HW: Complete Intro React

The Complete Introduction to React
All the fundamental React.js concepts in one place

You can count on this guide to always be recent. We update it after every major change in React. It was last updated after the 17.0 release.

• This is a beginner-friendly guide that covers the concepts I classify as **fundamentals** for working with React.
• It is not a complete guide to React but rather a complete introduction.
• At the end of this guide, I list a few next-level resources for you.
• This guide will pave the way for you to understand them.

React Definition

• React is a JavaScript library for building User interfaces
  ◇ Two parts of this in depth:
    1- **React is a JavaScript "library"**. Not a "framework".
    2- **React is for building User Interfaces**. It does this and does it well.

React is a library

• **React is a JavaScript library**, not a framework.
  ◇ Not a complete soluion
  ◇ Need to use more libraries with React to form a solution

• **Frameworks serve great purpose**

◇ Especially for young teams and startups:
■ Many smart decisions are made for you
■ Allows you to focus on writing good applications-level logic

- **Frameworks come with disadvanges**
◇ Not as friendly to experienced developers with large codebases:
■ Not flexible
■ Fights against you to code a certain way
■ Usually very bloated

React builds UIs

- **React is for building UIs**
◇ Follows the Unix philosophy
■ *“Write programs that do one thing and do it well. Write programs to work together. ”* - Doug McIlroy
◇ React is small
◇ UIs (user interfaces) include anything the users interact with
■ We use React for building web UIs

- **React is declarative**
◇ We describe the UI to React, which then builds it
■ React takes care of the “how”
◇ React shares declarative power with HTML UIs
■ with static AND dynamic data

- **React is performance friendly**
◇ Introduced the idea of the Virtual DOM
■ DOM meaning “Document Object Model”
- browser's programming interface for HTML and XML documents
- tree structure
- API used to change document structure, style, content
◇ Created common language between devs and browsers

- allows devs to declaratively describe UIs
- allows devs to manage actions on UI state
  - as opposed to actions of DOM elements
- The language of outcomes
  - devs describe the UI in terms of “final” state
  - takes care of UI actions to state based on DOM changes

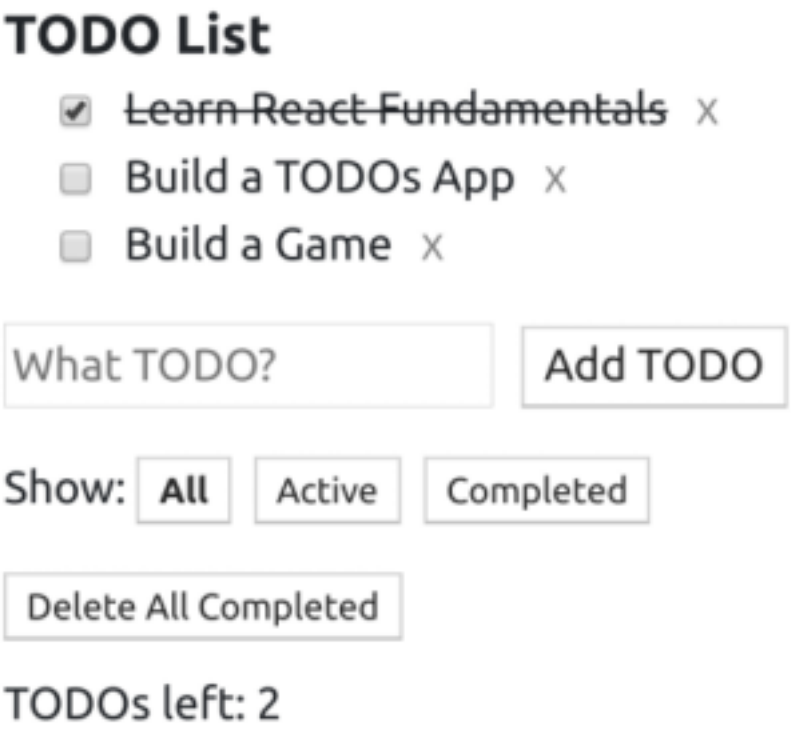
If someone asked you to give **one** reason why React is worth learning, this outcomes-based UI language is it. I call this language "the React language".

The React language

- We have a to-do list that need a UI

```
const todos: [
  { body: 'Learn React Fundamentals', done: true },
  { body: 'Build a TODOs App', done: false },
  { body: 'Build a Game', done: false },
];
```

- This array is the starting state of your UI
- You need to build a UI to display and manage the entries
- UI will have a way to:
  - add new entries
  - mark an entry complete
  - remove all completed entries



- Each of the actions require the app to do a DOM operation to:
  - ◇ create
  - ◇ insert
  - ◇ update
  - ◇ delete

DOM nodes.

  - ◇ You don't have to worry about DOM operations in React
    - ■ Nor when DOM ops need to happen
    - ■ Nor how the DOM ops efficiently perform
- Simply place the todos array in the “state” of the app
  - ◇ Then use React lang to “command” React to display that state in a certain way in UI:

```
<header>TODO List</header>

<ul>
  {todos.map(todo =>
    <li>{todo.body}</li>
  )}
</ul>

// Other form elements...
```

  - ◇ (Here we mapped an array of JS objects into an array of React elements)

◇ Afterwards you can simply focus on just doing data operations on the todos array

■ Add, remove, and update the items of the array and React will reflect the changes you make on this object in the UI in browser

• The above mental model about modeling the UI based on final state is easier to understand & work with

◇ especially with lots of data transitions

■ Example: a view of how many friends are online

■ View's “state” will simply be a single number of those online, instead of:

- that a moment ago three friends came online, then one of them disconnected, and then two more joined.

- Just “four friends are online” is the state

React's tree reconciliation

• **Before React, we needed to work on the DOM (browser's) API**

◇ We avoided traversing the DOM as much as possible

■ Any operation on DOM is done on a single thread responsible for everything else on the browser:

- like reactions to user events

→ scrolling

→ typing

→ resizing etc

• **Any expensive opertation on the DOM = slow, janky experience for user**

◇ applications must do minimum operations

◇ apps must batch as muct operations AMAP

◇ *React solves this problem*

• **React, when told to render a tree of elements to the browser:**

◇ generates a virtual representation of the tree

◇ keeps virt. in memory for later

◇ proceed to perform the DOM operations that make the tree show in browser

• **React, when told to update the tree of elements previously rendered:**
  ◇ generates a new virtual rep. of updated tree
  ◇ React has two versions of tree in memory
• **React, to render the updated tree in browser:**
  ◇ doesn't discard what's been rendered
  ◇ compares the 2 virtual versions of tree in memory
  ◇ compute the differences between the two
  ◇ figure which sub-trees in main need to be updated
  ◇ only update the changed sub-trees in browser
• **This is known as the *tree reconiliation algorithm***
  ◇ makes React an efficient way to work with a browser's DOM tree
  ◇ example later

• **Why React gained popularity:**
  ◇ declarative outcomes-based language
  ◇ efficient tree reconciliation
  ◇ Working with the DOM API is hard
    ■ React gives developers the ability to work with a **"virtual" browser** that is friendlier than the real browser
    ■ basically acts like your **agent** who will do the communication with the DOM on your behalf
  ◇ React is often given the "**Just JavaScript**" label
    ■ has a very small API to learn
    ■ your JavaScript skills are what make you a better React developer
    ■ an advantage over libraries with bigger APIs
    ■ the React API is mostly just functions
    - (and optionally classes if you need them)
    - When you hear that a UI view is a function of your data, in React that’s literally the case
  ◇ Learning React pays off big-time for iOS and Android mobile applications
    ■ **React Native** allows you to build native mobile applications
    - You can even share some logic between your web, iOS, and

Android applications

◇ React team at Facebook tests all improvements and new features that get introduced to React right there on **facebook.com**, which increases the trust in the library among the community

■ rare to see big and serious bugs in React releases because they only get released after thorough production testing

■ React also powers other heavily used web applications like Netflix, Twitter, Airbnb, and many more

Your first React example

• To see the practical benefit of the tree reconciliation process and the big difference it makes, let's work through a simple example focused on just that concept:

◇ generate and update a tree of HTML elements twice:

■ once using the native Web API

■ then using the React API (and its reconciliation work)

◇ To keep this example simple, I will not use components or JSX (the JavaScript extension that's popularly used with React)

■ will also do the update operation inside a JavaScript **interval** timer

■ not how we write React applications but let's focus on one concept at a time

◇ Start with this jsComplete playground session: https://jscomplete.com/playground/react-dom1

■ The session has an HTML element rendered using 2 methods:

- Directly with Web DOM API:

→
```
document.getElementById('mountNode').innerHTML = `
  <div>
    Hello HTML
  </div>
`;
```

- Using React's API

→
```
ReactDOM.render(
  React.createElement(
```

```
    'div',
    null,
    'Hello React',
  ),
  document.getElementById('mountNode2'),
);
```

→ The `ReactDOM.render` method and `React.createElement` method are the core API methods in a React application, a React webapp can't exist without them. Let's explain them:

ReactDOM.render

• Entry point for a React app into the browser's DOM
• Holds 2 *arguments*:
  ◇ WHAT to render to the browser
    ■ This is always a "React element"
  ◇ WHERE to render that React element in the browser
    ■ has to be a valid DOM node that exists in the statically rendered HTML
    - example above ("first React example") uses a special `mountNode2` element which exists in the playground's display area (the first mountNode is used for the native version)
• What is a React element?
  ◇ VIRTUAL element describing a DOM element
  ◇ What the React.createElement API method returns

React.createElement

• Instead of working with Strings, in React we represent DOM elements with **objects**
  ◇ We do this with calls to the `React.createElement` method
  ◇ Objects = React elements
• `React.createElement` function has many arguments:

1- The HTML “tag”