUser: (state, { payload }) => {
      state[payload.field] = payload.value;
    }
  }
});

export const { updateUser } = userSlice.actions;
export default userSlice.reducer; // "Ducks" - quack!

// Use this elsewhere in the app:
import userReducer, { updateUser } from "./userSlice";

const rootReducer = combineReducers({
  user: userReducer
});

store.dispatch(updateUser({ field: "name", value: "Eric"
}));
```

- Inspired by Eric Elliott's `autodux` project
- Accepts an object of reducers, and returns auto-generated action creators, action types, and a reducer function
- Uses our "magic" `createReducer` utility, so that your reducers can "mutate" their state
- Makes it easy to use the "ducks" pattern (default export reducer, named export action creators)

Redux Toolkit: `createSlice()`

```
import { createSlice } from "@reduxjs/toolkit";
import { increment } from "./counterSlice";

const userSlice = createSlice({
  name: "user",
  initialState: { name: "Fred", age: 20, pets: [] },
  reducers: {
    updateUser: (state, { payload }) => {
      state[payload.field] = payload.value;
    }
  },
  extraReducers: {
    // Handle other action types here
    [increment]: (state, action) => {
      state.age++;
    }
  }
});

export const { updateUser } = userSlice.actions;
export default userSlice.reducer; // "Ducks" - quack!

// Write side effects logic alongside, like thunks:
export const getUserPets = name => async dispatch => {
  const userPets = await fetchPets(name);
  dispatch(updateUser({ pets: userPets }));
  dispatch(increment());
};
```

- Usage patterns:
  - Can handle other action types using `extraReducers`
  - Slices are just actions + reducers - write side effects like thunks separately and use the generated action creators, same as usual

Redux Toolkit: `createSelector()`

```
import { createSelector } from "@reduxjs/toolkit";

const selectTodos = state => state.todos;
const selectStatusFilter = state =>
state.filters.status;

const selectFilteredTodos = createSelector(
  [selectTodos, selectStatusFilter],
  (todos, filter) => {
    // Only recalculates output result when inputs
changed
    return todos.filter(t => t.status === filter);
  }
);
```

- Re-exports the `createSelector` function from Reselect, for creating memoized selectors

Redux Toolkit: `createAsyncThunk()`

```
import { createAsyncThunk, createSlice } from
"@reduxjs/toolkit";
import { userAPI } from "./userAPI";

const fetchUserById = createAsyncThunk(
  "users/fetchByIdStatus",
  async (userId, thunkAPI) => {
    const response = await userAPI.fetchById(userId);
    return response.data;
  }
);

const usersSlice = createSlice({
  name: "users",
  initialState: { entities: [], loading: "idle" },
  reducers: {
    // standard reducer logic, with auto-generated
actions
  },
  extraReducers: {
    // Add reducers for additional action types here
    [fetchUserById.fulfilled]: (state, action) => {
      state.entities.push(action.payload);
    }
  }
});

// Later, dispatch the thunk as needed in the app
dispatch(fetchUserById(123));
```

- Standard async thunk pattern:
  - Dispatch “loading” action before fetch
  - Dispatch either “success” or “failure” action based on result
- `createAsyncThunk` implements that pattern:
  - Accepts an action type string prefix and a “payload creator” callback that returns a Promise
  - Autogenerates actions for `pending`, `fulfilled`, and `rejected` cases
  - Auto dispatches those actions based on promise resolution lifecycle

Redux Toolkit: `createEntityAdapter()`

```
import { createEntityAdapter, createSlice } from
"@reduxjs/toolkit";

const booksAdapter = createEntityAdapter({
  // Assume IDs are stored in a field other than
`book.id`
  selectId: book => book.bookId,
  // Keep the "all IDs" array sorted based on book
titles
  sortComparer: (a, b) => a.title.localeCompare(b.title)
});

const booksSlice = createSlice({
  name: "books",
  initialState: booksAdapter.getInitialState(),
  reducers: {
    // Can pass adapter functions directly as case
reducers
    bookAdded: booksAdapter.addOne,
    booksReceived(state, action) {
      // Or, call them as "mutating" helpers in a case
reducer
      booksAdapter.setAll(state, action.payload.books);
    }
  }
});

// Can create a set of memoized selectors
const booksSelectors = booksAdapter.getSelectors(state
=> state.books);
const allBooks =
booksSelectors.selectAll(store.getState());
```

- Redux docs recommend "normalizing" state structure
  - Single copy of each item
  - Track items in an object mapping IDs to items
- `createEntityAdapter` implements that pattern:
  - Defines prebuilt reducers for common CRUD logic (add / update / delete, one/many)
  - Reducers can be used directly or as "mutating" helpers
  - Maintains IDs array, with optional sorting based on items
  - Generates basic set of selector functions (select all as array, one by ID, etc)

Redux Toolkit: Safety Checks

- Most common mistake with Redux: accidental mutations!
  - Can happen in a reducer, *or* outside a reducer (`mapState`, etc)
  - Usually results in connected components not re-rendering, or bad data when debugging
  - *Really* hard to figure out where this happened... or is it?

Redux Toolkit: Safety Checks

- Most common mistake with Redux: accidental mutations!
  - Can happen in a reducer, *or* outside a reducer (`mapState`, etc)
  - Usually results in connected components not re-rendering, or bad data when debugging
  - *Really* hard to figure out where this happened... or is it?
- Several existing "mutation detection" middleware available - the best one is `redux-immutable-state-invariant`
- Redux Toolkit's `configureStore()` adds a port of `redux-immutable-state-invariant` by default!
  - Throws an error when accidental mutations are detected
  - Tells you the state path where the mutation occurred
- RTK also adds a check for "non-serializable values" in state and actions by default
  - Created by me, modeled on the mutation check middleware

Redux Toolkit: Roadmap

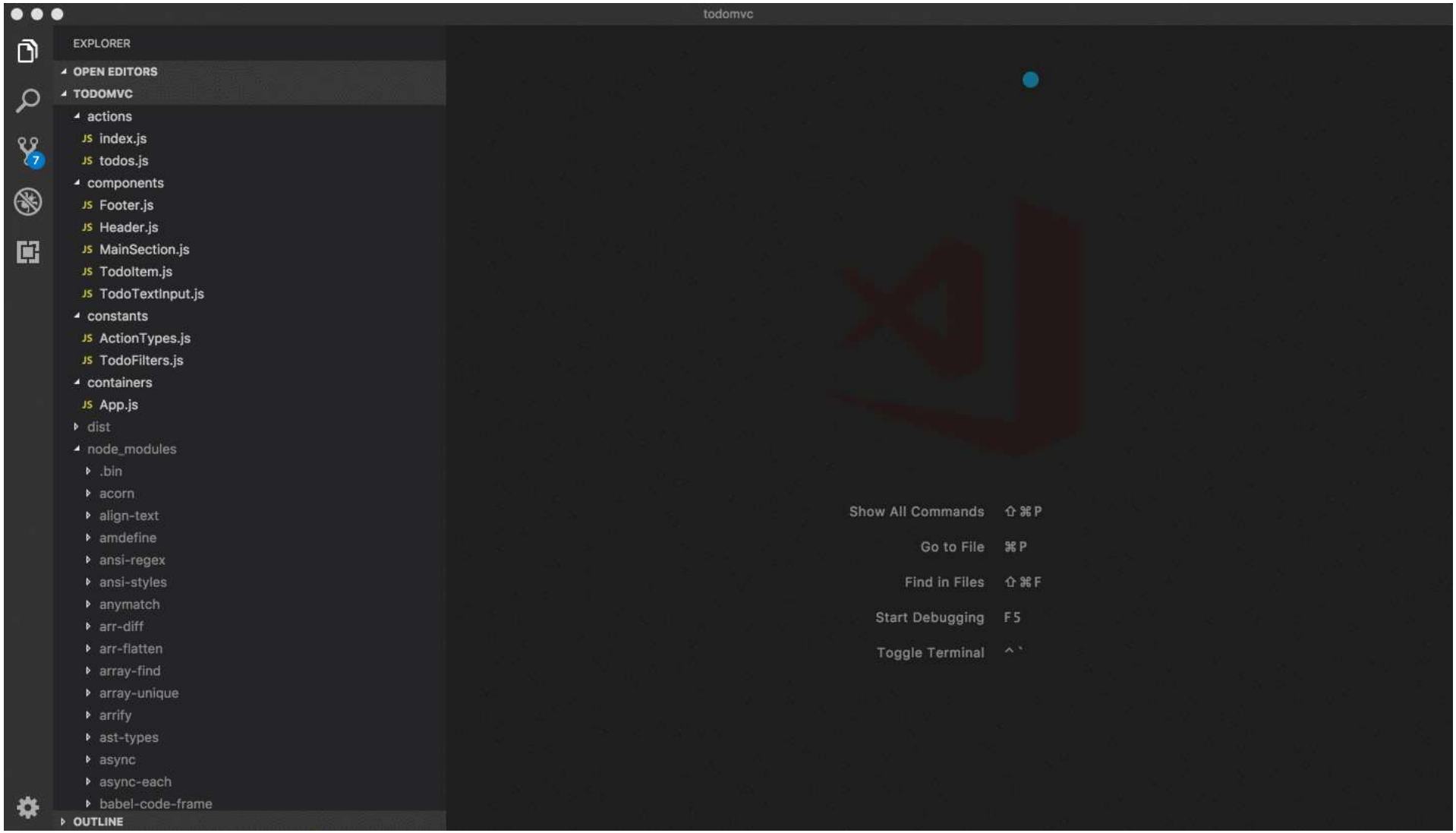
```
// features/pokemon/pokemonService.ts
import { createApi, fetchBaseQuery } from '@rtk-incubator/rtk-query';

// Define a service using a base URL and expected endpoints
export const pokemonApi = createApi({
  reducerPath: 'pokemonApi',
  baseQuery: fetchBaseQuery({ baseUrl: 'https://pokeapi.co/api/v2/' }),
  endpoints: (builder) => ({
    getPokemonByName: builder.query({
      query: (name: string) => `pokemon/${name}`,
    }),
  }),
});

// Export hooks for usage in functional components, which are
// auto-generated based on the defined endpoints
export const { useGetPokemonByNameQuery } = pokemonApi;

// in a component:
function PokemonEntry() {
  // Using a query hook automatically fetches data and returns query
values
  const { data, error, isLoading } =
useGetPokemonByNameQuery('bulbasaur');
}
```

- Current RTK version: 1.5.0
- New experimental API now available: **RTK Query**
  - Advanced data fetching and caching library built for Redux Toolkit
  - Just released as alpha - will iterate on API, then merge into RTK

Redux DevTools Extension

- History:
  - Original DevTools logic and component created by Dan Abramov
  - Turned into a browser extension by Mihail Diordiev and maintained separately
- New "action stack traces" feature:
  - Allows viewing the complete stack trace for any dispatched action - no more digging through files to see where the dispatch actually occurred!
  - Reuses the stack trace components from `create-react-app`, including showing original source if available
  - Clicking source lines will open file in DevTools debugger or your preferred editor
  - Prototyped by me, polished and merged by Mihail Diordiev
  - *Not* enabled by default - need to call `composeWithDevTools({trace : true})`
  - RTK's `configureStore` does turn this on automatically

Action Stack Traces



TypeScript

TypeScript

- Sales pitch:
  - "JavaScript, with static types on top"
  - "All JavaScript is valid TypeScript" 😂😜
- Overview:
  - Standalone compiler that compiles TS syntax to various flavors of ES, with compile-time type-checking
    - Can also be parsed by Babel, but without type-checking
  - Language aimed to be a superset of current JS (ES20xx, + stage 3 features)
  - Created by Microsoft (author of C#)
  - Open-source, but developed by an Microsoft-led team
  - Rapid adoption over last 5 years (some surveys indicate >40% industry usage in newer apps)

Why Use TypeScript?

- **Documentation**: static types tell devs what variables look like quickly - especially valuable when working with unfamiliar code
- **Compile-time errors**: common issues like typos or undefined values can be caught immediately, rather than at runtime; compiler prevents passing invalid values
- **Intellisense**: type declarations allow IDEs to provide proper autocompletion and type information when writing code
- **Refactoring**: can confidently rename / delete / extract code, rather than searching and "hope I found all the uses"
- **Long-term maintainability**: better codebase information for future developers who may rotate on and off the project
- **Code Quality**: doesn't replace unit tests, but can help minimize errors
- **Library Support**: most common 3rd-party libs either ship typings, or community has created their own

Downsides of Using TypeScript

- **Learning Curve**: additional syntax and concepts take time to understand, *on top of* knowing plain JS by itself
- **Time to Write Code**: literally more code to write out than just plain JS
- **Difficulties Typing Dynamic JS**: can be difficult to come up with good static types for highly dynamic JS behavior
- **Inconsistent/Missing Library Types**: not all libs have typings, and quality can vary
- **Compilation Time**: TS usage can slow down build times
- **Over-Emphasis on Type Coverage**: some TS users spend too much time trying to achieve "100% perfect static type coverage" of an entire codebase, leading to bizarrely complex types

TypeScript: Basic Syntax

```ts
// This TypeScript code with a type declaration:
const greeting = (person: string) => {
  console.log("Good day " + person);
};
greeting("Daniel");

// Compiles to this plain JS code:
var greeting = function (person) {
  console.log("Good day " + person);
};
greeting("Daniel");

// Basic TS/JS types:
let isAwesome: boolean = true;
let name: string = "Mark";
let meaningOfLife: number = 42;

// Arrays:
let letters: string[] = ["a", "b", "c"];

// "Tuples": fixed-size arrays with specific types
let tuple1: [string, number, boolean];
// ✅
tuple1 = ["chair", 20, true];
// ❌ - Should be a string, not a number
tuple1 = [5, 20, true];
```

```ts
// `any`: the "cheat code" for the type system.
let num1: number = 4;
num1 = "test"; // ❌ERROR: cannot assign a string to a number

let whoKnows: any = 4; // assigned a number
whoKnows = "a beautiful string"; // can be reassigned to a string
whoKnows = false; // can be reassigned to a boolean, or whatever

// `void`: equivalent to `undefined`
let logAndReturnNothing = (): void => {
  console.log("hey there");
};

// TS can often "infer" types based on values:
let x = 10; // x is given the number type

const tweetLength = (message = "A default tweet") => {
  // inferred string type, so this is okay
  return message.length;
};
```

TypeScript: Basic Syntax

```
enum Sizes {
  Small,
  Medium,
  Large,
}
Sizes.Medium; // => 1 // default to 0-indexed

// Enums can also be string-based
enum ThemeColors {
  Primary = "primary",
  Secondary = "secondary",
}

// "Interfaces": declaring the "shape" of an object
interface User {
  username: string;
  age: number;
  friends: User[];
}

// You can "union" values that may be more than one
type:
let x: number | string = 42;
x = "test"; // ✅
x = false; // ❌ - not a number or string

// `type` keyword allows declaring many type
combinations,
type NumberOrString = number | string;

// "Generics" are similar to Java, C#, and C++: declare
// a placeholder type value, which will be made
concrete.
const fillArray = <T>(len: number, elem: T) => {
  return new Array() < T > len.fill(elem);
};
const newArray = fillArray < string > (3, "hi"); // =>
['hi', 'hi', 'hi']
newArray.push("bye"); // ✅
newArray.push(true); // ❌ - only strings can be added
to the array

// "Intersection" types are combined together:
type Student = { id: string, age: number };
type Employee = { companyId: string };
let person: Student & Employee;
person.age = 21; // ✅
person.companyId = "SP302334"; // ✅
person.id = "10033402"; // ✅
person.name = "Henry"; // ❌ - not in Student & Employee

// Optional types are indicated with a `?`:
interface Person {
  name: string;
  age: number;
  favoriteColor?: string; // This property is optional
}
```

TypeScript: Basic Syntax

```ts
// Function declarations:
function addTwoNumbers(a: number, b: number): number {
  return a + b;
}

// Still use `?` for optional values
// TS can probably "infer" that the result is a number
function addTwoOrThreeNumbers(a: number, b: number, c?:
number) {
  let result = a + b;

  // Must check to see if `c` is defined before using!
  if (c !== undefined) {
    result += c;
  }

  return result;
}

function addManyNumbers(a: number, ...others: number[])
{
  let finalResult = others.reduce((sum, current) => {
    return sum + current;
  }, a);
}
```

```ts
// You can use "type guards" to ensure that a value is
// the correct type in a particular code branch
function lower(x: string | string[]) {
  if (typeof x === "string") {
    // x is guaranteed to be a string
    return x.toLowerCase();
  } else {
    // definitely a string[], so we can use reduce
    return x.reduce((val: string, next: string) => {
      return (val += `, ${next.toLowerCase()}`);
    }, "");
  }
}

function clearElement(element: string | HTMLElement) {
  if (element instanceof HTMLElement) {
    // element is guaranteed to be an HTMLElement in
here
    // so we can access its innerHTML property
    element.innerHTML = "";
  } else {
    // element is a string in here
    const el = document.querySelector(element);
    if (el !== null) {
      el.innerHTML = "";
    }
  }
}
```

TypeScript Usage Tips

- Goal: **"80% sweet spot" of type coverage!**
  - See "Lessons and Takeaways" section in Mark's post **Learning and Using TS as an App Dev and Lib Maintainer**
  - Want correct types, but not wasted effort
- Aim to type: function arguments / return values, React component props and state, API responses, Redux state
- Infer values as much as possible - don't insist on declaring types for *every* variable!
  - Sometimes may want to declare function return types instead of inferring, just to be sure they're correct or catch errors
- Avoid use of `any` and `// @ts-ignore` ... *unless* absolutely necessary
  - valid escape hatches, but they're a last resort
- On the other hand, prefer using these escape hatches instead of wasting hours fighting TS compiler
  - You know what the code does - sometimes hard to convince the compiler you're right
  - Can use a type like `type $FixTypeLater = any` as a placeholder

Typing React Components

```tsx
interface MyComponentProps {
  a: string;
  b: number;
  c: SomeObject[];
}

function MyComponent({
  a,
  b,
  c
} : MyComponentProps) { // declare type of props
  // Basic state values can be inferred
  const [counter, setCounter] = useState(0);
  // declare complex state types via generics
  const [todos, setTodos] = useState<Todo[] | null>(null);

  const onChange = (e: React.ChangeEvent<HTMLInputElement>) => {

  }

  let content: React.ReactNode;

  if (counter > 0) {
    content = <div>Counter {counter}</div>
  }

}
```

- Read and follow the React TypeScript Cheatsheet!
- Declare interfaces for React props
- Do *not* use `React.FC`!
- Infer or declare types for `useState`
- Declare types for event handlers if declaring separately
- `React.ReactNode`: JSX element, string, number, null
- `React.CSSProperties`: type for `style` prop
- Gotcha: TS doesn't like React function components that only return an array

Typing Redux

```ts
interface Todo {
  text: string;
  completed: boolean;
}

interface TodosState {
  todos: Todo[];
  status: "idle" | "loading";
}

// Declare type of initial state
const initialState: TodosState = {
  todos: [],
  status: "idle",
};

const todosSlice = createSlice({
  name: "todos",
  initialState,
  reducers: {
    // Declare type of actions in createSlice
    todoAdded(state, action: PayloadAction<Todo>) {
      state.push(action.payload);
    },
    todoToggled(state, action: PayloadAction<number>) {
      const todo = state[action.payload];
      todo.completed = !todo.completed;
    },
  },
});
```

- Read and follow the Redux Toolkit “Usage with TS” docs
- Declare data types in slice files
- Declare types of initial state for slices
- Declare type of actions in reducers as `PayloadAction<TypeOfPayloadHere>`

Typing React-Redux

```
import { RootState, AppDispatch } from "app/store";

function TodosList() {
  const todos = useSelector((state: RootState) => state.todos);
  const dispatch: AppDispatch = useDispatch();
}
```

- Declare `(state: RootState)` as arg in selector functions
- May need to declare `const dispatch : AppDispatch = useDispatch()` in components to be able to dispatch thunks properly
  - Note: Can define pre-typed versions of both of these hooks, then import those pre-typed hooks everywhere - see React-Redux “Usage with TS” docs page

TypeScript: Using Libraries

- Libraries may include an `index.d.ts` file containing type declarations
  - Allows TS compiler to know what types the library exports
  - Could be written by hand, or generated by TS compiler
- Many libraries ship their own typings
- Community maintains 3rd-party library typing file declarations in the DefinitelyTyped repo
- 3rd-party library typings can be installed from the `@types` namespace on NPM:
  - `npm i react @types/react`
- If a library has no types, or the types are inaccurate, you can create your own custom typings file for that library:
  - Could just mock it out with `any`
  - Could try to declare sort-of accurate types

TypeScript: Overriding Library Types

Post: Adding Custom Type Definitions to a Third-Party Library

Config Changes

- Edit your `tsconfig.json`, add a `compilerOptions` > `typeRoots` section, add `./types` to the search path, and exclude from compile:

```
{
    "compilerOptions": {
        ...
        "typeRoots": [ "./types", "./node_modules/@types"]
    },
    "exclude": ["node_modules", "types", ...]
}
```

Declare Module Types

- Create a folder and type file for that lib:
  - `types/third-party-library-name/index.d.ts`
- "Declare" that module to make it `any`:
  - `declare module 'third-party-library-name'`
- You can then declare real types for the lib if desired
- Alternately, you may be able to create a `$PROJECT/global.d.ts` file, and put the module declarations there

Further Information

Learning Resources: Web Fundamentals

How Web Apps Work series

Overviews of key web dev terms, technologies, and concepts:

- HTTP and Servers
- Client Development and Deployment
- Browsers, HTML, and CSS
- JavaScript and the DOM
- AJAX, APIs, and Data Transfer

JavaScript for Java Devs

A cheatsheet to modern JS syntax and concepts, along with key tools in the JS ecosystem:

- JavaScript for Java Devs

Learning Resources: React

React Docs

- Getting Started (docs overview and related resources)
- Main Concepts (read the whole series, but especially these two):
  - Lifting State Up
  - Thinking In React
  - React Hooks guide (lays out the motivation, teaches hooks, API reference, in-depth FAQ)

The React docs still teach classes in the tutorials. A rewrite is in progress (first beta due early 2021), but until then, there's a "React with Hooks" version of the React docs that uses hooks and function components for all examples:

- "React with Hooks" docs port

Learning Resources: React

React Tutorials

- Tania Rascia: Getting Started with React
- Kent C Dodds: React Tutorial for Beginners (videos)
- Scrimba: Learn React (interactive Tutorials)

Creating React Projects

- CodeSandbox.io (an online IDE that uses VS Code's editor, and can let you develop and run your apps completely in the browser)
- Create-React-App (the official CLI tool for creating a React app with one command. Sets up a project with good default build settings out of the box.)

Learning Resources: React

React Concepts and Topics

- Dan Abramov: React as a UI Runtime (deep dive - not *required* reading, but will definitely help you understand React better)
- Dan Abramov: A Complete Guide to useEffect (very long article, but a must-read. Teaches how hooks use closures, defining when effects run, and much more.)
- Mark Erikson: A (Mostly) Complete Guide to React Rendering Behavior (detailed walkthrough of many critical aspects of how rendering works)
- Dave Ceddia: Immutability in React and Redux: The Complete Guide
- Gosha Arinich: Controlled and uncontrolled form inputs in React

Learning Resources: React

Additional React Resources

- Mark Erikson's React-Redux links collection (many categories of links to articles)
- Mark Erikson's blog
- Dave Ceddia's blog
- Robin Wieruch's blog
- Kent C Dodds' blog

Learning Resources: Redux

Redux Core Tutorials

- "Redux Essentials" tutorial: explains "how to use Redux, the right way", using the latest recommended techniques and practices like Redux Toolkit and the React-Redux API, while building a real-world-ish example app.
- "Redux Fundamentals" tutorial: teaches "how Redux works, from the ground up". including core Redux data flow and why standard Redux patterns exist.

Additional Tutorials

- Dave Ceddia: A Complete React-Redux Tutorial
- Dan Abramov's Redux video tutorials:
  - Getting Started with Redux
  - Building React Apps with Idiomatic Redux

Learning Resources: Redux

Redux Concepts and Topics

- Mark Erikson: The Tao of Redux, Part 1: Implementation and Intent
- Mark Erikson: The History and Implementation of React-Redux
- Mark Erikson: Using Reselect Selectors

Learning Resources: TypeScript

TypeScript Tutorials

- Get Started with TypeScript in 2019
- TypeScript Deep Dive (a free complete online book that's considered the best available resource on TypeScript)
- The Definitive TypeScript Guide
- The TypeScript Guide

React/Redux + TypeScript

- The React + TypeScript Cheatsheet
- Redux Toolkit: Usage with TypeScript
- React-Redux: Usage with TypeScript