Professional React Development

version 19.2.0,

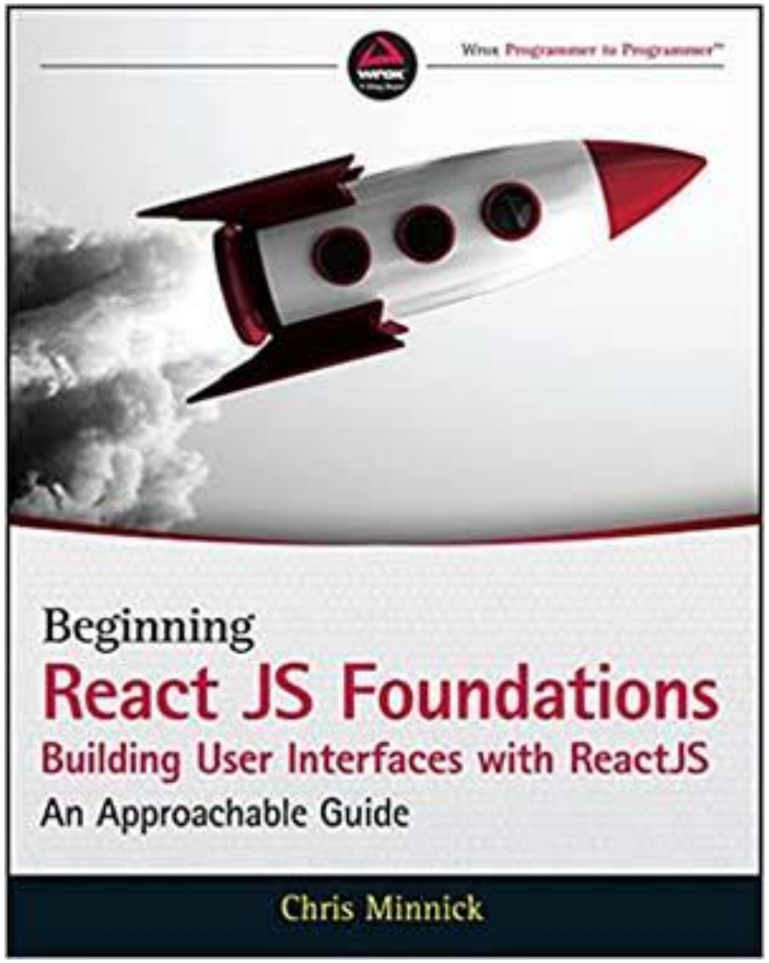
February 2025

Introduction

- Objectives
  - Who am I?
  - Who are you?
  - Daily Schedule
  - Course Schedule and Syllabus

About Me

- Chris Minnick
  - Author of 20+ books, including **React JS Foundations** (Wiley, March 2022) and JavaScript All-In-One For Dummies (Wiley, May 2023)
  - 20+ years experience in full-stack web development
  - Training & coding with React since 2015


Wrox Programmer to Programmer™
Beginning
React JS Foundations
Building User Interfaces with ReactJS
An Approachable Guide
Chris Minnick

Introductions

- What's your name?
- What do you do?
- Where are you?
- JavaScript level (beginner, intermediate, advanced)?
- What do you want to know at the end of this course?
- What do you do for fun?

The Big Picture

Some of the Topics Covered in this Course

- React.js
- Modern JavaScript
- Testing
- TypeScript
- Backend integration
- Redux
- Error Handling
- Best Practices
- React Router
- Advanced Topics

Course Repo and More Examples

- github.com/chrisminnick/professional-reactjs
  - Here you'll find all the files you need to complete the labs, as well as the completed solutions for each lab.
- https://reactjsfoundations.com
  - This is the website for *ReactJS Foundations*, by Chris Minnick (Wiley, March, 2022). It contains all the example code from the book, and more.

Daily Schedule

- 9:00 - 12:00: Lecture / Labs
- Breaks on the hours (5-10 minutes)
- 1 hour lunch break
- 1:00 - 4:00: Lecture / Labs
- Breaks on the hours
- 4:00 - 6:00 : labs and independent study

The Plan

Each day will be approximately 50% lecture and 50% labs

- **Monday:** Introduction to React, Advanced JavaScript, JSX
- **Tuesday:** React components, Props, Class vs. Function components, React State, Events
- **Wednesday:** Forms, Component Lifecycle, Async code, AJAX, Hooks, PropTypes, TypeScript
- **Thursday:** Testing, React Testing Library, Redux
- **Friday:** Redux Middleware, Routing, Local Storage, Deploying, Advanced Topics

The Labs

- Instructions: Professional-React-v19.2.0-labs.pdf
- Time to complete labs vary between 15 mins and 1 hour+.
- Let me know if something doesn't work.
- If you finish a lab early:
    - try lab challenges,
    - help other students,
    - do additional reading,
    - take a break.

What is React.js?

Component-based

Library for building user interfaces

What is React NOT?

- React is not a framework.
- React is not a web server.
- React is not a programming language.
- React is not a database.
- React is not a development environment.
- React is not the solution to every problem.

When can you use React?

- Complex single-page applications (SPAs) can be built entirely using React.
- React can generate static HTML on the server.
- React can be used to create native mobile apps.
- Universal App (server side components + client side components)

React Quick Start

- Objectives
  - Create a React App with Vite
  - Test and Run a React App

React with a Toolchain

Most Web Apps today are built using a set of tools that enable professional, team development.

The tools, collectively, are known as the JavaScript development toolchain.

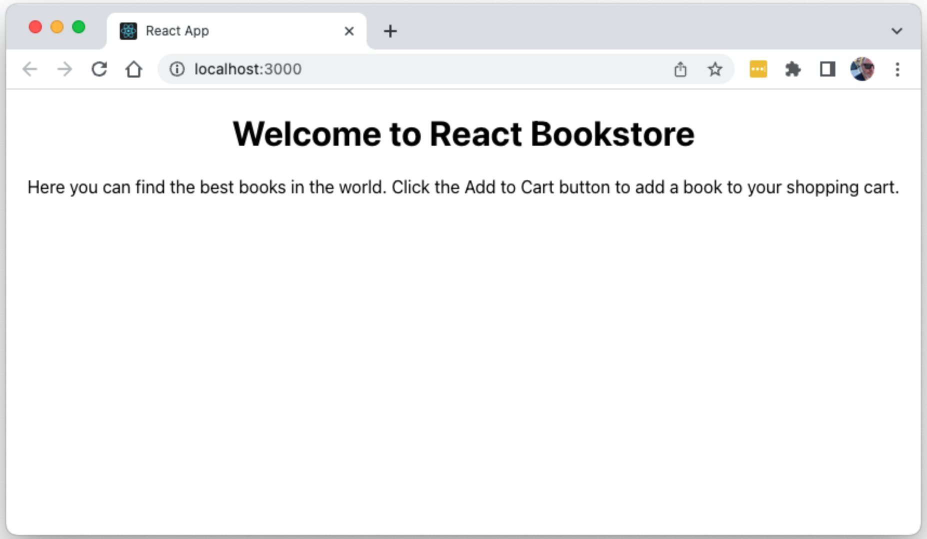
What is Vite?

- Creates JavaScript apps with no build configuration
- Supports multiple libraries / frameworks
- Simplifies setup of toolchain

Alternatives to Bootstrapping with Vite

- Build your own toolchain
  - Install and configure a module bundler or compiler such as Webpack, ESbuild, or Parcel
- Create React App
  - deprecated, but still works.
- Next JS
  - a framework for full-stack React applications

Lab 01:
Get Started with Vite / React



Welcome to React Bookstore

Here you can find the best books in the world. Click the Add to Cart button to add a book to your shopping cart.

Introduction to ReactJS

What is React?

A JavaScript library for building user interfaces

React Philosophy

Thinking in Components

Composition vs. Inheritance

Declarative vs. Imperative

Idiomatic JavaScript

Thinking in Components

A component describes a piece of the UI

Each component returns an element

An element can be used in other components

A React UI is a tree of elements

Thinking in Components

A component should be an independent piece of the UI that can be reused.

Single Responsibility says that a component should only have one reason to change.

Composition vs. Inheritance

Inheritance uses parent classes to create more specific classes.

- class WelcomeMessage extends Message

React favors creation of configurable and composable components instead.

```
class WelcomeMessage extends
   React.Component {
  return (
    <Message text="Welcome  to
    my app!" />
  )
}
```

Imperative vs. Declarative

Imperative

- Focuses on the steps to complete a task
- Example:
  - Walk to the stairs
  - Walk down stairs
  - Go to the kitchen
  - Open refrigerator
  - Take out salami, cheese, mustard
  - Put salami, cheese, mustard on bread

Declarative

- Focuses on what to do without saying how
  - Bring me a sandwich.

Imperative vs. Declarative Screen Updates

Imperative

```
getElementById('header')
  .innerHTML = "Welcome to my App";
```

Declarative

```
render (
  return <header>
           Welcome to my App
         </header>;
);
```

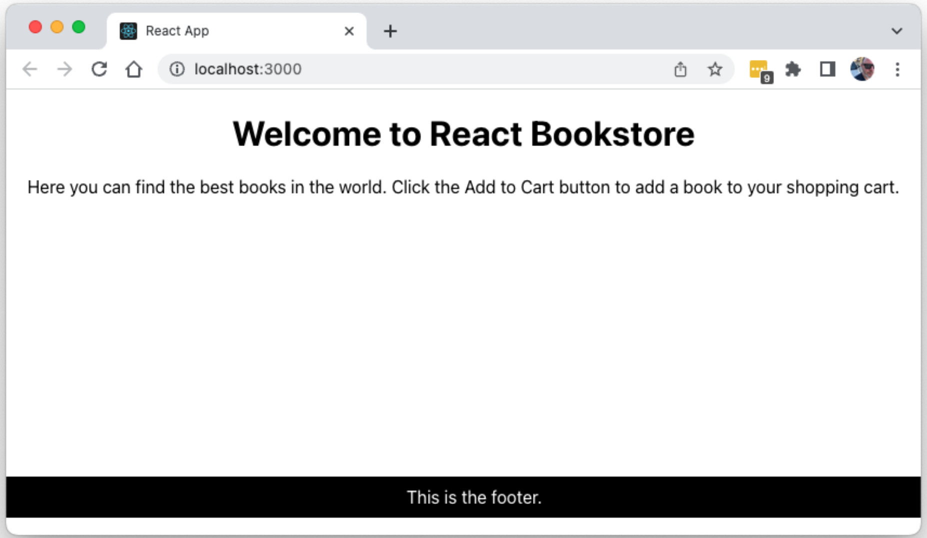
Idiomatic JavaScript

React is just JavaScript

React components are easily understandable to anyone who knows JavaScript.

If you know JavaScript, you can quickly start writing React.

Lab 02: Your First Component



Welcome to React Bookstore

Here you can find the best books in the world. Click the Add to Cart button to add a book to your shopping cart.

This is the footer.

How React Works

Implements Web Components using JavaScript

Components are rendered using a Virtual DOM

State is stored in a “state” object

Updating a component’s state causes the component to re-render.

Updates to the Virtual DOM are applied to the actual DOM using the ReactDOM library’s render method.

Virtual DOM

Virtual DOM is updated (in memory) as the **state** of the data model changes.

React calculates the difference between the Virtual DOM and the real DOM.

React updates only what needs to be updated in the DOM.

Batches changes

Virtual DOM vs. HTML DOM

Virtual DOM is a local and simplified copy of the HTML DOM

- (It's an abstraction of an abstraction)

The goal of the Virtual DOM is to only re-render when the **state** changes.

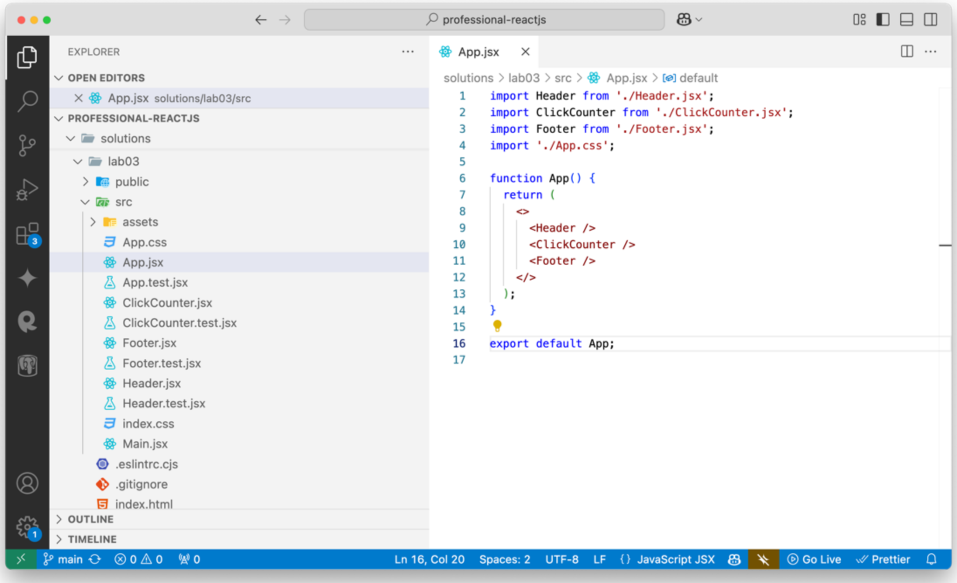
- This makes it more efficient than direct DOM manipulation.

Developers can write code as if the entire tree is being re-rendered.

- This makes it easier to understand.

Behind the scenes, React/Virtual DOM works out the details and creates a patch for the HTML DOM, which causes the browser to re-render the changed part of the scene.

Lab 03: Create Tests and More Components



```jsx
import Header from './Header.jsx';
import ClickCounter from './ClickCounter.jsx';
import Footer from './Footer.jsx';
import './App.css';

function App() {
  return (
    <>
      <Header />
      <ClickCounter />
      <Footer />
    </>
  );
}

export default App;
```

ReactDOM

- Exported from react-dom package.
- Provides methods that are used at the top-level of your app.

ReactDOM/client

Module for working with the Browser DOM

Methods

- createRoot() - Creates a React root, which can be used to render a React element in the browser.
- hydrateRoot() - Creates a React root from HTML rendered by ReactDOMServer

ReactDOM/server

- Used for rendering React components on the server (in Node.js)
- Methods
  - renderToStaticMarkup()
  - renderToString()
  - renderToReadableStream()
  - renderToPipeableStream()
  - renderToStaticNodeStream()

root.render()

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
const container = document.getElementById('root');
const root = ReactDOM.createRoot(container);
root.render(<App />);
```

Other Rendering Engines

React Native	• Renders to native mobile apps
ReactDOM/server	• Renders to static HTML
React Konsul	• Renders to the browser console
react-pdf	• Renders to PDF

Using JSX

- Objectives
- Write JSX
- Use React with JSX

What is JSX?

- A syntax extension to JavaScript
- Allows you to put markup in JavaScript
- Is not part of React, but it is used in most React codebases.

JSX is not HTML

- XML syntax required
  - Elements must be closed
- Attributes use DOM property names
  - className instead of class, htmlFor instead of for
- React components start with upper-case
- HTML elements start with lower-case
- Attributes become props in the child

Components Return UI

- A component can only return one thing.
  - string
  - number
  - array
  - Boolean
  - a single JSX element
  - undefined
- As long as your elements are properly nested, the single element can have as many children as necessary.

Elements are Instances of Components

- A component is a function or a class.
- An element invokes the function or creates an instance of the class.

Two Types of Components

- React DOM has built-in components that produce HTML
  - Examples: <div>, <p>, <h1>, <form>, <input>
  - HTML elements are always lowercase.
- You create custom components
  - Custom components are upper camelcase.
  - May be made of HTML components and other custom components.
  - Examples: <LoginForm>, <WeatherChart>, <StockTicker>

Using React with JSX

```
function LoginBox(props){
return (
   <div>
   <label>Log In <input type="text" id="username"
      placeholder={props.placeholderText} />
   </label>
   </div>
   )
}
```

Using React without JSX

```
return(
  React.createElement("div",null,
    React.createElement("label",null,
      "Log In",
        React.createElement("input",
          { type: "text",
            id: "username" }
        )
      )
    )
  );
```

Using Literal JavaScript in JSX

- Enclose JavaScript that shouldn't be interpreted as JSX in curly braces
  - Object literals must be in double curly braces
  - Enclose comments in curly braces
- Literal JS in JSX must be an expression.
  - Expressions resolve to a value.
  - Not an expression: function doSomething(){...}
  - Is an expression: doSomething()

Literal JS in JSX

```
return (
  <p>Welcome, {props.firstName}.
     Today is
     {new Date().toLocaleDateString()}
  </p>
);
```

Conditional Rendering with JSX

- Three Methods
  - With Element Variables
  - With &&
  - With the conditional operator

Conditional Rendering with Element Variables

```
function Welcome({loggedIn}) {
  let header;
  if (loggedIn) {
    header = <Header />;
  } else {
    header = <Login />;
}
return ( <div> {header} </div> ) }
export default Welcome;
```

Conditional Rendering with the && Logical Operator

```
function Welcome({loggedIn}){
  return (
    <div> {loggedIn&&<Header />}
    Note: if you don't see the
    welcome messsage, you're not
    logged in. </div>
  )
}
export default Welcome;
```

Conditional Rendering with the Conditional Operator

```
function Welcome({loggedIn}){
  return(
    <div>
    {loggedIn?<Header />:<Login />}
    </div>
  )
}
export default Welcome;
```

React.Fragment

- The single-element return rule in React can create unnecessary elements in the browser.
- React.Fragment can be used to wrap multiple elements without rendering an element.

```
return(
  <React.Fragment>
    <h1>The heading</h1>
    <h2>The subheading</h2>
  </React.Fragment>
)
```

React.Fragment Shorthand

- You can use <> ... </> as a shorthand for React.Fragment.

```
function MyComponent(){
  return(
    <>
      <h1>The heading</h1>
      <h2>The subheading</h2>
    </>
  );
}

export default MyComponent;
```

HTML element attributes

- React DOM supports all HTML attributes on built in HTML elements.
- A few are renamed.
- Multi-word attributes are lower camelCase
- Example:

```
<div id="companyLogo" className="companyLogo">
  <img src="/images/logo.png" height="100"
   width="200" alt="Our company logo" />
</div>
```

props in Custom Components

- Props are passed from the parent component to its children
- JSX attributes are passed to the child as properties of an object.

- Parent: `<Hello name="Chris" />`

- Class Child:

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

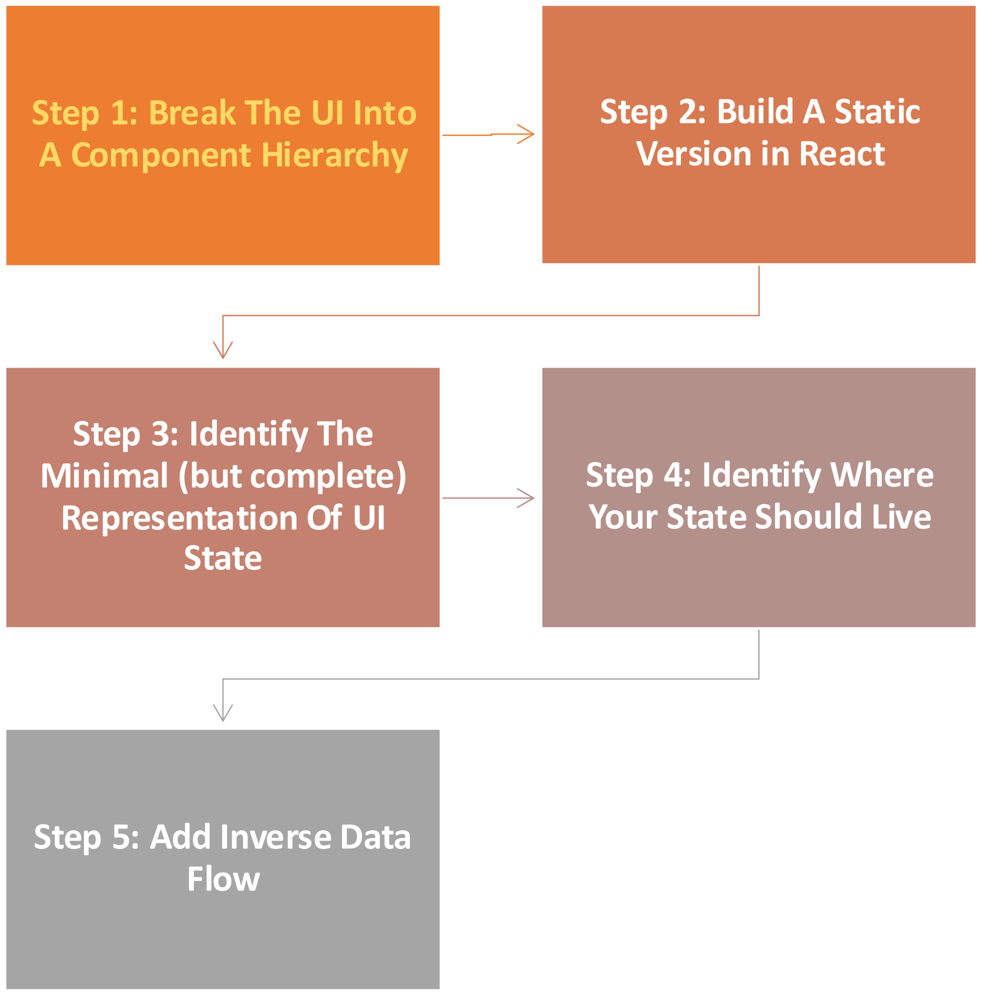
props.children

- Automatically passed in props object.
- Returns the child elements of a React element.
- Makes composition of elements possible.

```
<BorderBox>
  <p>The first paragraph.</p>
  <p>The second paragraph.</p>
</BorderBox>

function BorderBox(props){
  return(<div style={{border:"1px solid black"}}>
    {props.children}
  </div>);
}
```

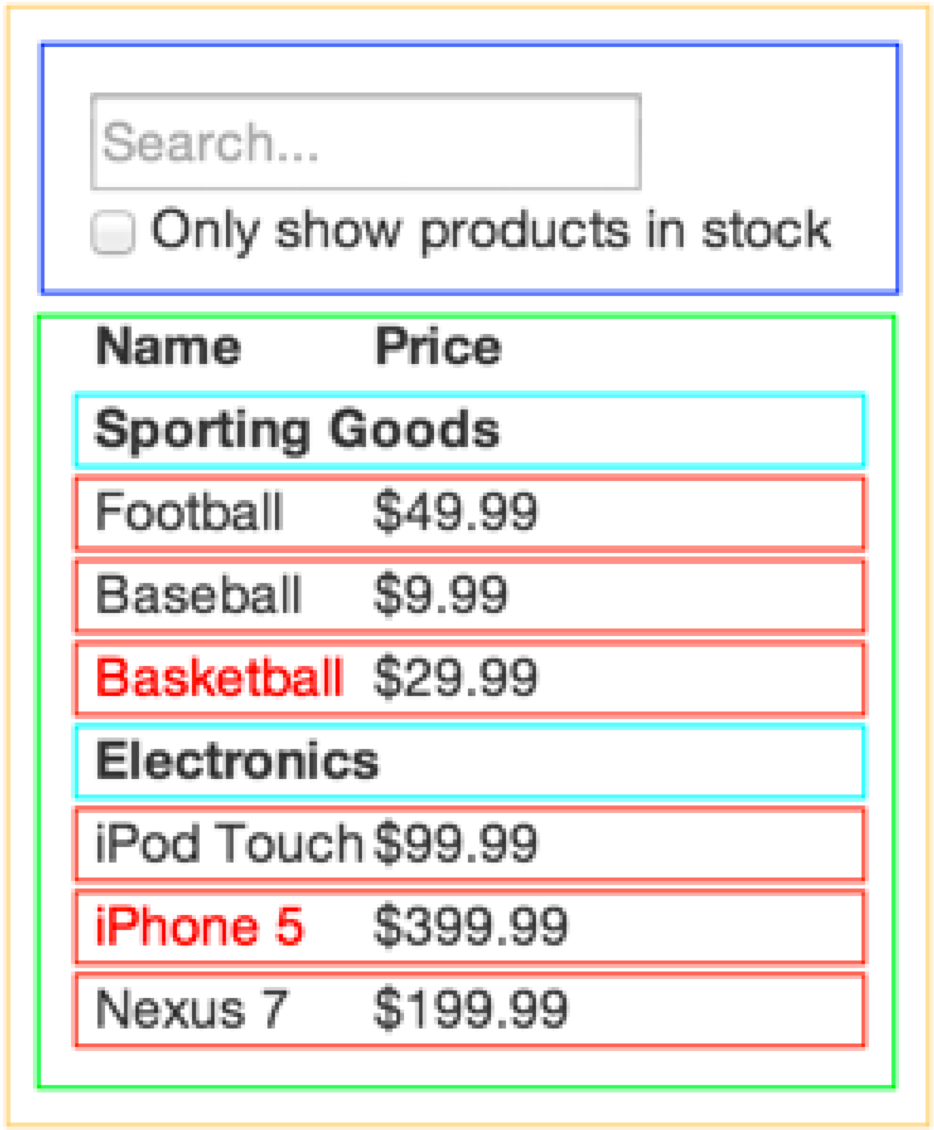
React Development Process

Step 1: Break The UI Into A Component Hierarchy


Step 1: Break The UI Into A Component Hierarchy
Step 2: Build A Static Version in React
Step 3: Identify The Minimal (but complete) Representation Of UI State
Step 4: Identify Where Your State Should Live
Step 5: Add Inverse Data Flow

Creating a Component Hierarchy

How do you know what should be a component?



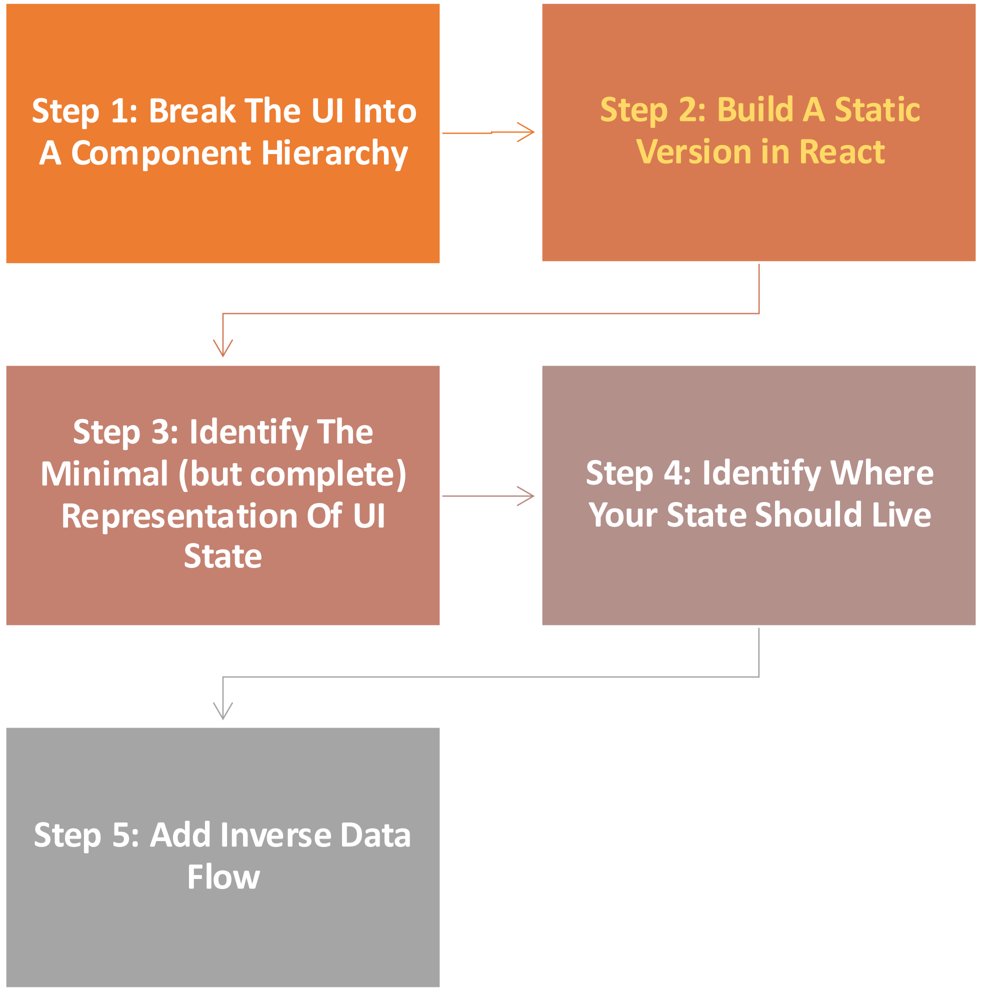
Search...

Only show products in stock

Name	Price
Sporting Goods	
Football	$49.99
Baseball	$9.99
Basketball	$29.99
Electronics	
iPod Touch	$99.99
iPhone 5	$399.99
Nexus 7	$199.99

React Development Process

Step 2: Build A Static Version in React


Step 1: Break The UI Into A Component Hierarchy
Step 2: Build A Static Version in React
Step 3: Identify The Minimal (but complete) Representation Of UI State
Step 4: Identify Where Your State Should Live
Step 5: Add Inverse Data Flow

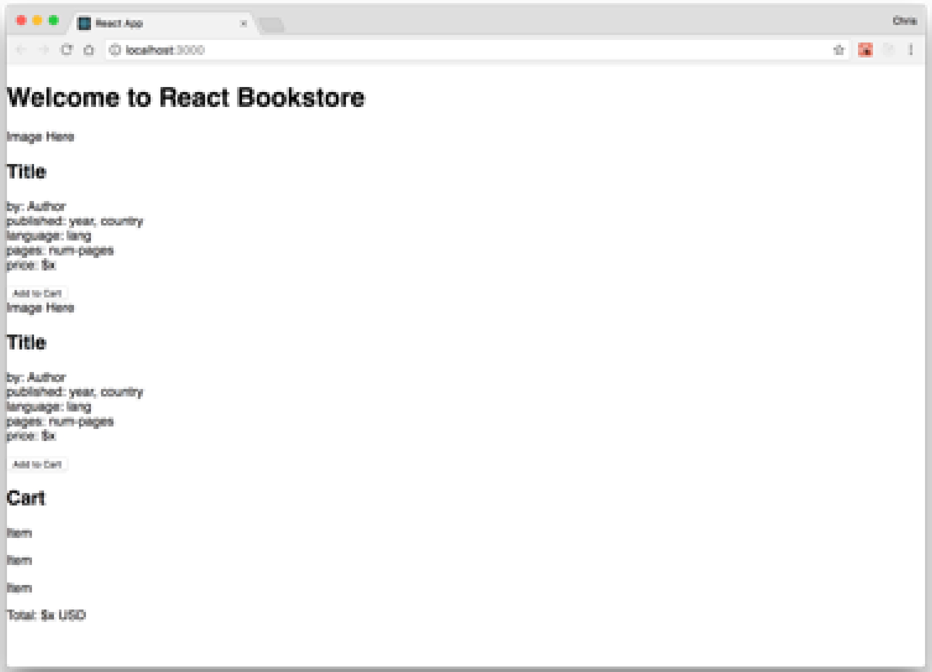
Goals of a Static Version

Render the UI with no interactivity

Create a library of reusable components that render the data model

```
function ProductCategoryRow(){
 return (
     <tr>
       <th colSpan="2">
         test data
       </th>
     </tr>
     );
};
export default
ProductCategoryRow;
```

Lab 04: Static Version


React App

localhost:3000

Welcome to React Bookstore

Image Here

Title

by: Author
published: year, country
language: lang
pages: num-pages
price: $x

Add to Cart

Image Here

Title

by: Author
published: year, country
language: lang
pages: num-pages
price: $x

Add to Cart

Cart

Item

Item

Item

Total: $x USD

Advanced JavaScript for React

- Objectives
  - Use arrow functions and block-scoped variables
  - Use classes and modules
  - Understanding ‘this’

Variable Scoping with `const` and `let`

- `const` creates constants
  - Immutable and block-scoped
  - Cannot be reassigned new content
  - The assigned content isn't immutable, however,
    - If you assign an object or array to a constant, the properties or elements of the variable can be changed.
- `let` creates block-scoped variables
  - Main difference between `let` and `var` is that the scope of `var` is the entire enclosing function.
  - Redeclaring a variable with `let` raises a syntax error
- Unlike with var, variables created with const and let aren't hoisted.
    - You can't reference a variable before its declaration.

let vs. var

var

```
var a = 5;
var b = 10;

if (a===5) {
   var a = 4;
   var b = 1;
}
console.log(a); //4
console.log(b); //1
```

let

```
let a = 5;
let b = 10;

if (a===5) {
   let a = 4;
   let b = 1;
}
console.log(a); //5
console.log(b); //10
```

Arrow Functions

- Compact alternative to function expressions.
  - (params) => {expression}
- Doesn't have its own `this`
  - Lexical scope – `this` refers to the enclosing scope
  - Should not be used as methods.
- Can't be used for constructor functions
- Can't be used with call, apply, or bind
- Are always anonymous function expressions.

Arrow Functions – basic example

```
const sum = (num1,num2) => {
   return num1 + num2;
}
```

Arrow Functions (cont.)

- Parentheses around parameters are optional if there's only one param.

```
const sum = numbers => {
  return numbers.reduce((sum,current)=>sum+current);
}
```

- return keyword and curly braces are optional if the function only returns data.

```
const sum = numbers =>
  numbers.reduce(
    (sum,current)=>sum+current
  );
```

Default Parameter Handling

- OLD

```
function f (x, y, z) {
    if (y === undefined)
        y = 0;
    if (z === undefined)
        z = 13;
    return x + y + z;
};
```

- NEW

```
function myFunc (x, y = 0, z = 13) {
  return x + y + z;
}
```

Rest Parameter

- Aggregation of remaining arguments into single parameter of variadic functions.

```
function myFunc (x, y, ...a) {
    return (x + y) * a.length;
}
console.log(myFunc(1, 2, "hello", true, 7));
```

- http://jsbin.com/pisupa/edit?js,console

Spread Operator

- Spreading of elements of an iterable collection (like an array or a string) into both literal elements and individual function parameters.

```
let params = [ "hello", true, 7 ];
let other = [ 1, 2, ...params ];
console.log(other); // [1, 2, "hello", true, 7]


console.log(MyFunc(1, 2, ...params));


let str = "foo";
let chars = [ ...str ]; // [ "f", "o", "o" ]
```

- http://jsbin.com/guxika/edit?js,console

Template Literals

- String Interpolation

```
let customer = { name: "Penny" }
let order = { price: 4, product: "parts", quantity: 6 }
message = `Hi, ${customer.name}. Thank you for your order
of ${order.quantity} ${order.product} at ${order.price}.`;
```

- http://jsbin.com/pusako/edit?js,console

Method notation in object property definitions

```
const shoppingCart =
{
  itemsInCart: [],
  addToCart (id) {
    this.itemsInCart.push(id);
  },
};
```

Array Destructuring

- Intuitive and flexible destructuring of Arrays into individual variables during assignment

```
const list = [ 1, 2, 3 ];
let [ a, , b ] = list; // a = 1 , b = 3
[ b, a ] = [ a, b ];
```

- http://jsbin.com/yafage/edit?js,console

Object Destructuring

- Flexible destructuring of Objects into individual variables during assignment

```
let { a, b, c} = {a:1, b:2, c:3};
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

- http://jsbin.com/kuvizu/edit?js,console

`Array.map()`

- **Array.map()**
  - Creates a new array with the results of calling a provided function on every element in this array.
- Syntax
  - const newarray = arr.map(arrayElement => arrayElement + 1);
- Parameters passed to the callback
  - `currentValue`
    - The current element being processed
  - `index`
    - The index (number) of the current element
  - `array`
    - The array map was called upon

Array.filter()

- **Array.filter()**
  - Creates a new array with the results of a test.
- Syntax
  - let new_array = arr.filter(test)
- Example

```
const customersNamedBill =
customerNames.filter(name=>name === "Bill");
```

Array.reduce()

- **Array.reduce()**
  - Executes a reducer function on each element of an array.
- Syntax

```
let value =
    arr.reduce((previous,current)=>previous +
          current,initValue);
```

- Example

```
let orderTotal = items.reduce((total,item) =>
total + item.price,0);
```

References, Shallow, and Deep Copies

- JavaScript arrays and objects are reference values
- If you use the assignment operator you create a reference. The new name references the same object.

```
let arr = ['red','green','blue'];

let newArr = arr;

newArr.push('orange');

arr
```

> ['red','green','blue','orange']

Shallow Copy an Array

- A shallow copy is an array created by copying each element of the original array to the new array.
- Can be done in several ways:
  - Using a loop
    ```
    for (i = 0; i < numbers.length; i++) {
      numbersCopy[i] = numbers[i];
    }
    ```
  - Using the slice method
    ```
    numbersCopy = numbers.slice();
    ```
  - Using the spread operator
    ```
    numbersCopy = [...numbers];
    ```

Shallow Copy an Object

- A shallow copy is an array created by copying each property of the original object to the new object.
- Primitive properties are separate from the original object
- Object properties are references to the original
- Can be done using spread operator:

```
let person = { foo: 'bar', x: 0 };
let p1 = { ...person };
```

- Or by using object.assign:

```
let p2 = Object.assign({}, person);
```

Chapter 5: Modularity

Objectives

- Explain modularity
- Learn different methods of using modules in JS
- Understand methods of front-end module management

Why is Modularity Important?

- Individual modules can be tested
- Allows distributed development
- Enables code reuse
- Reduce coupling
- Increase cohesion

CommonJS

- Modularity for JavaScript outside of the browser
- Node.js is a CommonJS module implementation
- uses `require` to include modules

- export an anonymous function

hello.js

```
module.exports = function () {
  console.log("hello!");
}
```

app.js

```
let hello = require("./hello.js");
```

- export a named function

hello.js

```
exports.hello = function () {
  console.log("hello!");
}
```

app.js

```
let hello = require("./hello.js").hello;
```

ES Modules

- 2 Types
  - named exports
    - multiple per module
  - default exports
    - 1 per module

- Named export
- lib.js

```
export function square(x) {
  return x * x;
}
```

- main.js

```
import {square} from 'lib';
```

- Default export
- myFunc.js

```
export default function() {
…
};
```

- main.js

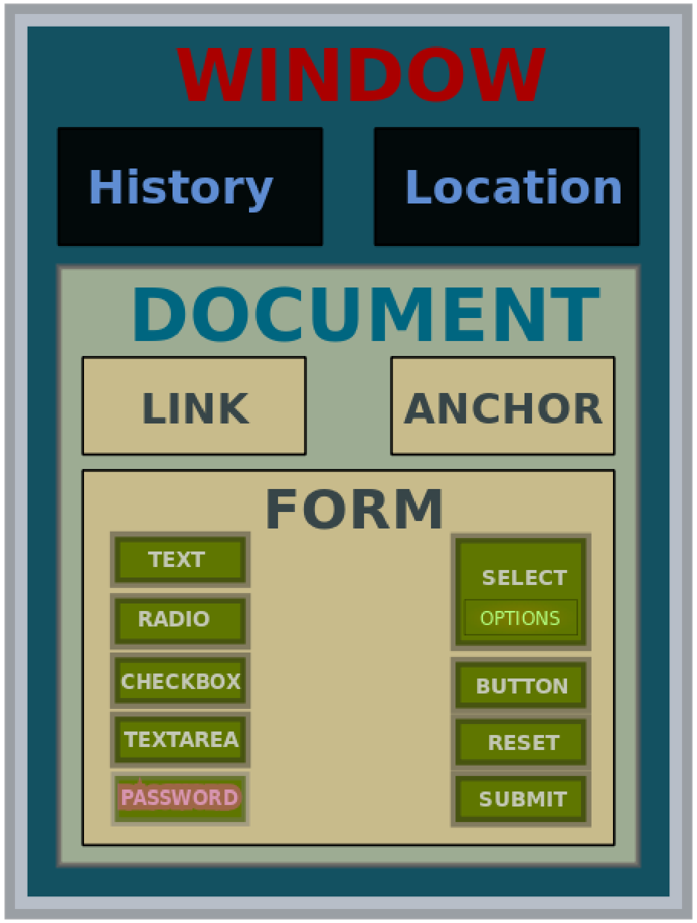
```
import myFunc from './myFunc';
```

The Document Object Model

- Objectives
  - Understand how the DOM works
  - Select DOM nodes
  - Manipulate the DOM with JavaScript

What is the DOM?

- JavaScript API for HTML documents
- Represents elements as a tree structure
- Objects in the tree can be addressed and manipulated using methods.



JohnManuel [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Understanding Nodes

- DOM interfaces that inherit from Node
  - Document
  - Element
  - CharacterData
    - Text
    - Comment
  - DocumentType
- Nodes inherit properties from EventTarget

EventTarget

- An interface implemented by objects that can receive events (aka event targets)
- Examples:
  - Element
  - Document
  - Window
- Many event targets support setting event handlers.

DOM Events

- Things that happen in the DOM
  - abort
  - beforeinpu
    t
  - blur
  - click
  - compositio
    nend
  - compositionup
    date
- dblclick
- error
- focus
- focusin
- focusout
- input
- keydown
- keyup
- load
- mousedown
- mouseenter
- mouseleave
- mousemove
- mouseout
- mouseover
- mouseup
- resize
- scroll
- select
- unload
- wheel

Element

- Interface for elements within a Document
- Inherits properties and methods from Node and EventTarget
- Most common properties:
  - `innerHTML`
  - `attributes`
  - `classList`
  - `id`
  - `tagName`
- Most common methods
  - `getElementById`
  - `addEventListener`
  - `querySelectorAll`

Manipulating HTML with the DOM

- You can get and set properties of HTML elements with JavaScript through the DOM

```
//starting HTML and DOM Element
<p id="favoriteMovie">The Matrix</p>

<script>
getElementById("favoriteMovie")
       .innerHTML = "The Godfather";
</script>

//updates DOM, which updates the browser
<p id="favoriteMovie">The Godfather</p>
```

Manipulating HTML with the DOM (cont.)

```
<ol id="favoriteSongs">
  <li class="song"></li>
  <li class="song"></li>
  <li class="song"></li>
</ol>

<script>
let mySongs=document
      .querySelectorAll("#favoriteSongs .song");
mySongs[0].innerHTML = "My New Favorite Song";
</script>
```

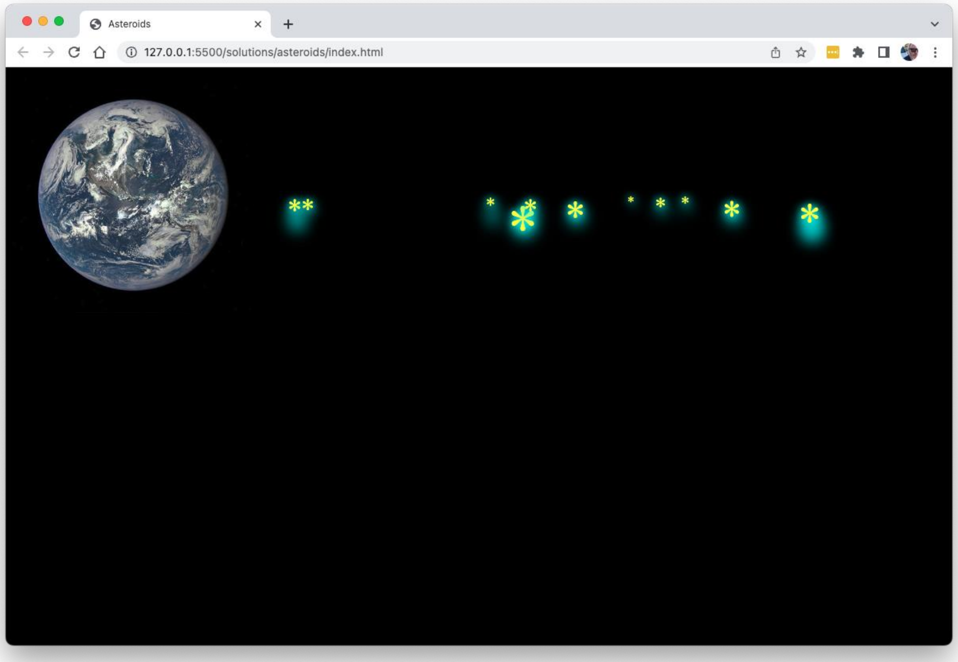
Manipulating HTML with JQuery

```
<ol id="favoriteSongs">
  <li class="song"></li>
  <li class="song"></li>
  <li class="song"></li>
</ol>

<script>
$("#favoriteSongs .song").first()
      .html("My New Favorite Song");
</script>
```

Manipulating HTML with **React**

```
function ThingsILike() {
  return (
    <FavoriteSongs songList = {["song1","song2","song3"]) />
  );
}

function FavoriteSongs(props) {
  return (
    <ol>
      {props.songList.map(song => <li>{song}</li>)}
    </ol>
  );
}
```

Lab 5: DOM Manipulation and Modern JavaScript



Styles in React

Two Approaches to Style

- Use CSS
  - Import CSS files
  - Use a CSS library (Bootstrap for example)
  - Use CSS Modules

- Use JavaScript
  - Inline styles
  - Style modules

Importing CSS

- If your module bundler is configured to bundle CSS files (Create React App does this by default), you can simply import CSS into a component.

```
import "styles.css";
  function MyComponent(props){
    return (
      <div className="article-link">
        <h1 className="title">
        {props.title}</h1>
      </div> );
    }
export default ArticleLink;
```

CSS Modules

- Importing CSS creates global styles.
- A CSS Module is a CSS file with class names scoped locally by default.
- Is supported by Create React App
- CSS Modules must be named using [name].modules.css
- Can be imported as JS
  - import styles from './MyStyles.module.css';
- Can be used with dot notation
  - return <button className={styles.warning}>Warning</button>

Inline Styles

- The style attribute on React's built-in elements will apply styles in the DOM using JavaScript

```
function WarningMessage(props){
 return (
    <p style={{
        color:"red",
        padding:"6px",
        backgroundColor:"#000000"}}>

      {props.warningMessage}

    </p> );
}
```

Style Objects

- Assign a style object to a variable to make it more reusable.

```
function WarningMessage(props){
 const textColors = {
    warning:{color:"red",
        padding:"6px",
        backgroundColor:"#000000"}
 };
 return (
    <p style={textColors.warning}>

      {props.warningMessage}

    </p> );
}
```

Style Modules

- Export style objects to create a library of reusable styles.

```
export const warningStyle =
{color:"red",padding:"6px",backgroundColor:"#000000"}
;

export const infoStyle =
{color:"yellow",padding:"6px",backgroundColor:"#00000
0"};

export const successStyle =
{color:"yellow",padding:"6px",backgroundColor:"#00000
0"};
```

CSS-in-JS

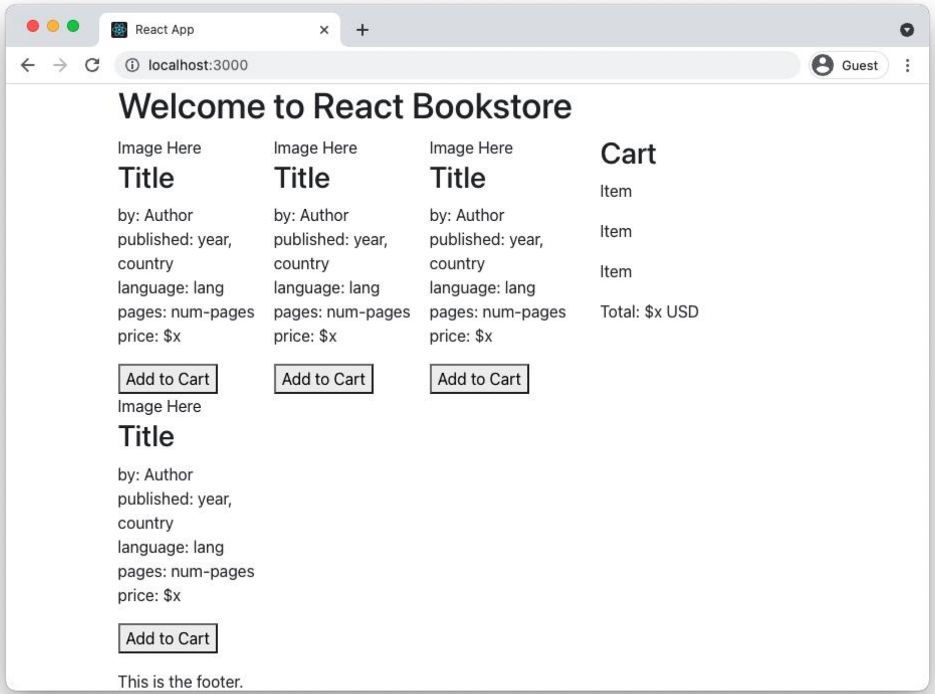
- There are MANY different CSS-in-JS libraries available.
- Most support Scoped CSS, Server-side Rendering, Automatic vendor prefixing, and more.
- Examples include:
  - styled-components
  - Emotion
  - Goober
  - Compiled
  - Fela
  - JSS
  - Treat
  - Styled JSX

Styled Components

1. import styled from 'styled-components';
2. Call styled.[object] function, passing in style info using Tagged Template Literal Notation.

```
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

<Title>This is a styled component</Title>
```

Lab 06: Styling React


React App

localhost:3000

Guest

Welcome to React Bookstore

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

Cart
Item
Item
Item
Total: $x USD

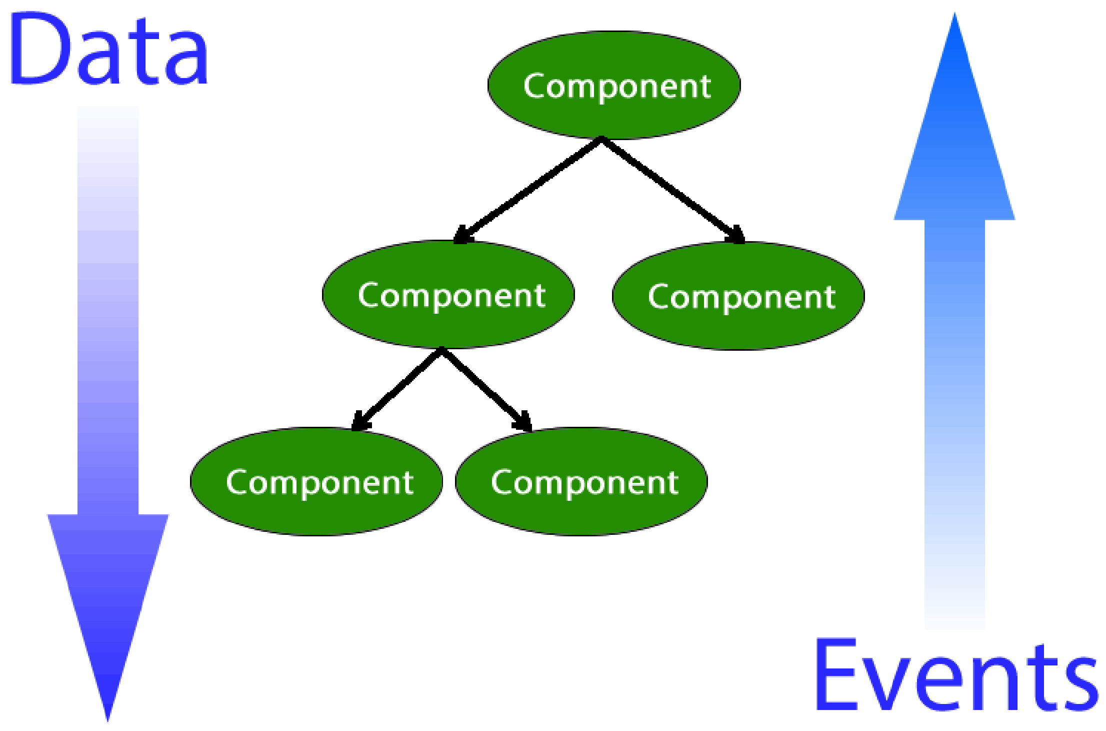
Image Here
Title
by: Author
published: year, country
language: lang
pages: num-pages
price: $x
Add to Cart

This is the footer.

Data Flow

One-way Data Flow

- Each UI element represents one component
- All data flows from owner to child



Props vs. State

Props

- Passed to a component instance using JSX attributes
- Immutable
- Better performance

State

- Internal data of a component instance
- State of the parent can be passed to child components as props.
- Mutable

Communication Between Components

- Parent to Child
    - pass props
- Child to Parent
    - Callback Functions
        - Parent passes a function to a child
            - `<MyChild myFunc={handleChildFunc} />`
        - Child calls the function
            - `props.myFunc();`

Communication Between Components (cont.)

- Between Siblings
  - Use a parent component.
- Any to Any
  - Observer Patterns
    - Components subscribe to messages.
    - Other components publish messages to subscribers.
  - Context
    - Provides data to an entire subtree

Container (aka Page) Components

- Wrap presentational components
- Contain the logic and state
- Correspond to pages
- Examples:
    - HomePage
    - ServicesPage

Presentational Components

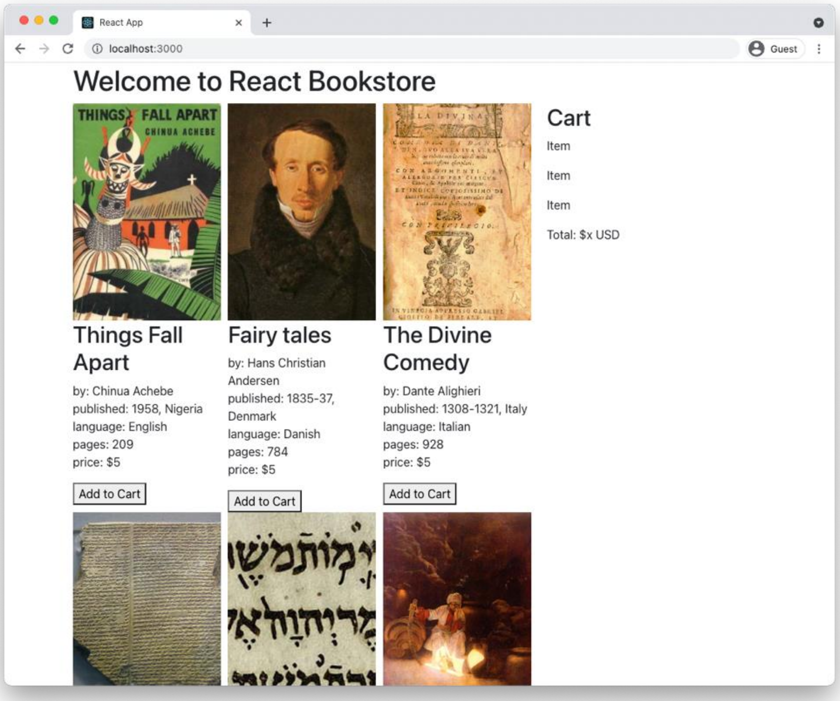
- Components that just output presentation
- Contain no logic
- Responsible for generating UI elements
- Display data passed by Pages

Folder Structure

- Organize projects by Feature
- Keep Pages and Presentational components separate

```
- /pages
  - /pages/App.jsx
  - /pages/ShoppingCart.jsx
  - /pages/ShoppingCart.css
- /components
  - /components/Header.jsx
  - /components/ProductInfo.jsx
- /services
-   /services/api.js
- /context
- /hooks
- /index.jsx
```

Lab 07: Props and Containers (aka Pages)



Welcome to React Bookstore

Things Fall Apart
by: Chinua Achebe
published: 1958, Nigeria
language: English
pages: 209
price: $5
Add to Cart

Fairy tales
by: Hans Christian Andersen
published: 1835-37, Denmark
language: Danish
pages: 784
price: $5
Add to Cart

The Divine Comedy
by: Dante Alighieri
published: 1308-1321, Italy
language: Italian
pages: 928
price: $5
Add to Cart

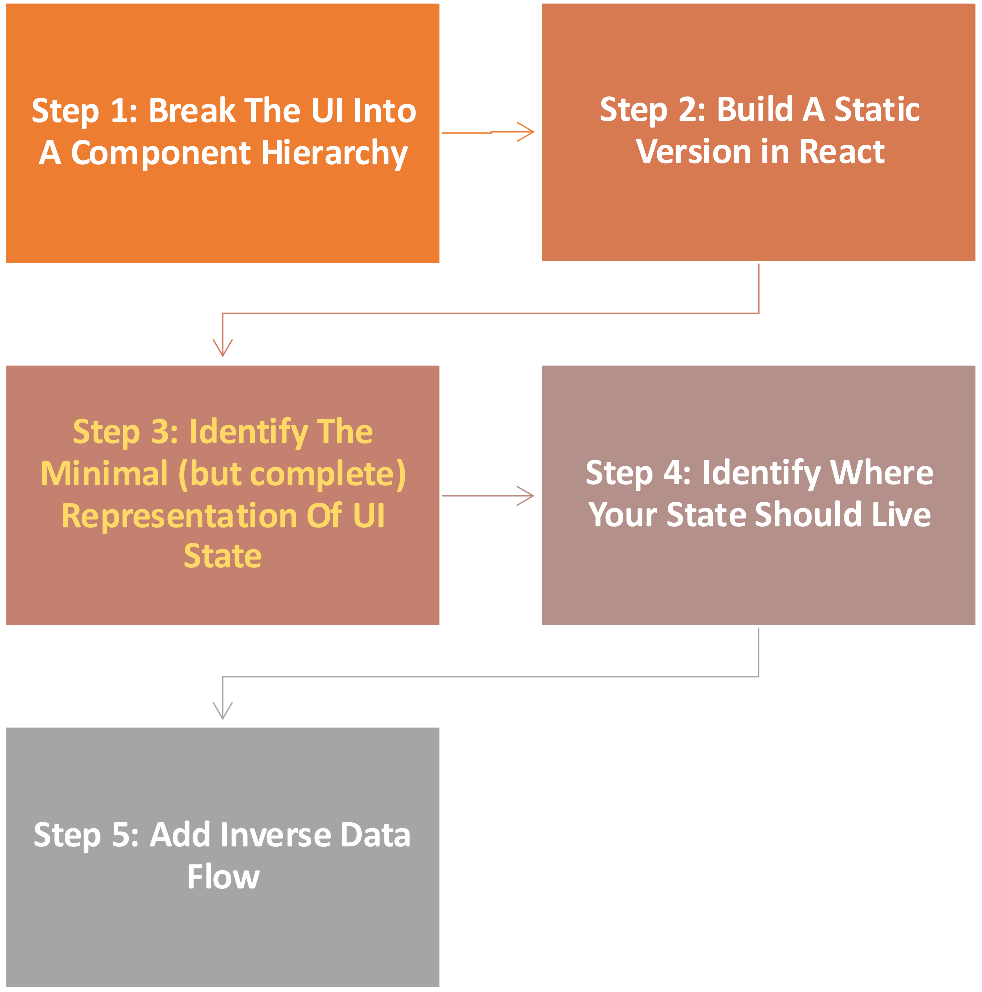
Cart
Item
Item
Item
Total: $x USD

State

React Development Process

Step 3:

Identify The Minimal (but complete) Representation Of UI State


Step 1: Break The UI Into A Component Hierarchy
Step 2: Build A Static Version in React
Step 3: Identify The Minimal (but complete) Representation Of UI State
Step 4: Identify Where Your State Should Live
Step 5: Add Inverse Data Flow

What is State?

- A JavaScript object used to hold INTERNAL data that influences the output of the component.

How State Affects render()

- The state object can only be changed using a state setting function. This method modifies the state object and then causes the component to render.

How to Know if it Should Be State

- Is it state?
    - Is it passed from the parent via props?
        - Probably not state
    - Does it change over time?
        - Might be state
    - Can you compute it based on any other state or props in your application?
        - Probably not state

Setting Initial State

- In Class Components:

```
class MyComponent extends React.Component {
  constructor() {
    this.state = { /* some initial state */ }
  }
```

- In Function Components:
  - useState Hook returns a stateful variable and a function for setting it.
  - Argument passed to useState is the initial value of the variable.

```
function MyComponent(){
  const [myState, setmyState] = useState("hi!");
}
```

Updating State

in Class Components

- setState()
  - Takes an object and merges the object into the state object.

in Function Components

- setter function returned by useState
  - replaces current value with argument

setState

- Function for updating a class component's state outside of the constructor.
- Takes an object or a function as its argument and uses the argument to schedule an update to the state.
- setState is asynchronous

setState with an Object

- The object will be merged with the current state.

setState with a Function

- The function receives the state and props of the component and returns an object.
- The object will be merged with the current state.
- Use a function when the new state depends on the current state.

useState setter function

- The 2nd element of the array returned by the useState hook is a function for updating the state variable (the first element returned by the hook).
- Unlike setState in class components, the updater function returned by useState does not merge the new state, it replaces the old state with the new state.

Valid arguments

- You can pass any value or a function to a setter function.
- If the new state depends on the old state, pass a function to the setter function.

What to Put in State

- Properties of a component that can change over time.

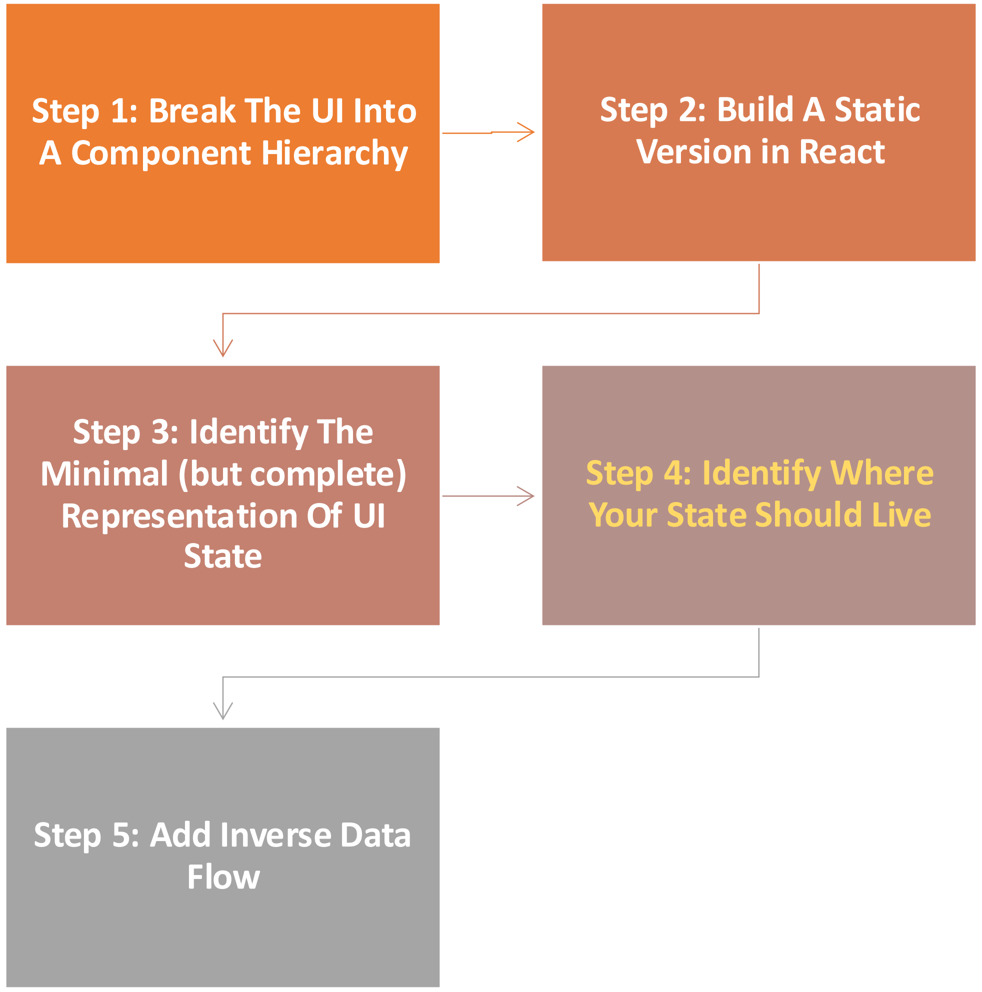
What Not to Put in State

- Anything that can be calculated.
- Anything passed down from a parent component.

React Development Process

Step 4:

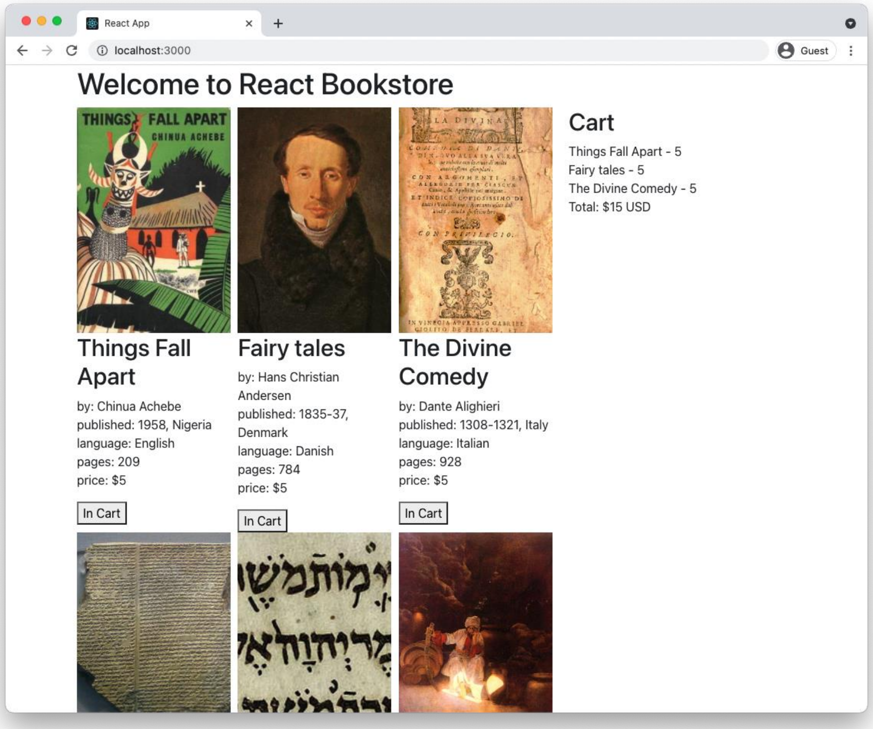
Identify Where Your State Should Live


Step 1: Break The UI Into A Component Hierarchy
Step 2: Build A Static Version in React
Step 3: Identify The Minimal (but complete) Representation Of UI State
Step 4: Identify Where Your State Should Live
Step 5: Add Inverse Data Flow

Where Should Your State Live?

- For each piece of state:
  - Identify every component that renders something based on that state.
    - Find a common owner component
    - The common owner or a component higher in the hierarchy should own the state.
    - If you can't find a common owner, create a new component higher up in the hierarchy just for holding the state.

Lab 08: Adding State


React App
localhost:3000
Guest

Welcome to React Bookstore

Things Fall Apart
by: Chinua Achebe
published: 1958, Nigeria
language: English
pages: 209
price: $5
In Cart

Fairy tales
by: Hans Christian Andersen
published: 1835-37, Denmark
language: Danish
pages: 784
price: $5
In Cart

The Divine Comedy
by: Dante Alighieri
published: 1308-1321, Italy
language: Italian
pages: 928
price: $5
In Cart

Cart
Things Fall Apart - 5
Fairy tales - 5
The Divine Comedy - 5
Total: $15 USD

Events

SyntheticEvent

- A cross-browser wrapper around the browser's native event
- Hides the implementation details of how React events translate to DOM events
- Contains the same properties as native events, along with some React-specific ones.

Event Listener Attributes

- React's built-in components support HTML-style event listener attributes.
- There are some differences between how to use these in React and how they're used in HTML
- React event listener attributes use camelCase
  - onSubmit
  - onClick
  - onChange

The Event Object

- React events create a SytheticBaseEvent object
- SyntheticBaseEvent has a property called NativeEvent, which contains the properties of the native DOM event.
- You can use SyntheticBaseEvent the same as you the native Event object in JavaScript without React.
- SyntheticBaseEvent is passed to the callback function when an event happens.

Event Handler Functions

- Unlike HTML event listeners, which take a function call, React event handlers take a function name or a function.
- HTML
  - `<button onclick="handleChange()">Click Me</button>`
- React (JSX)
  - `<button onClick={handlechange}>Click Me</button>`
- The function will be called, and the Synthetic Event will be passed to it, when the event happens.

Binding Event Handlers

- When you pass a function from a class component to another component, you need to bind the function to the parent class to provide the correct context for the `this` keyword.
- Only necessary when using method notation or the function keyword in class components.
- Arrow functions are automatically bound to the object where they're created.
- Function components don't have an instance (no `this`), so no binding inner functions.

Passing Data to Event Handlers

- Many times, the event object contains all the data you need for the event handler function, so there's no need to explicitly pass arguments.

```
handleEvent(e){
  this.setState({firstName:e.target.value});
}


<input type="text" onChange={handleEvent} />
```

- If you want to pass data to an event handler, one way is to use an arrow function as the value of the listener.

```
<input type="text"
  onChange={(e)=>updateName(e.target.value)} />
```

Important Event Properties

- Event.cancelable indicates whether an event can be canceled.
- Event.target references the object onto which the event was originally dispatched (such as an element that was clicked or a form input that was typed into).
- Event.type contains the name of the event, such as click, change, load, mouseover, and so forth.
- Event.preventDefault cancels an event if it’s cancelable.

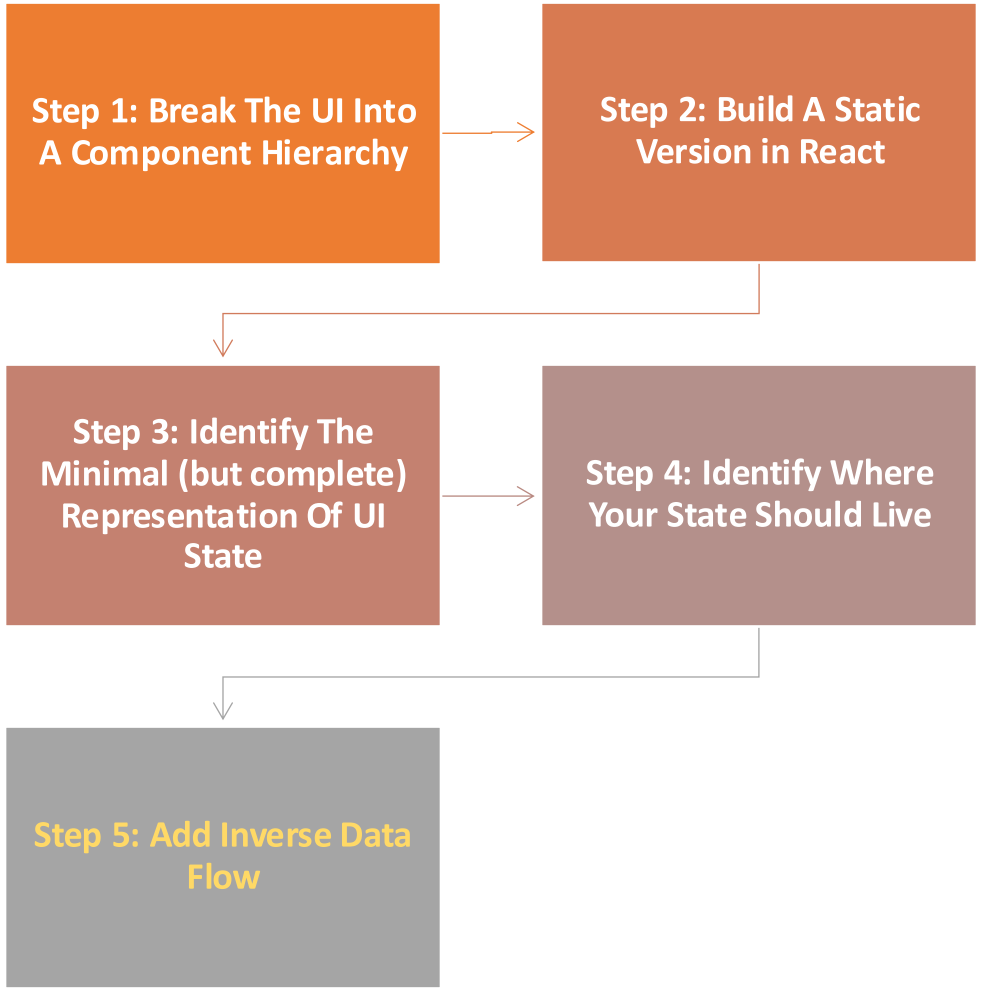
Forms

- Objectives
- Use Controlled Components
- Use Uncontrolled Components

React Development Process

Step 5:

Add Inverse Data Flow


Step 1: Break The UI Into A Component Hierarchy
Step 2: Build A Static Version in React
Step 3: Identify The Minimal (but complete) Representation Of UI State
Step 4: Identify Where Your State Should Live
Step 5: Add Inverse Data Flow

What is "Inverse Data Flow"?

- Use a function to update state in components higher up in the hierarchy.

Forms Have State

- They are different from other native components because they can be mutated based on user interactions.
- Properties of Form components
  - value
    - supported by `<input>` and `<textarea>`
  - checked
    - supported by `<input type="checkbox | radio" />`
  - selected
    - supported by `<option>`
- `<textarea>` should be set with value attribute, rather than children in React

Form Events

- Form components allow listening for changes using onChange
- The onChange prop fires when:
  - The value of `<input>` or `<textarea>` changes.
  - The checked state of `<input>` changes.
  - The selected state of `<option>` changes.

Controlled Components

- A controlled `<input>` is one with a value prop.
- User input has no effect on the rendered element.
- To update the value in response to user input, you can use the onChange event.
- Controlled vs. Uncontrolled Demo:
  - https://codesandbox.io/s/controlled-vs-uncontrolled-0f1esu

Uncontrolled Components

- An `<input>` without a value property is an uncontrolled component.

```
render() {
   return <input type="text" />;
}
```

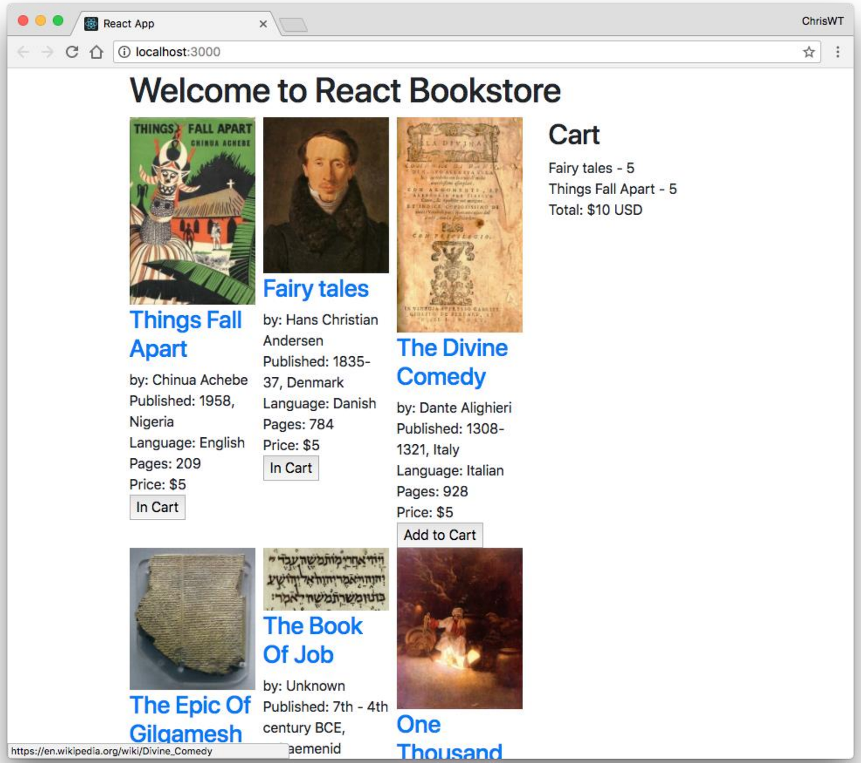
- User input will be reflected immediately by the rendered element.
- Maintains its own internal state

Preventing Default Actions

- Use Event.preventDefault to prevent an element's default action from happening.
- Commonly used with forms to prevent submitting from trying to load a separate page.

```
handleSubmit(e){
  e.preventDefault();
  // do something here
}
```

Lab 09: Interactions, Events, Callbacks



Welcome to React Bookstore

Things Fall Apart
by: Chinua Achebe
Published: 1958, Nigeria
Language: English
Pages: 209
Price: $5
In Cart

Fairy tales
by: Hans Christian Andersen
Published: 1835-37, Denmark
Language: Danish
Pages: 784
Price: $5
In Cart

The Divine Comedy
by: Dante Alighieri
Published: 1308-1321, Italy
Language: Italian
Pages: 928
Price: $5
Add to Cart

Cart
Fairy tales - 5
Things Fall Apart - 5
Total: $10 USD

The Epic Of Gilgamesh

The Book Of Job
by: Unknown
Published: 7th - 4th century BCE,
aemenid

One Thousand

https://en.wikipedia.org/wiki/Divine_Comedy

Component Life-Cycle Events

- Categories
    - Mount
    - Updating
    - Unmounting
    - Error handling

Life-Cycle Methods

- Methods of components that allow you to hook into views when specific conditions happen.

Mount/Unmount

- Mount and unmount methods are called when components are added to the DOM (Mount) and removed from the DOM (Unmount).
- Each is invoked only once in the lifecycle of the component
- Used for:
  - establish default props
  - set initial state
  - make AJAX request to fetch data for component
  - set up listeners
  - remove listeners

Mount/Unmount Lifecycle Methods

- `constructor`
- `static getDerivedStateFromProps`
  - `checks whether props have changed and uses new props to update state`
- `render`
- `componentDidMount`
- `componentWillUnmount`

Updating Lifecycle Methods

- static getDerivedStateFromProps
- shouldComponentUpdate
  - returns a Boolean, which determines whether updating will happen.
- getSnapshotBeforeUpdate
  - use for capturing information about the state before updating
- render
- componentDidUpdate

Error Handling Methods

- getDerivedStateFromError
  - Runs when an error happens in a descendant component.
  - Receives the error that occurred and can return an object for updating the state.
- componentDidCatch
  - Runs after a descendant component throws an error.
  - Useful for logging errors.

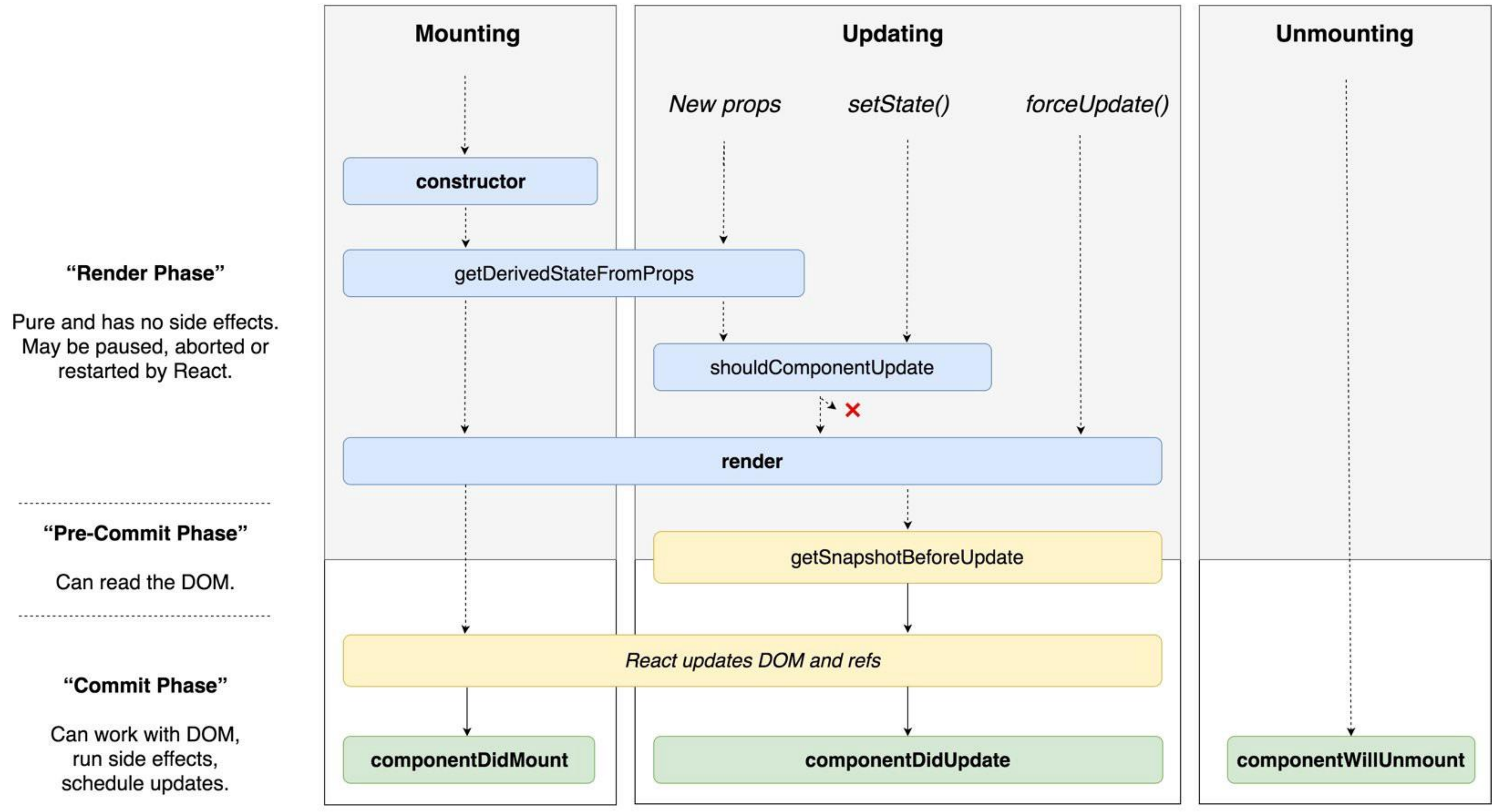
What is an error boundary?

- Error boundaries use the getDerivedStateFromError and/or componentDidCatch lifecycle methods to catch and handle errors in a component’s children.
- Can be used to keep an error in a child component from crashing the entire UI
- Can also be used for logging the error

Implementing Error Boundaries

```
class ErrorBoundary extends React.Component {
  constructor(props) {
  super(props);
  this.state = { hasError: false }; }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }


  componentDidCatch(error, errorInfo) {

    logErrorToMyService(error, errorInfo);

  }

  render() {

    if (this.state.hasError) {

    return <h1>Something went wrong.</h1>;

  }

  return this.props.children; }

}
```

Component Life Cycle

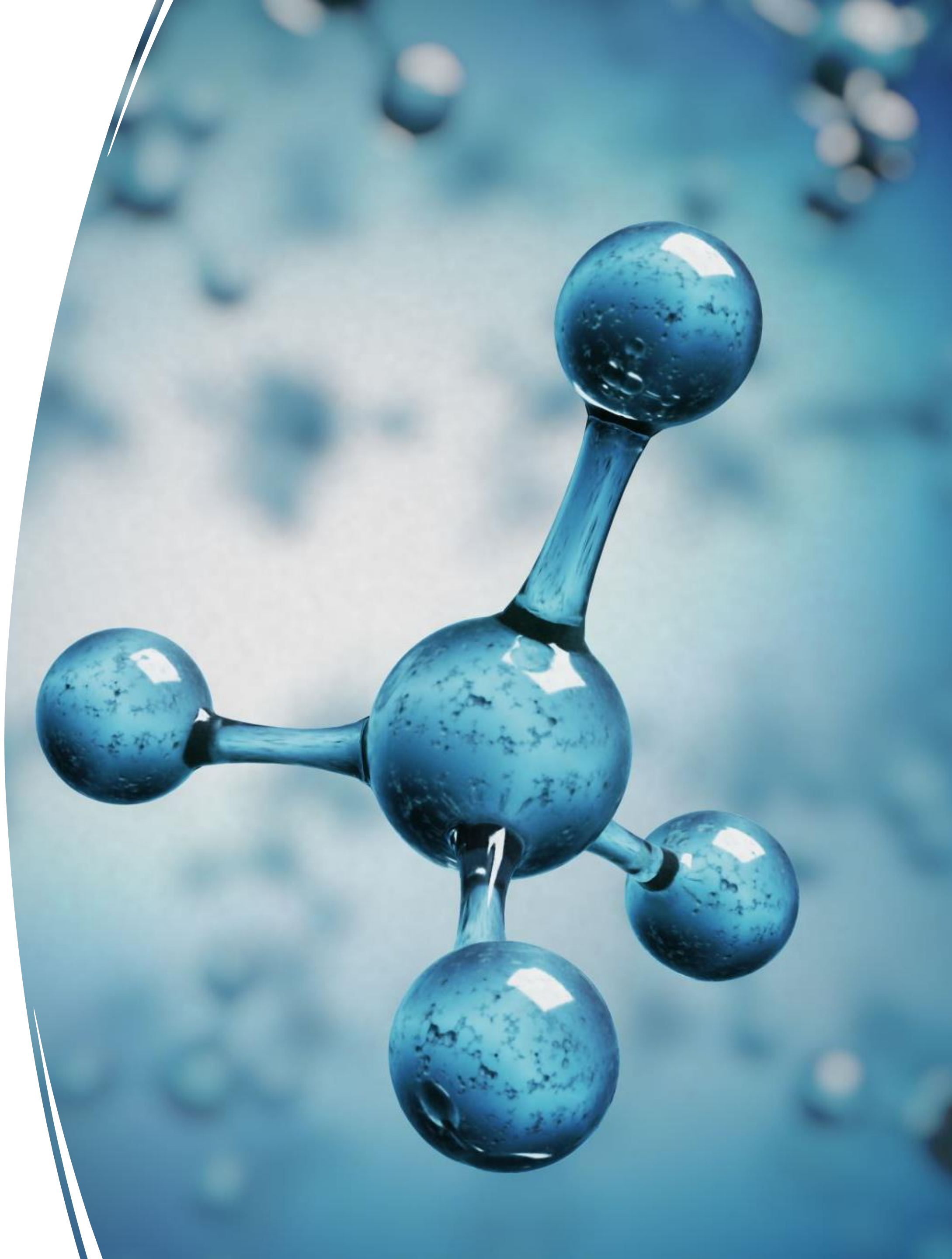


React Components

- Objectives
  - Understanding components and elements
  - Understand how to write components
  - Understand component life-cycle
  - Use events and event handlers
  - Communicate between components

What are components?

- React components define React Elements
- Elements are composed to create UI



Creating Components

- Two techniques
  - Extend React.Component
  - Function Component

Class Definition

- ES6 introduces more OOP-style classes
- Can be created with Class declaration or Class expression

Class Declaration

```
class Square {
  constructor (height, width) {
    this.height = height;
    this.width = width;
    }
}
```

Class Expressions

- Can be unnamed

```
const Square = class {
  constructor(height,width) {
    this.height = height;
    this.width = width;
  }
};
```

- Or named

```
class Square {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Class Inheritance

```
class Rectangle extends Shape {
       constructor (id, x, y, width, height) {
         super (id, x, y);
         this.width = width;
         this.height = height;
       }
}
class Circle extends Shape {
       constructor (id, x, y, radius) {
         super (id, x, y);
         this.radius = radius;
       }
}
```

Private Methods

- Methods of an object can be private to that object.
- Private methods can only be used within the object.
- To define private methods in classes or objects, preface the name of the object with #.

```
class Customer {
  #getFullName() {
    return `${this.fname} ${this.lname}`
  }
  greetCustomer() {
    return `Welcome, ${#getFullName()}`;
  }
}
```

Static methods

- Static methods are methods that are called on the class, rather than on an instance of the class.
- Generally used for utilities.

```
class Customer {
  static getActiveCustomers(){
    // return active customers
  }
}
Customer.getActiveCustomers();
```

Understanding `this`

- Allows functions to be reused with different context.
- Which object should be focal when invoking a function.

4 Rules of `this`

- Implicit Binding
- Explicit Binding
- New Binding
- Window Binding

What is `this`?

- When was function invoked?
- We don't know what `this` is until the function is invoked.

Implicit Binding

- `this` refers to the object to the left of the dot.

```
const author = {
  name: 'Chris',
  homeTown: 'Detroit',
  logName: function() {
    console.log(this.name);
  }
});

author.logName();
```

Explicit Binding

- .call, .apply, .bind

```
let logName = function() {
  console.log(this.name);
}

const author = {
  name: 'Chris',
  homeTown: 'Detroit"
}
logName.call(author);
```

Explicit Binding with .call

- Calls a function with a given `this` value and the arguments given individually.

```
let logName = function(lang1) {
  console.log(this.fname + this.lname + lang1);
};
let author = {
  fname: "Chris",
  lname: "Minnick"
};
let language = “JavaScript”;
logName.call(author,language);
```

Explicit binding with .apply

- Calls a function with a given `this` value and the arguments given as an array

```
logName = function(food1,food2,food3) {
  console.log(this.fname + this.lname);
  console.log(food1, food2,
    food3);
};
let author = {
  fname: "Chris",
  lname: "Minnick"
};

let favoriteFoods= ['Tacos','Soup','Sushi'];

logName.apply(author, favoriteFoods);
```

Explicit Binding with .bind

- Works the same as .call, but returns `new` function rather than immediately invoking the function

```
let logName = function(food) {
  console.log(this.fname +" " + this.lname +
              "\'s Favorite Food was " + food);
};
const person = {
  fname: "George",
  lname: "Washington"
};
let logMe = logName.bind(person, "tacos");

logMe();
```

- http://jsbin.com/xikuzog/edit?js,console

`new` Binding

- When a function is invoked with the `new` keyword, then this keyword inside the object is bound to the new object.

```
const City = function (lat,long,state,pop) {
  this.lat = lat;
  this.long = long;
  this.state = state;
  this.pop = pop;
};
let sacramento = new City(38.58,121.49,"CA",480000);
console.log (sacramento.state);
```

`window` Binding

- What happens when no object is specified or implied
- `this` defaults to the `window` object

```
let logName = function() {
  console.log(this.author);
}
let author = {
  name: 'Chris',
  homeTown: 'Detroit"
}
logName(); //undefined(error in 'strict' mode)
window.author = "Harry";
logName(); // "Harry"
```

React.Component

- Base class for React Components when defined using JavaScript classes.

```
class HelloMessage extends React.Component {
  constructor(props){
    super(props);
  }
  render() {
    return (<div>Hello {this.props.name}</div>);
  }
}
```

super()

- Used for calling the parent’s class’s constructor.
- JavaScript classes must call `super()` if they are subclasses.
- If you don't have a constructor, you don't need to call `super()`.
- If you want to use `props` in your constructor, you need to call `super(props)` in the constructor.

Classes Can be Confusing...

- When to use a constructor?
- What does super() do?
- When do you bind function?
- Where should the state live?

The Solution: Function Components

- Function components gave developers a way to create “stateless” components using JavaScript functions

```
function MyComponent(props){
  return (
   <p>Welcome to my component, {props. name}</p>
  );
}
```

- Or using arrow functions:

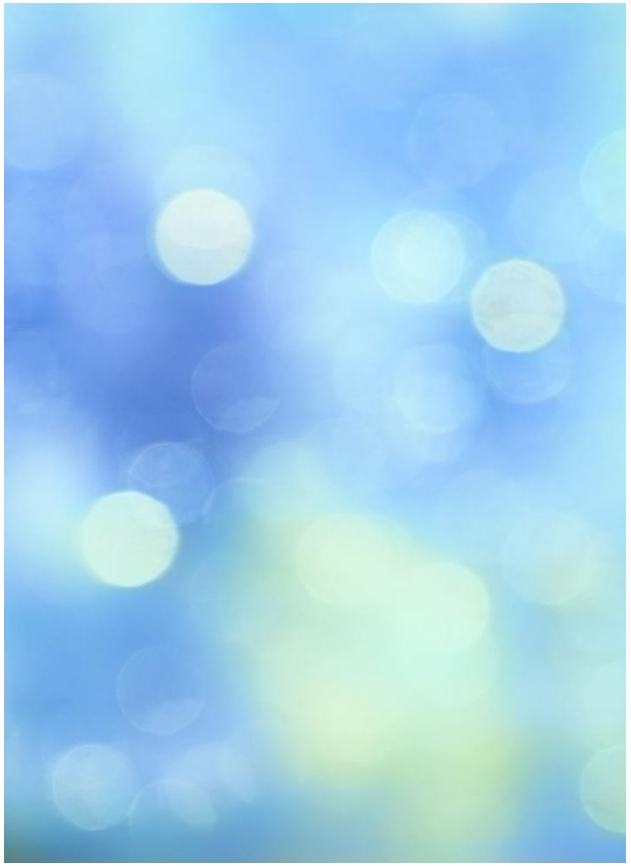
```
const MyComponent = (props)=>{

    <p>Welcome to my component, {props.name}</p>

 }
```

Component Children

Components used inside components are called Child Components

A Component that contains other components is called a Parent Components

Components can be both parents and children, depending on how their used.



Promises

What Are Promises?

- An abstraction for asynchronous programming
- Alternative to callbacks
- A promise represents the result of an async operation
- Is in one of three states
  - pending – the initial state of a promise
  - fulfilled – represents a successful operation
  - rejected – represents a failed operation

Promises vs. Event Listeners

- Event listeners are useful for things that can happen multiple times to a single object.
- A promise can only succeed or fail once.
- If a promise has succeeded or failed, you can react to it at any time.

```
readJSON(filename).then(success,failure);
```

Why Use Promises?

- Chain them together to transform values or run additional async actions
- Cleaner code
    - Avoid problems associated with multiple callbacks
        - Callback Hell
        - Christmas Tree
        - Tower of Babel
        - Etc.

Demo: Callback vs. Promise

- Callback

  -
    ```
    fs.readFile('text.txt', function(err,
    file){
            if (err){
                    //handle error
            } else {
                    console.log(file.toString());
            }
    });
    ```
- Promise

  -
    ```
    readText('text.txt')
            .then(function(data) {
                        console.log(data.toString());
                    }, console.error
            );
    ```

Using Promises

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


function readFileAsync (file, encoding) {
    return new Promise(function (resolve, reject) {
        fs.readFile(file, encoding, function (err, data) {
            if (err) return reject(err);
            resolve(data);
        })
    })
}


readFileAsync('myfile.txt')

       .then(console.log, console.error);
```

Async / Await

- Simplifies using promises.

```
async function f() {
    return 1;
}
```

- An async function always returns a promise.

```
f().then(alert);
```

- You can use the await keyword inside an async function to wait until the promise resolves.

```
async function getUser() {
    let resp = await fetch('http://url.com/users/’);
    let resp2 = await …

    return resp;
}
```

AJAX

- Fetch vs. Axios
- AJAX in class components
- AJAX in function components
- Four Ways
  - Root Component
    - *Best for small apps and prototypes.*
  - Container Components
    - *Create a container component for every presentational component that needs data from the server.*
  - Redux Thunk
    - *Lab: Thunk*
  - Redux Saga
  - Suspense & Async Rendering
- Demo: Authentication with React and JWT

AJAX in Class Components

- Use componentDidMount lifecycle method

```
componentDidMount() {
axios.get(`http://www.reddit.com/r/${this.
props.subreddit}.json`)
  .then(res => { const posts =
    res.data.data.children.map(obj =>
obj.data);
    this.setState({ posts });
  });
}
```

AJAX in Function Components

- use useEffect hook

```
useEffect(() => {
 async function fetchData() {
 const response = await fetch(url);
 const json = await response.json();
 setData(json);
}
fetchData();
}, [url]);
```

Fetch vs. Axios

- Fetch is built-in to browsers
- Axios is a separate library
- Axios converts JSON data automatically
- The response from Fetch must be converted to an object using response.json()

Suspense

- Lets your components wait for something (such as data fetching).

```
<Suspense fallback={<h1>Loading profile...</h1>}>
  <ProfileDetails />
</Suspense>
```

- New in React 18

Cross-origin Resource Sharing (CORS)

- Browsers block JavaScript from accessing data from API endpoints on different domains.
- Specifically, CORS is triggered for HTTP requests from JavaScript to:
  - different domain (example.com calling api.com)
  - different subdomain (example.com calling api.example.com)
  - different port (example.com calling example.com:3000)
  - different protocol (https://example.com calling http://example.com)

CORS: Simple Request

- Simple request
  - GET or POST
- Non-simple request:
  - Content-Type other than application/x-ww-form-urlencoded, multipart/form-data, or text-plain or a request with cookies.
  - Server returns Access-Control-Allow-Origin header, which is checked by the browser.

Access-Control-Allow-Origin

- Header returned by HTTP server
- Can allow any origin:
  - Access-Control-Allow-Origin: *
- Or specific origins:
  - Access-Control-Allow-Origin: https://example.com
- Example (using Node / express):

```
const cors = require('cors');
app.use(cors({
  origin: 'https://www.example.com'
}));
```

CORS: Non-simple request

- If the request is non-simple, the browser will make a preflight request.
- Preflight request uses the OPTIONS method to determine if the non-simple request can be made.
- Example:

```
curl -i -X OPTIONS localhost:3001/api/ping
\ -H 'Access-Control-Request-Method: GET'
\ -H 'Access-Control-Request-Headers:
Content-Type, Accept'
\ -H 'Origin: http://localhost:3000'
```

Preflight Response

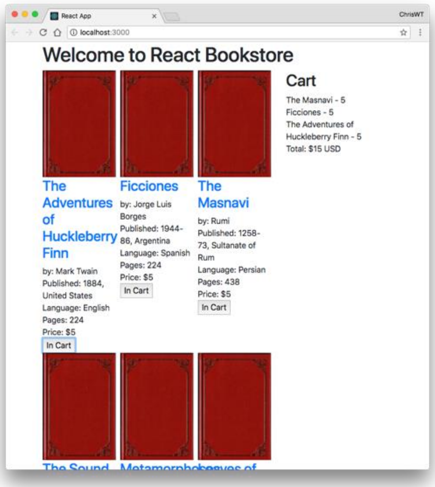
- Server responds to Preflight request with headers.
- Example:

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods:
GET,HEAD,PUT,PATCH,POST,DELETE
Vary: Access-Control-Request-Headers
Access-Control-Allow-Headers: Content-Type,
Accept
Content-Length: 0
Date: Fri, 05 Apr 2019 11:41:08 GMT
Connection: keep-alive
```

Preflight (cont.)

- Browser examines headers returned by server to determine whether to make the non-simple request.

Lab 10: Component Life-Cycle and AJAX



Welcome to React Bookstore

The Adventures of Huckleberry Finn
by: Mark Twain
Published: 1884, United States
Language: English
Pages: 224
Price: $5
In Cart

Ficciones
by: Jorge Luis Borges
Published: 1944-86, Argentina
Language: Spanish
Pages: 224
Price: $5
In Cart

The Masnavi
by: Rumi
Published: 1258-73, Sultanate of Rum
Language: Persian
Pages: 438
Price: $5
In Cart

Cart
The Masnavi - 5
Ficciones - 5
The Adventures of Huckleberry Finn - 5
Total: $15 USD

The Sound
Metamorpho
Leaves of

Hooks

The Problem: Functions Aren’t Classes

Functional Components:

- can’t have state.
- can’t use lifecycle methods.
- can’t update state

The Solution

- Hooks!

What are Hooks?

- Hooks provide access to imperative ways of working in React
- Completely opt-in
- Backwards compatible
- Reduces complexity
- Provides a more direct API to props, state, context, refs, and lifecycle
- Provides a standard way to reuse behavior between components
- Hooks let you use more of React's features without Classes

But What are Hooks?

- Hooks are functions that let you hook into React state and lifecycle features from function components.
- Hooks allow you to reuse stateful logic without changing your component hierarchy.

Built-in Hooks

- React has several “built-in” hooks.
- useState
  - Takes an initial value and returns a state variable and a method for updating that variable.
- useEffect
  - Adds the ability to perform side effects from a function.
  - Serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount.
  - Lets you call a function when any of these lifecycle events happen.

Rules of Hooks

- Must be called at the top level
  - Not inside loops, conditions, or nested functions.
- Can only be called from function components.
- Hooks start with “use” by convention.

useState

```
import { useState } from 'react';
function Example() {
 const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useState can be used multiple times

```
function ManyStates() {
  const [animal, setAnimal] = useState(‘monkey’);
  const [name, setName] = useState(‘Chris’);
  const [todos, setTodos] = useState([{ text: ‘eat 
lunch’}]);
  // ..
}
```

useEffect

- Adds the ability to perform side effects from a function component.
- Serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount
- By default, React runs effects after every render, including the first.

useEffect example

```
function LifecycleDemo() {
  useEffect(()=> {
    console.log('render!');
    //optional "cleanup" return statement to run before render
or unmount
    return () =>
      console.log('unmounting');
  })

  return “This is a lifecycle demo”;
}
```

When does useEffect run?

- By default: before every render.
- Want to run it less often?
  - Provide a 2nd argument
    - Array
    - If one of them has changed, the effect will run again.
    - Will still run after the initial render.
  - To run only on Mount and Unmount, pass an empty array ([])
    - Simulates componentDidMount lifecycle method

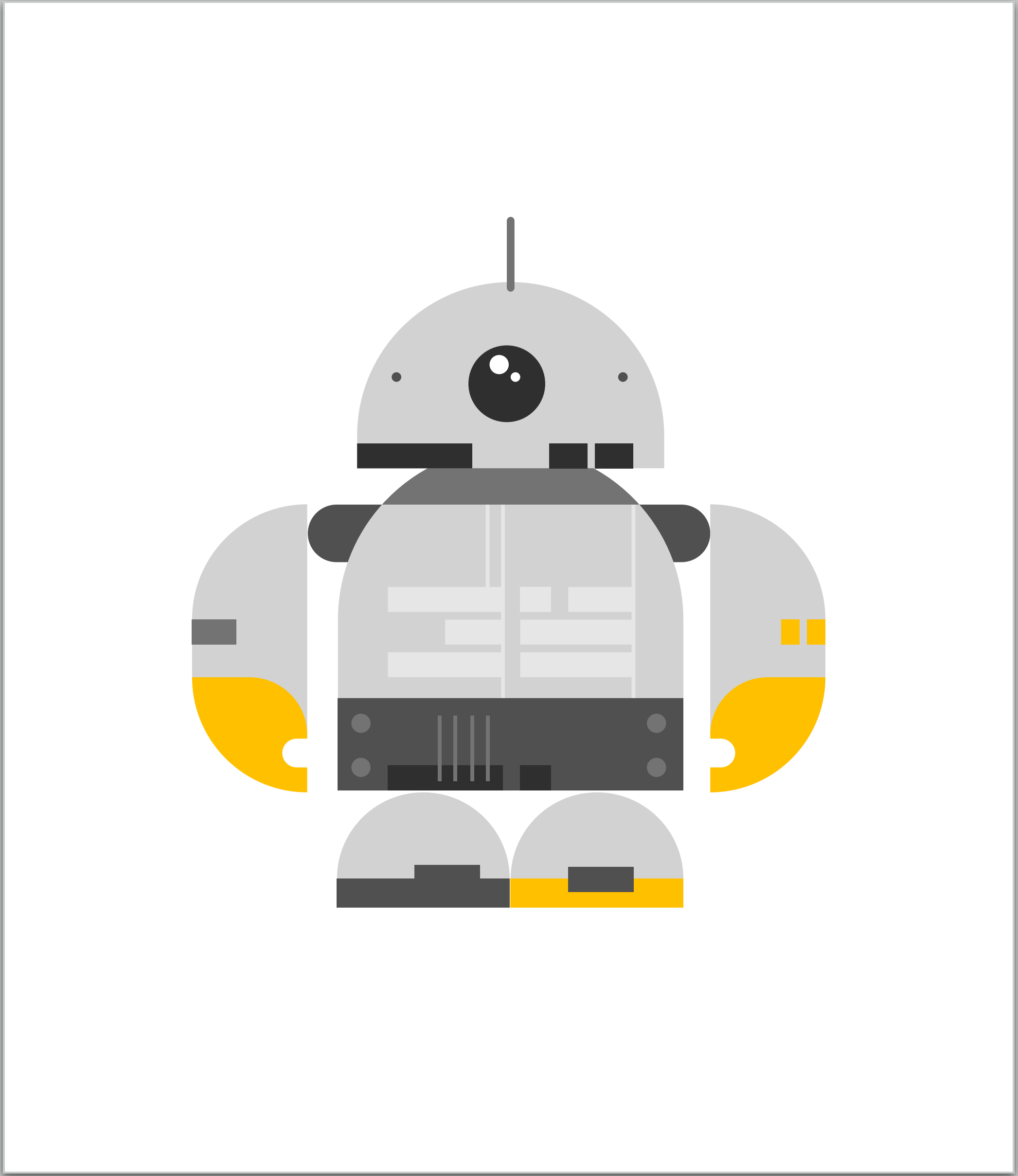
Uses for useEffect

- Fetch initial data
- focus inputs on first render
- Re-fetching when data changes

More Built-in Hooks

- useContext
- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHandle
- useLayoutEffect
- useDebugValue

Lab 11: Converting a Class Component to a Function Component



Context and Refs

Context API

- Context allows parents to pass data implicitly to children, no matter how deep the component tree is.

Creating and Using Context for Color Themes

- Define a ColorContext to manage color theme state.
- Use Context.Provider to wrap components needing theme access.
- Utilize useContext hook for easy access to the theme.

Creating a Context

```
import React, { createContext, useState, useContext } from 'react';

// Create a context with a default value
const ColorContext = createContext();
export const ColorProvider = ({ children }) => {
  const [color, setColor] = useState('light'); // Default theme is 'light'
  const toggleColor = () => {
    setColor((prevColor) => (prevColor === 'light' ? 'dark' : 'light'));
  };
  return (
    <ColorContext.Provider value={{ color, toggleColor }}>
      {children}
    </ColorContext.Provider>
  );
};


// Custom hook to use the ColorContext
export const useColor = () => {
  return useContext(ColorContext);
};
```

Lab 12:

Creating a Color Theme Context

Refs

- Gives React access to a DOM node or an instance of a component.

Creating a ref in a Class Component

```
class TextReader extends Component {

constructor(props) {
  super(props);
  this.textView = React.createRef();
}
  render() {
    return (
      <textarea ref={this.textView}
                value={this.props.bookText} />
    );
  }

}

export default TextReader;
```

Creating a ref in a Function Component

```
import {useRef} from 'react';

function TextReader(props) {

  const textView = useRef(null);

    return (
      <textarea ref={textView}
                value={props.bookText} />
    );
}

export default TextReader;
```

Using refs

```
class TextReader extends Component {

  constructor(props) {
    super(props);
    this.textView = React.createRef();
  }

  componentDidMount(){
    this.textView.current.focus();
  }


  render(){

    return (
      <textarea style={{width:'380px',height:'400px'}}
                ref={this.textView}
                value={this.props.bookText} />
    );
  }
}

export default TextReader;
```

When to Use Refs

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

Custom Hooks

- A JavaScript function whose name starts with "use" and that may call other Hooks.
- Lets you extract component logic into reusable functions.
- Isn't a feature of React, but is enabled by Hooks.

Why Use Custom Hooks

- Share logic between components.
- Hide complex logic behind a simple interface.

Custom Hooks Examples - useFetch

- data fetching Hook
- Several different versions currently exist

```
const { data, loading, setUrl } =
useFetch('https://your.api.url/search/');
```

Making Custom Hooks

- Make a function with a name starting with 'use'
- Use other hooks inside it
- Custom hooks can be reused in multiple components and each instance will have its own isolated state.

Lab 13: Custom Hooks

React with TypeScript

TypeScript Basics

- JavaScript with Syntax for Types
- Strongly typed language
- Converts to JavaScript

Why TypeScript?

- Static type checking
- Better tooling
- Better documentation
- More confidence in code
- Can be applied gradually

JS vs. TS

JS

```
function greet(name) {
  return `Hello, ${name}!`;
}

greet(17);

"Hello 17!"
```

TS

```
function greet(name:string) {
  return `Hello, ${name}!`;
}

greet(17);
```

Argument of type 'number' is not assignable to parameter of type 'string'.

Type inference

- When you declare a variable using var, let, or const, you can optionally add a type annotation
  - let myName: string = "Chris";
- However, this isn't necessary most times. TypeScript will infer the type from the type of the initial value.
  - let myName = "Chris"; // myName is inferred as 'string'

Type annotations in functions

- Parameters with type annotations will be checked.
- Return values can be annotated too, but it's not usually necessary because TypeScript can infer it.

Annotating Objects

- List properties and their types.

```
function doSomething(options: {speed: number,
priority: number, comments?: string})
```

- The ? indicates that a property is optional.
- If you don't specify a type, it will be assumed to be 'any',

Union Types

- Types formed from two or more types.

```
function (loginId: number | string)
{
    ...
}
```

- loginID can be either a number or a string

Type aliases

- A name for a type
- Allows reuse of types

```
type Coords = {
   x: number,
   y: number
}
```

Interfaces

- Another way to name a type.

```
interface Coords {
   x: number,
   y: number
}
```

- Interfaces can be extended, unlike aliases.

Extending Interfaces

```
Interface Person {
  name: string
}

Interface Customer extends Person {
  creditLimit: number
}
```

Type Assertions

- Specifies a more specific type than TS can know about.

```
const myCanvas =
document.getElementById("main_canvas") as
HTMLCanvasElement;
```

Literal Types

- Sets a type to a specific value.
  - let x: "hello" = "hello";
- Is useful when used with unions to specify possible values for a variable.
  - function printText(s: string, alignment: "left" | "right" | "center") {

Generics

- Allow type specification to be deferred.

```
function arrayFilled<T>(item1: T, item2: T): T[] {
  let list = new Array<T>();

  list.push(item1);
  list.push(item2);

  return list;
}

let myArray = arrayFilled<number>(1,2);
```

React TypeScript Cheatsheet

- https://github.com/typescript-cheatsheets/react#reacttypescript-cheatsheets
- Really handy copy / paste examples

File Naming

- For files containing JSX:

filename.tsx

- For other TS files:

filename.ts

Lab 14: Converting to TypeScript

React Patterns and Best Practices

F.I.R.S.T.

- React Components should be:
  - **F**ocused
  - **I**ndependent
  - **R**eusable
  - **S**mall
  - **T**estable

Single Responsibility

- A component should only do one thing.
- If it ends up growing, it should be decomposed into smaller subcomponents.
- A responsibility is a "reason to change."
- Single responsibility makes components more robust.

"A class should have only one reason to change."

-*Robert C. Martin*

Pure Functions

Pure functions always return the same result given the same arguments.

Pure function's execution doesn't depend on the state of the application.

Pure functions don't modify the variables outside of their scope (no side effects).

Benefits of Pure Functions

- Easy to test
- Easy to reason about
- Easy to reuse
- Easy to reproduce the results

Function Comparison

slice()

```
let toppings =
['cheese','pepperoni','mushrooms'];


toppings.slice(0,2);
// ["cheese", "pepperoni"]
toppings.slice(0,2);
// ["cheese", "pepperoni"]
toppings.slice(0,2);
// ["cheese", "pepperoni"]
```

- Always returns the same result given the same arguments
- Doesn't depend on the state of the application
- Doesn't modify variables outside its scope
- It's a Pure Function!

splice()

```
let toppings =
['cheese','pepperoni','mushrooms'];


toppings.splice(0,2);
// ["cheese", "pepperoni"]
toppings.splice(0,2);
// ["mushrooms"]
toppings.splice(0,2);
// []
```

- Not Pure!

React.PureComponent

- If your Component returns the same result given the same props and state, use React.PureComponent
- PureComponent does a shallow state and prop comparison and doesn't update if the component is unchanged.
- May give a performance boost

```
class MyComponent extends React.PureComponent {
  ...
}
```

React.memo

- Works the same as React.PureComponent, but for functional components.

```
const MyComponent = React.memo(
   function MyComponent(props) {
         ...
   });
```

Composition

- Composition is combining smaller components to form a larger whole.

Container (aka Page) Components

- Wrap presentational components
- Contain the logic and state
- Correspond to pages
- Examples:
  - HomePage
  - ServicesPage

Presentational Components

- Components that just output presentation
- Contain no logic
- Responsible for generating UI elements
- Display data passed by Pages

Folder Structure

- Organize projects by Feature
- Keep Pages and Presentational components separate

```
- /pages
  - /pages/App.jsx
  - /pages/ShoppingCart.jsx
- /components
  - /components/Header.jsx
  - /components/ProductInfo.jsx
- /services
-   /services/api.js
- /context
- /index.jsx
```

Higher Order Functions

- A **function** that can take another **function** as an argument and/or that returns a **function** as a result.

```
const multiplyBy = (multiplier) => (number) => number * 
multiplier

const double = multiplyBy(2); // returns (number) => number * 2

double(10) // returns 20
```

Higher Order Components

- A **function** that takes (wraps) a component and returns a new component.
- Allow us to abstract over actions, not just values.

```
const EnhancedComponent =
higherOrderComponent(WrappedComponent);
```

Reusable Components

- Break down the common design elements (buttons, form fields, layout components, etc.) into reusable components with well-defined interfaces.
- The next time you need to build some UI, you can write much less code.
- This means faster development time, fewer bugs, and fewer bytes down the wire.

TDD and Testing React

- Objectives
- Learn the TDD Steps
- Write Assertions
- Understand exception handling in JS
- Create tests with Jest
- Automate cross-browser testing

Goal of TDD

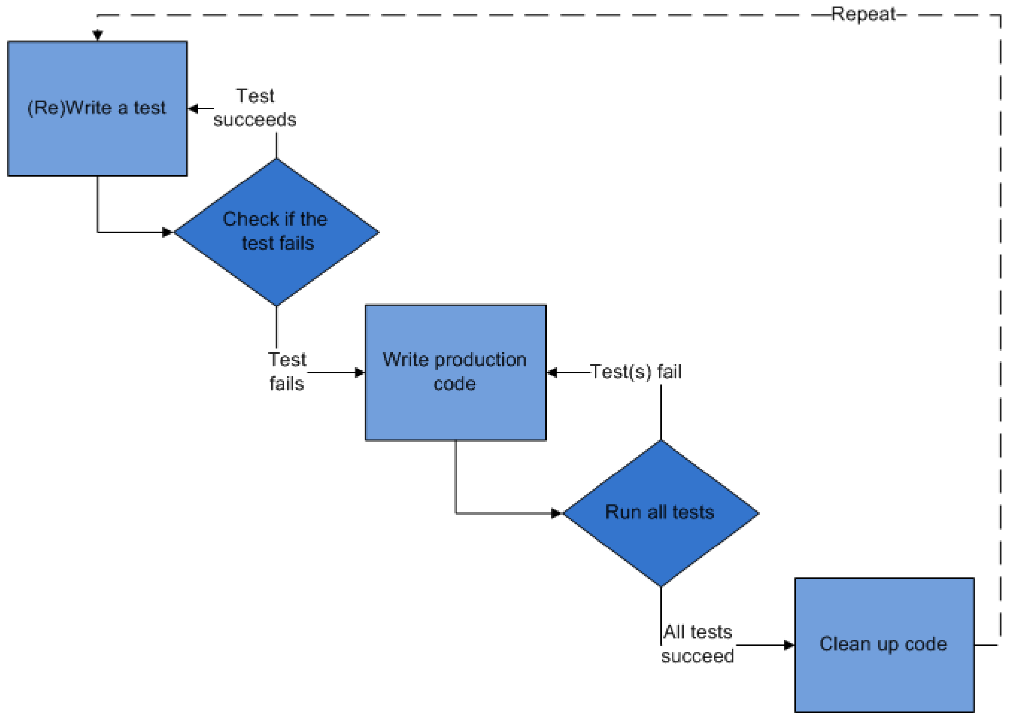
- Clean code that works.

The TDD Cycle

- Red
  - write a little test that doesn't work.
- Green
  - make the test work, as quickly as possible.
  - don't worry about doing it right.
- Refactor
  - eliminate duplication created in making the test work.

TDD Steps

- Write a test
- Check that test fails
- Write code
- Run test - passes!
- Refactor
- Repeat



Red

- Write the story.
- Invent the interface you wish you had.
- Characteristics of a good tests:
  - Each test should be independent of the others.
  - Any behavior should be specified in only one test.
  - No unnecessary assertions
  - Test only one code unit at a time
  - Avoid unnecessary preconditions

Green

- Get the test to pass as quickly as possible.
- Three strategies:
  - Fake it
    - Do something, no matter how bad, to get the test to pass.
  - Use an obvious clean solution.
    - But don't try too hard!
  - Triangulation
    - only generalize code when you have two examples or more.
    - When the 2nd example demands a more general solution, then and only then do you generalize.

Refactor

- Make it right.
- Remove duplication.
- Improve the test.
- Repeat.
- Add ideas or things that aren't immediately needed to a todo list.

Assertions

- Expression that encapsulates testable logic
- Assertion Libraries
  - Chai, should.js, expect.js, better.assert
- Examples
  - `expect(buttonText).toEqual('Go!'); // jasmine`
  - `result.body.should.be.a('array'); // chai`
  - `equal($("h1").text(), "hello"); //QUnit`
  - `assert.deepEqual(obj1, obj2); //Assert`

JavaScript Testing Frameworks

- Jasmine
- Mocha
  - doesn't include its own assertion library
- QUnit
  - from JQuery
- js-test-driver
- YUI Test
- Sinon.JS
- Jest
- Vitest

JS Exception Handling

```
function hello(name) {
  return "Hello, " + name;
}
let result = hello("World");
let expected = "Hello, World!";
try {
  if (result !== expected) throw new Error
     ("Expected " + expected + " but got " +
       result);
} catch (err) {
      console.log(err);
}
```

Vitest Overview

Objectives

- Write test suites
- Create specs
- Set expectations
- Use matchers

How Vitest Works

- Suites describe your tests
- Specs contain assertions

```
describe("This is a suite", function() {
  let a;

  test("A spec test assertions", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Test Suites

- Created using the `describe` function
- Contain one or more specs
- 2 params
  - Text description
  - Function

```
describe("Hello", function() {
    ...
}
```

Specs

- Created using the `it` or `test` function
    - they're the same thing
- Contains one or more expectations
- expectations === assertions

```
describe("Hello", function() {

      it("Concats Hello and a name", function() {
            let expected = "Hello, World!";
            let actual = hello("World");
            expect(actual).toEqual(expected);
      });
});
```

Expectations

- AKA assertions
- Made using `expect` function,
    - Takes a value
- Chained to a Matcher
    - Takes the expected value

```
expect(actual).toEqual(expected);
```

Matchers

- `expect(fn).toThrow(e);`
- `expect(instance).toBe(instance);`
- `expect(mixed).toBeDefined();`
- `expect(mixed).toBeFalsy();`
- `expect(number).toBeGreaterThan(number);`
- `expect(number).toBeLessThan(number);`
- `expect(mixed).toBeNull();`
- `expect(mixed).toBeTruthy();`
- `expect(mixed).toBeUndefined();`
- `expect(array).toContain(member);`
- `expect(string).toContain(substring);`
- `expect(mixed).toEqual(mixed);`
- `expect(mixed).toMatch(pattern);`

Testing React Components

- Objectives
- Learn about different rendering modes
- Learn about Jest
- Write Unit Tests with Jest

What to Test in a React Component

- Does it render?
- Does it render correctly?
- Test every possible state / condition
- Test the events
- Test the edge cases

Vitest

- Runs any tests in __tests__ directories, or named .spec.js, or named .test.js
- Simulates browser environment with jsdom

Mocking

- Mock Function - erase the implementation of a function – vi.fn()
- Manual Mocking - stub out functionality with mock data
- Timer Mocking - swap out native timer functions

Mock Function

- const mockCallback = vi.fn();
- forEach([0, 1], mockCallback);

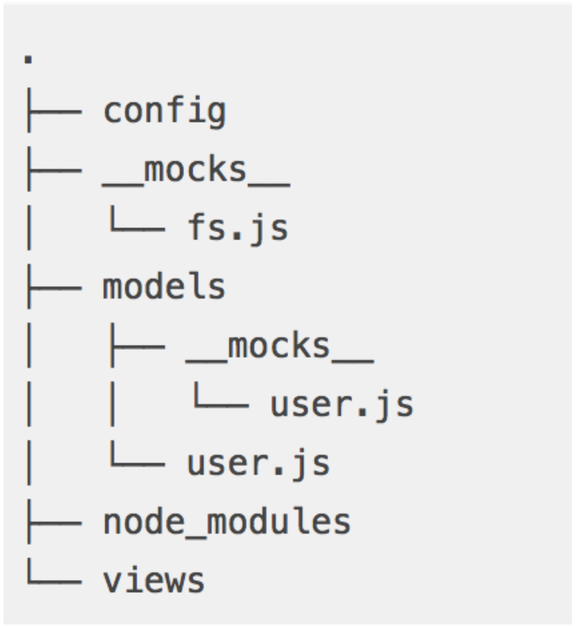
- // The mock function is called twice
- expect(mockCallback.mock.calls.length).toBe(2);

- // The first argument of the first call to the function was 0
- expect(mockCallback.mock.calls[0][0]).toBe(0);

- // The first argument of the second call to the function was 1
- expect(mockCallback.mock.calls[1][0]).toBe(1);

Manual Mock

- Ensures tests will be fast and reliable by mocking external data and core modules.
- Define in a __mocks__ subdirectory adjacent to the module



```
.
├── config
├── __mocks__
│   └── fs.js
├── models
│   ├── __mocks__
│   │   └── user.js
│   └── user.js
├── node_modules
└── views
```

Snapshot Testing

1. Renders a component
2. Creates a "snapshot file" on first run
3. Compares subsequent runs with first and fails test if different.

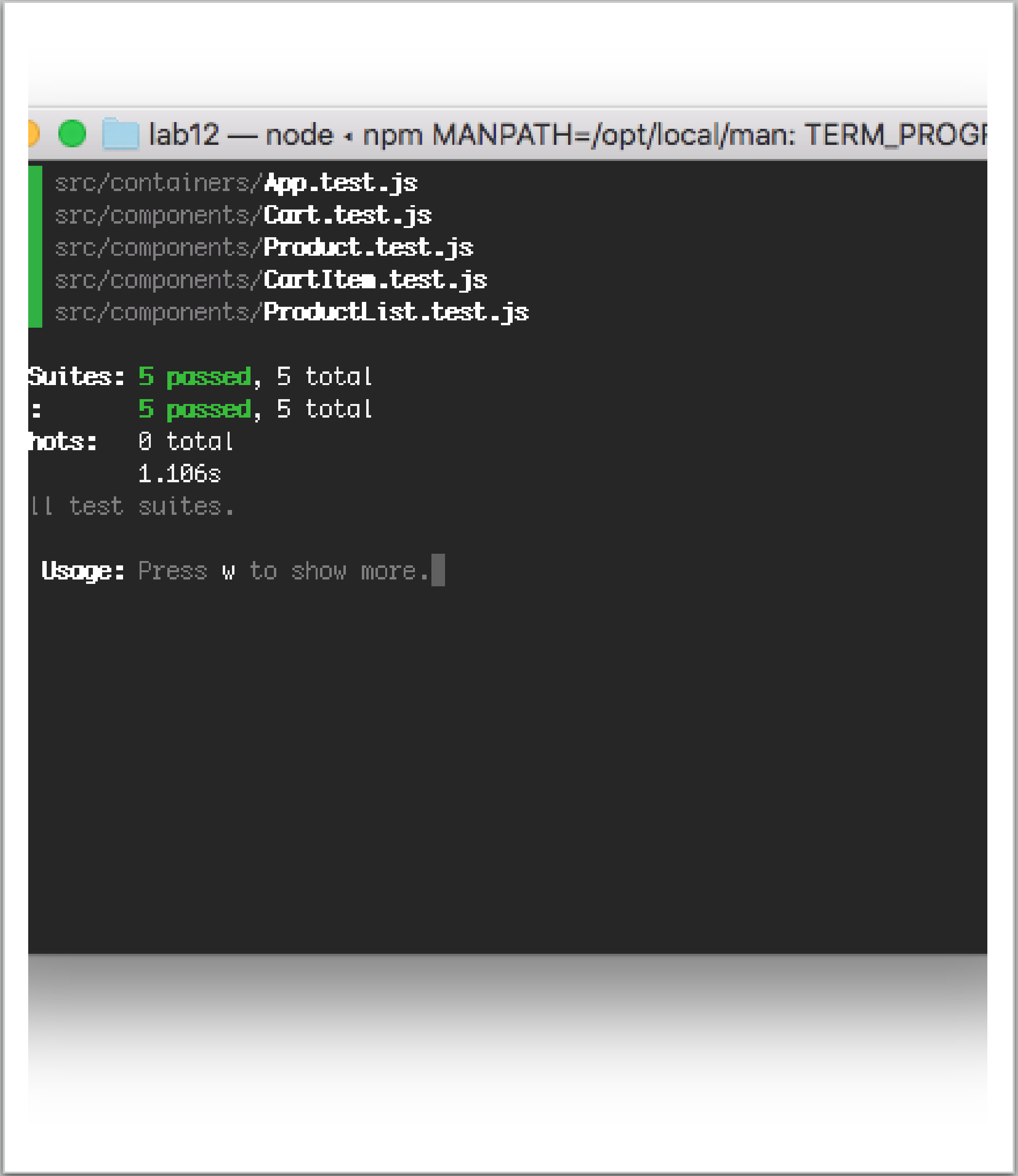
Sample Snapshot Test

```
import React from 'react';
import App from './App';

it('renders as expected', () => {
const { container } = render(<App />);
expect(container).toMatchSnapshot();
});
```

React-Testing-Library

- `const container = render(<App />);`

Lab 15: Writing Tests with Vitest and RTL



lab12 — node ◂ npm MANPATH=/opt/local/man: TERM_PROGF

```
 src/containers/App.test.js
 src/components/Cart.test.js
 src/components/Product.test.js
 src/components/CartItem.test.js
 src/components/ProductList.test.js

Suites: 5 passed, 5 total
:       5 passed, 5 total
hots:   0 total
        1.106s
ll test suites.

 Usage: Press w to show more.
```

Flux and Redux

- Objectives
  - Understand the Flux pattern
  - Explain Redux's architecture
  - Create Redux actions
  - Write pure functions
  - Use Reducers
  - Use Redux with AJAX

Flux

- Flux isn't a library or module.
- It's a design pattern.
- npm install flux installs Facebook's dispatcher.
- It's possible to use Flux design principles without Facebook's module.

Flux Flow

1. Some sort of interaction happens in the view.
2. This creates an action, which the dispatcher dispatches.
3. Stores react to the dispatched action if they're interested, updating their internal state.
4. Stateful view component(s) hear the change event of stores they're listening to.
5. Stateful view component(s) ask the stores for new data, calling setState with the new data.

Flux Action

- An action in flux is what's made when something happens.
- In other words, when you click on something, that's not an action
  - it creates an action. Your click is an interaction.
- Actions should have (but aren't required to have) a type and a payload. Most of the time they will have both, occasionally they'll just have a type.

Flux Dispatcher

- Broadcasts actions when they happen and it lets things tune in to those broadcasts
- Instead of an onClick function using a callback passed to it to set the state of your application, you have it (onClick) use the dispatcher to dispatch a specific action for anyone who's interested to listen for.

Flux Stores

- Represents the ideal state of your application
- If a user enters something into a form, it dispatches an action. If the store is listening for this action, it will update its internal state accordingly.
- Stores don't contain any public setters, just public getters. The only ones who can change the data in a store is the store itself when it hears an action from the dispatcher that it's interested in.

EventEmitter

- Stores emit change events that don't contain data.
- If the view is listening for the particular store's change event, the view should ask the store for the new data that will bring the view back into sync, call setState, and re-render.

Redux

- An implementation of Flux
- Stores state of the app in an object tree in a single store
- The state tree can only be changed by emitting an action
- Specify how the actions transform the state tree using pure reducers

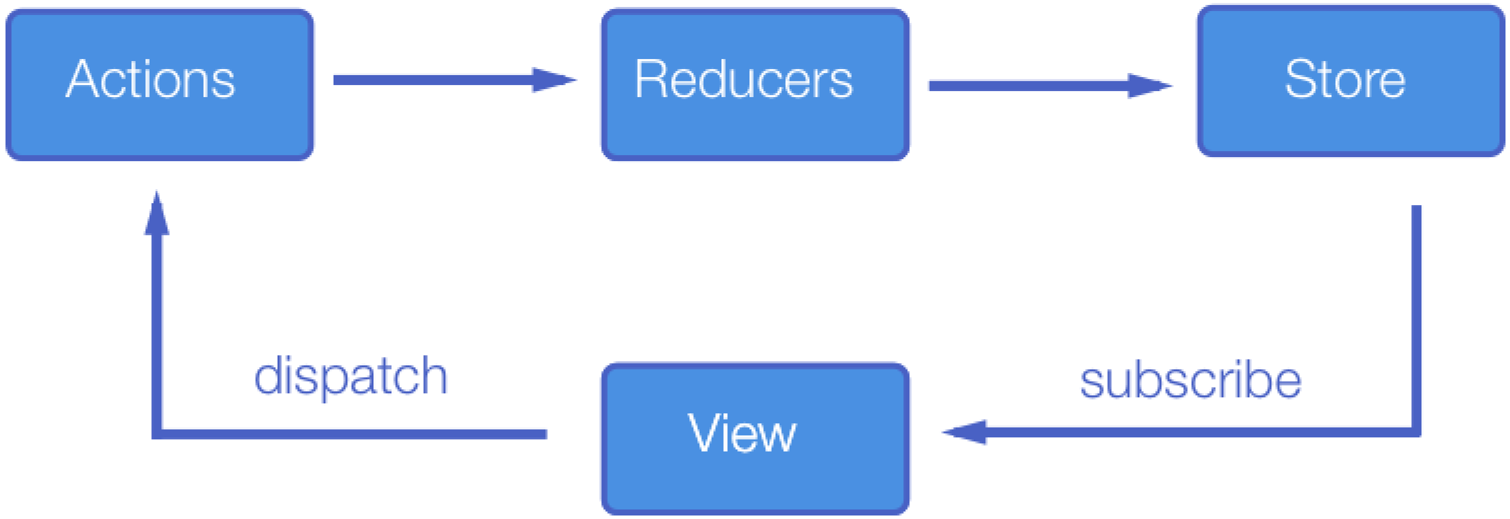
Stores & Immutable State Tree

- The biggest difference between Flux and Redux is that Redux has a single store.
- Redux has a single store with a single root reducing function.
- Split the root reducer into smaller reducers to grow the app.
- Trace mutations by replaying actions that cause them.

Redux Actions

- Payloads of information that send data from the application to the store.
- Actions are the only way to mutate the internal state.
- Use `store.dispatch()` to send them to the store.

```
{
  type: 'ADD_TODO',
  text: 'Build my first Redux app'
}
```


Actions
Reducers
Store
dispatch
View
subscribe

Reducers

- Specifies how the application's state changes in response to something happening (an action).
- A pure function that takes the previous state and an action, and returns the next state.
- `(previousState, action) => newState`

Things You Should Never do in a Reducer

- Mutate its arguments
- Perform side effects like API calls and routing transitions
- Call non-pure functions

Reducer Composition

- The fundamental pattern of building Redux apps
- Split up reducer functions using child reducers.

- Workflow
  - Write top-level reducer to handle a particular function.
  - Break up the top-level reducer into a master reducer that calls smaller reducers to separate concerns.

Higher Order Reducer

- combineReducers() is a higher order reducer
- A higher order reducer is a higher order function that returns a new reducer.

Redux Store

- Holds the application state
- Allows access to state via `getState()`
- Allows state to be updated via `dispatch(action)`
- Registers listeners via `subscribe(listener)`
- Handles unregistering of listeners via the function returned by `subscribe(listener)`

Redux Store Design

- Many applications use data that is relational or "nested"

```
let orders = [
  {
    id:1,
    customer:{id:20,name:"Wilma",address:""},
    items:[{id:10,name:"Toaster",price:20}
  }
]
```

- Relational data mapped to objects can be complex and cause repetition.

Redux Store Design

- Solution: Normalize part of your store, and treat it like a database.

- **Data normalization:**
- Each type of data gets its own "table" in the state.
- Each "data table" should store the individual items in an object, with the IDs of the items as keys and the items themselves as the values.
- Any references to individual items should be done by storing the item's ID.
- Arrays of IDs should be used to indicate ordering.

Normalized Redux Store Example

```
{
  orders : {
    byId : {
      "order1" : {id : "order1", customer : "customer1",
                  items: ["item1","item2"]},
      "order2" : {id : "order2", customer : "customer1",
                  items: ["item1","item6"]}
    },
    allIds : ["order1", "order2"] },
  customers : {
    byId : {
      "customer1" : { ... }
      ...
}
```

Benefits of Normalizing Store

- Each item is defined in only one place
- Simplifies Reducer logic (flatter structure)
- Simplifies logic for retrieving and updating items
- Because each data type is separated, updates to a single type of data require fewer components to be re-rendered

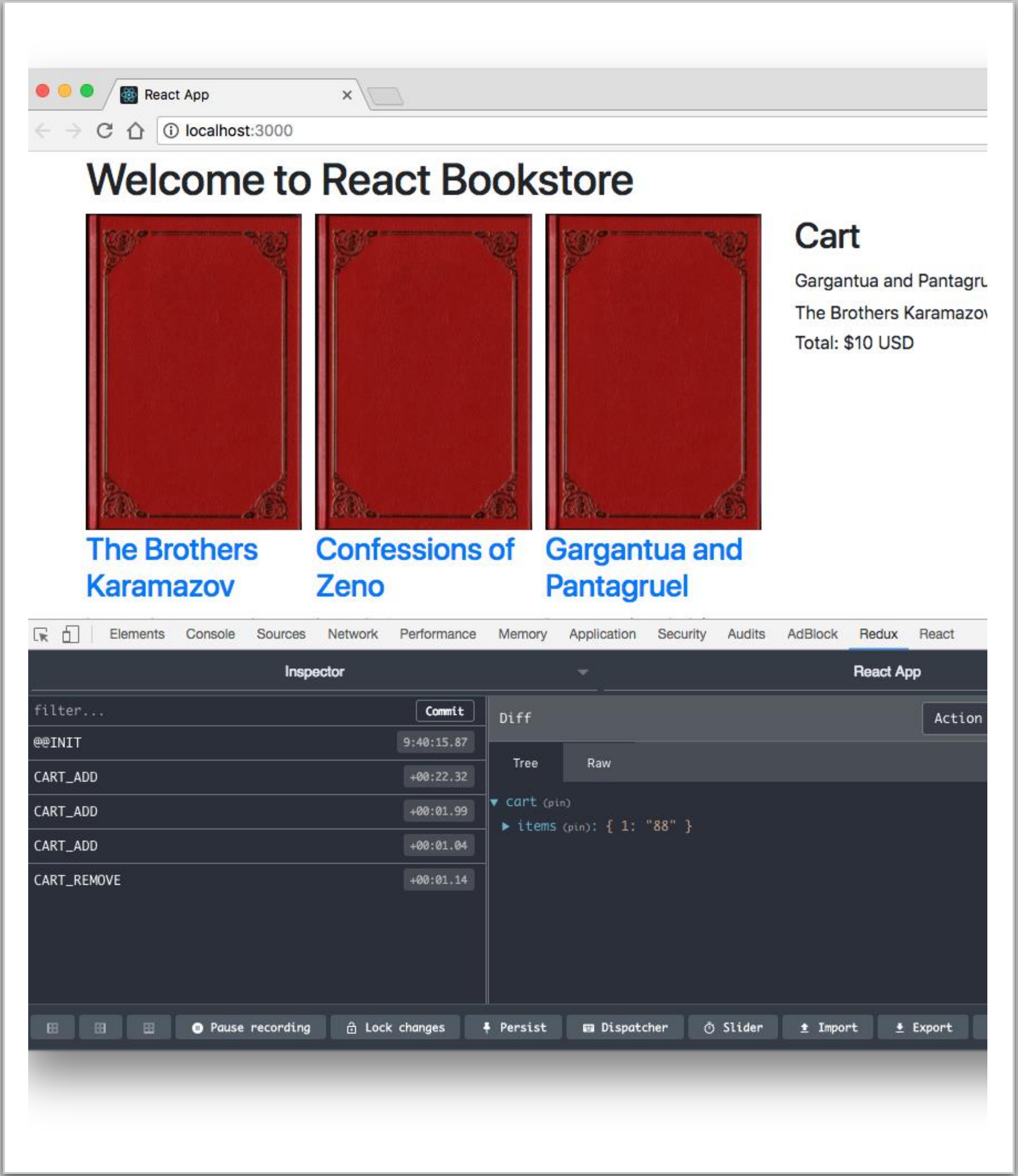
Redux Pros and Cons

- Redux Pros
  - Declarative
  - Immutable state
  - Mutation logic separate from views
  - Great for testing
- Redux Cons
  - More complicated than just using plain react components

Redux Toolkit

- Utilities for simplifying common Redux use cases.

Lab 16: Implementing Redux



Welcome to React Bookstore

The Brothers Karamazov

Confessions of Zeno

Gargantua and Pantagruel

Cart

Gargantua and Pantagru

The Brothers Karamazov

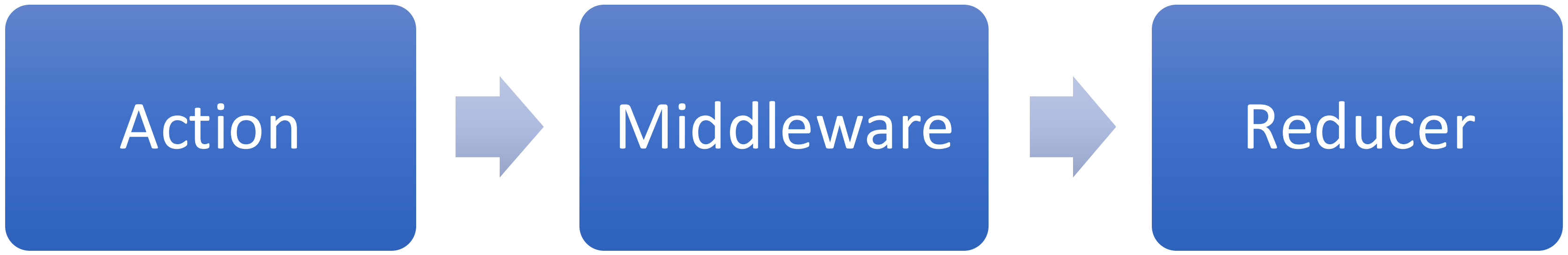
Total: $10 USD

```
@@INIT        9:40:15.87
CART_ADD      +00:22.32
CART_ADD      +00:01.99
CART_ADD      +00:01.04
CART_REMOVE   +00:01.14
```

```
▼ cart (pin)
  ▶ items (pin): { 1: "88" }
```

What is Redux Middleware?

- a higher-order function that composes a dispatch function to return a new dispatch function.
- a third-party extension point between dispatching an action and the reducer.



What is Middleware Good For?

- Logging actions
- Reporting errors
- Dispatching new actions
- Asynchronous requests

Redux Thunk

- Allows you to write action creators that return a function instead of an object.

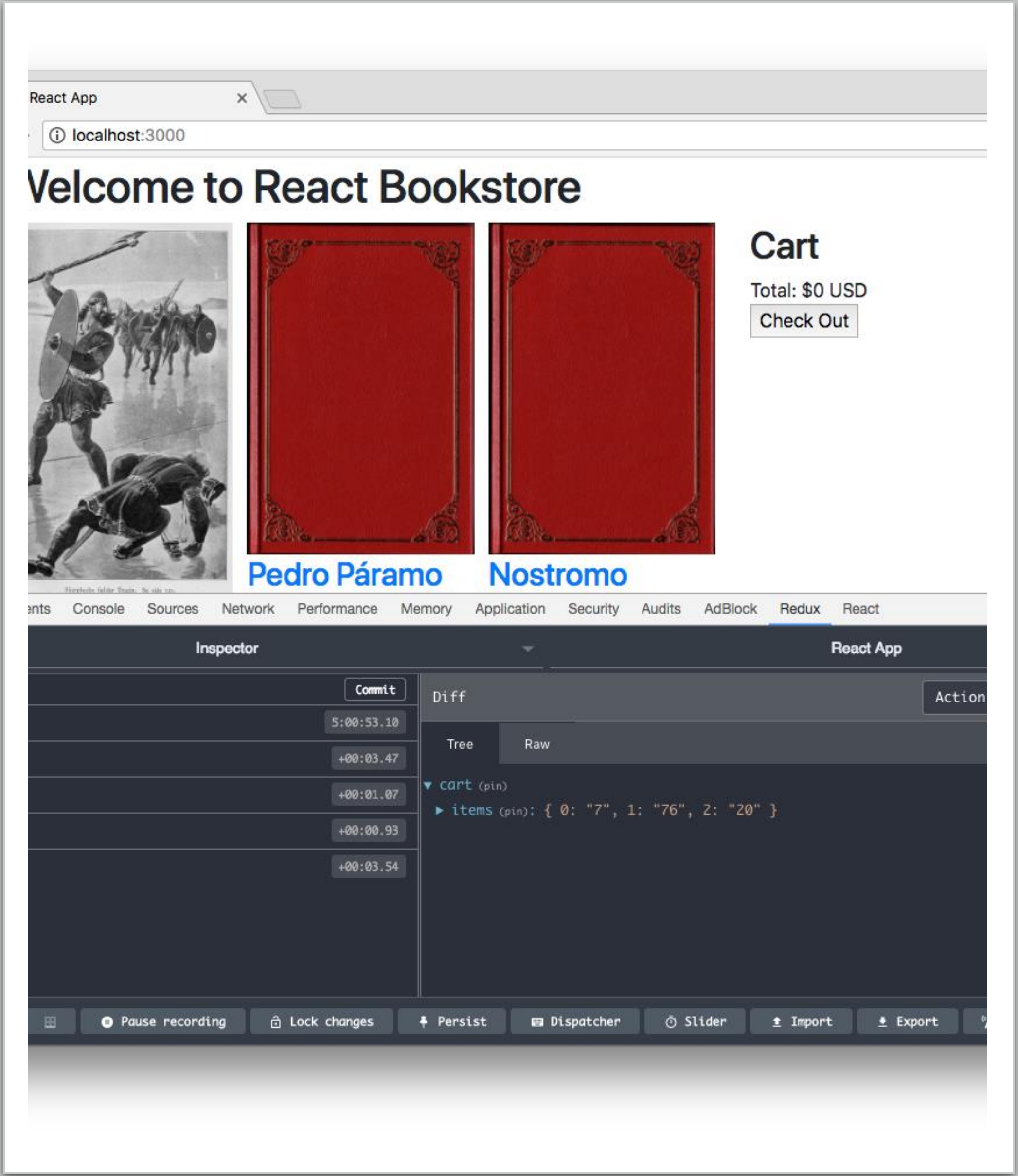
How is Thunk Useful?

- delay the dispatch of an action
- dispatch only if a certain condition is met
- perform async network operation
    - database
    - AJAX

How does Thunk work?

1. Redux Thunk middleware is added to the store
2. When an action creator returns a function, that inner function will get executed by the Thunk middleware.
3. The inner function receives the store methods `dispatch` and `getState()` as parameters.
4. On success, the Thunk action calls a standard action.

Lab 17: Redux Thunk



Welcome to React Bookstore

```
5:00:53.10
+00:03.47
+00:01.07
+00:00.93
+00:03.54
```

```
▼ cart (pin)
  ▶ items (pin): { 0: "7", 1: "76", 2: "20" }
```

Lab 18: Persisting Data in localStorage with Redux

Routing

React Router

- Declarative way to do routing
- Maps elements to URLs
- Dynamic routing
    - Routing takes place as the component is rendering
    - Not in a configuration
    - Almost everything in React Router is a component

Using React Router

- Import the version of React Router for your target environment (i.e. DOM or Native), plus other components

```
import { BrowserRouter, Routes, Route, Link } from 'react-
router-dom'
```

- Render the Router

```
root.render((
<BrowserRouter> <App/> </BrowserRouter>)
```

- Use <Link> to link to a new location (in `<App>` in this case)

```
<Link to="/dashboard">Dashboard</Link>
```

- Render Routes.

```
<Routes>
  <Route path="/dashboard" element={<Dashboard />}/>
</Routes>
```

Router Rendering Example

```
import { BrowserRouter } from 'react-router-
dom';

root.render((
  <BrowserRouter>
    <App/>
  </BrowserRouter>
))
```

- Use BrowserRouter when you have a server
- Use HashRouter if you're using a static file server.

Route Matching

- Route
  - Compares the value of the path prop to the current location's pathname.
  - Renders the element specified by the element prop.
  - Can be used anywhere you want to render based on location.
- Switch (v5), Routes (v6)
  - Can be (optionally) used for grouping Routes. Will iterate through a group and stop when a match is found.

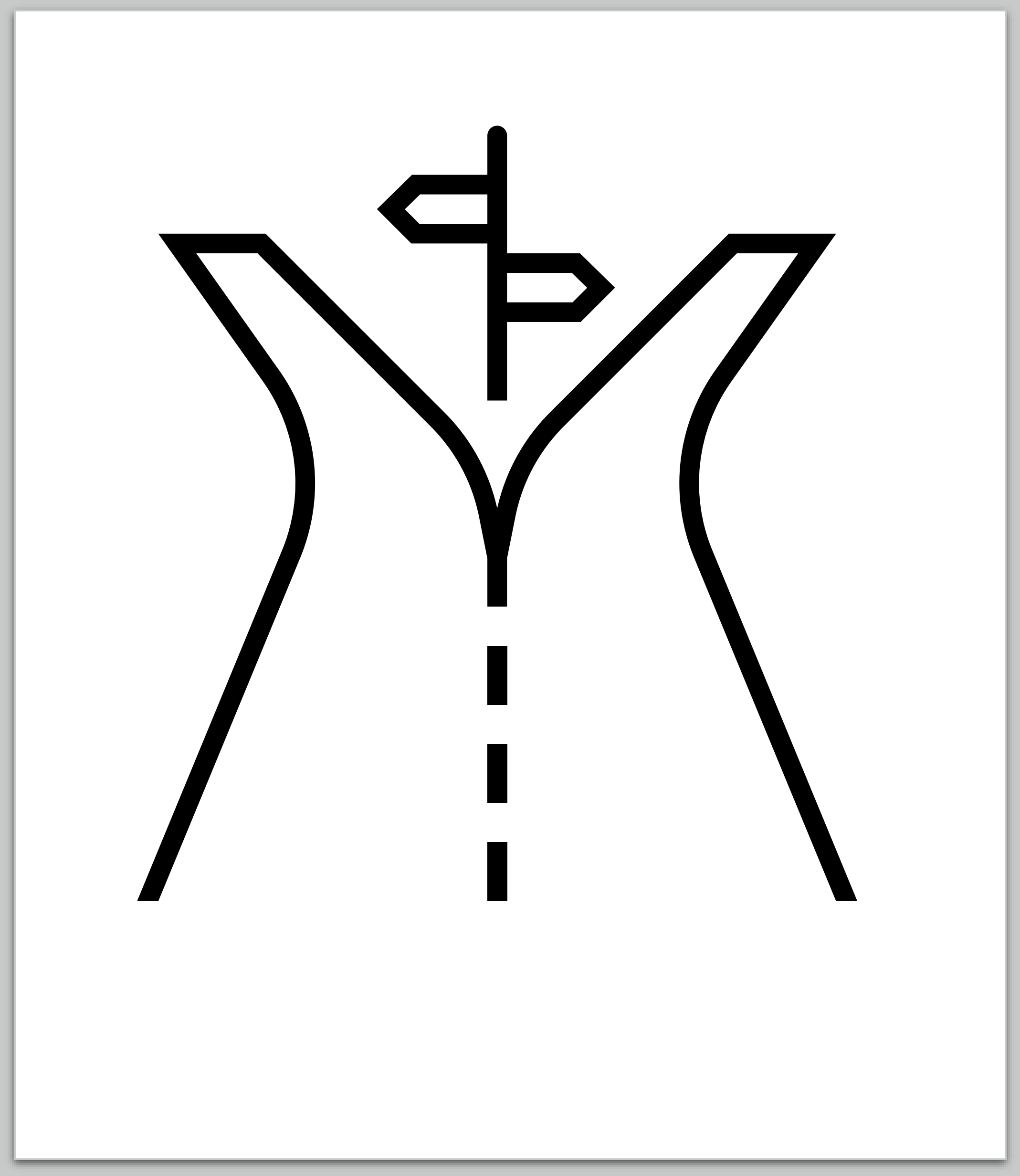
The match Object

- The Route component passes an object named match to components.
- The match object contains information about the match.
- One imported property of match is match.params
- Params are passed in the URL
  - `<Link to="/users/4">View User Details</Link>`
  - `<Route path="/users/:uid" element="<User />" />`
- You can access them using React Router's useParams hook.

Navigation

- <Link>
  - Creates links in your application.
  - Inserts an <a> in your HTML
- <NavLink>
  - Can be styled as "active" when it matches the current location.

Lab 19:
React Router



Deploying React

Development vs. Production

- Development build
  - Uncompressed.
  - Displays additional warnings that are helpful for debugging.
  - Contains helpers not necessary in production
  - ~669kb
- Production build
  - Compressed.
  - Contains additional performance optimizations.
  - ~147kb

Code Splitting

- Improves the performance of your app by lazy loading only the components that are currently needed.
- Two ways
  - dynamic import

```
import ("./math").then(math => {
  console.log(math.add(43,32));
})
```

  - `React.lazy`

```
const MyComponent =
  React.lazy(()=>import ('./Header'));
```

Using Lazy Loaded Component

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() =>
                       import('./OtherComponent'));

function MyComponent() {

  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

Building Your Project

- In create-react-app, run the build script
- npm run build
- Build makes a build directory, which contains:
  - index.html
  - static assets (js and css)
  - sourcemap files
  - asset-manifest.json

Deploying React

- React apps can be served using plain old HTTP servers, node servers, or cloud hosting.
- Popular hosting options include:
  - Netlify
  - Firebase
  - Heroku
  - AWS
- To deploy/host for free, you can use Github Pages

Server-side React

- Allows you to pre-render components' initial state on the server
- Methods:
  - `renderToString`
    - Returns an HTML string
    - Send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.
  - `renderToStaticMarkup`
    - Works the same as `renderToString`, but doesn't add in the extra React DOM elements
    - Good for using React as a static page generator

Render Caching

- A technique for making web pages load faster on subsequent visits.

1. Encapsulate load state so that no server call is required to render the initial view
2. Make all API calls before render()
3. Cache locally in the unload handler
4. Restore the last known state on load
5. Render the last known state in react using ReactDOM.hydrate

ReactDOM.hydrate

- import {render, hydrate} from "react-dom"

```
if (window.hasRestoredState) {
  hydrate(<MyPage />, renderTarget);
  } else {
  render(<MyPage />, renderTarget);
}
```

Advanced Topics

GraphQL

What is GraphQL

a query language for APIs

a runtime for fulfilling queries

How does GraphQL Work?

Query using JSON-like syntax

Data returned matches shape of query

Example Query and Response

```
{
    customer {
        firstname
        lastname
        address
        city
        state
        zip
    }
}
```

```
{
    ”customer”: {
        ”firstname”: “June”,
        “lastname”:”Sommerville”,
        “address”:”861 Oak St”,
        “city”:”Sacramento”,
        “state”:”California”,
        “zip”:”95814”
    }
}
```

GraphQL is a pattern

Can be implemented using any language

Tools are available for working with JavaScript, Go, PHP, Java, C#, Python, Swift, Rust, Ruby, and more.

GraphQL and JavaScript

Server

- Apollo Server
- Express GraphQL

- Client
  - Apollo Client
  - AWS Amplify
  - Relay

GraphQL Pros and Cons

- Pros
  - Even more declarative
  - No custom getter logic
  - Tight server integration
- Cons
  - Requires GraphQL server
  - More complexity
  - Less flexible

Micro Frontends

- Micro Frontends decompose frontend applications into independent units in the same way that microservices decompose the back end.
- Gives teams the ability to independently develop, test, and deploy each micro frontend.

React Micro Frontends

- Challenges:
  - Overhead: Each micro frontend needs to access the same libraries, CSS, routes
    - Can create massive problem of duplication of code and poor performance
  - Cross-cutting concerns: Each frontend needs to share authentication, for example.
  - Communication between micro frontends is more complex than between components in an app.

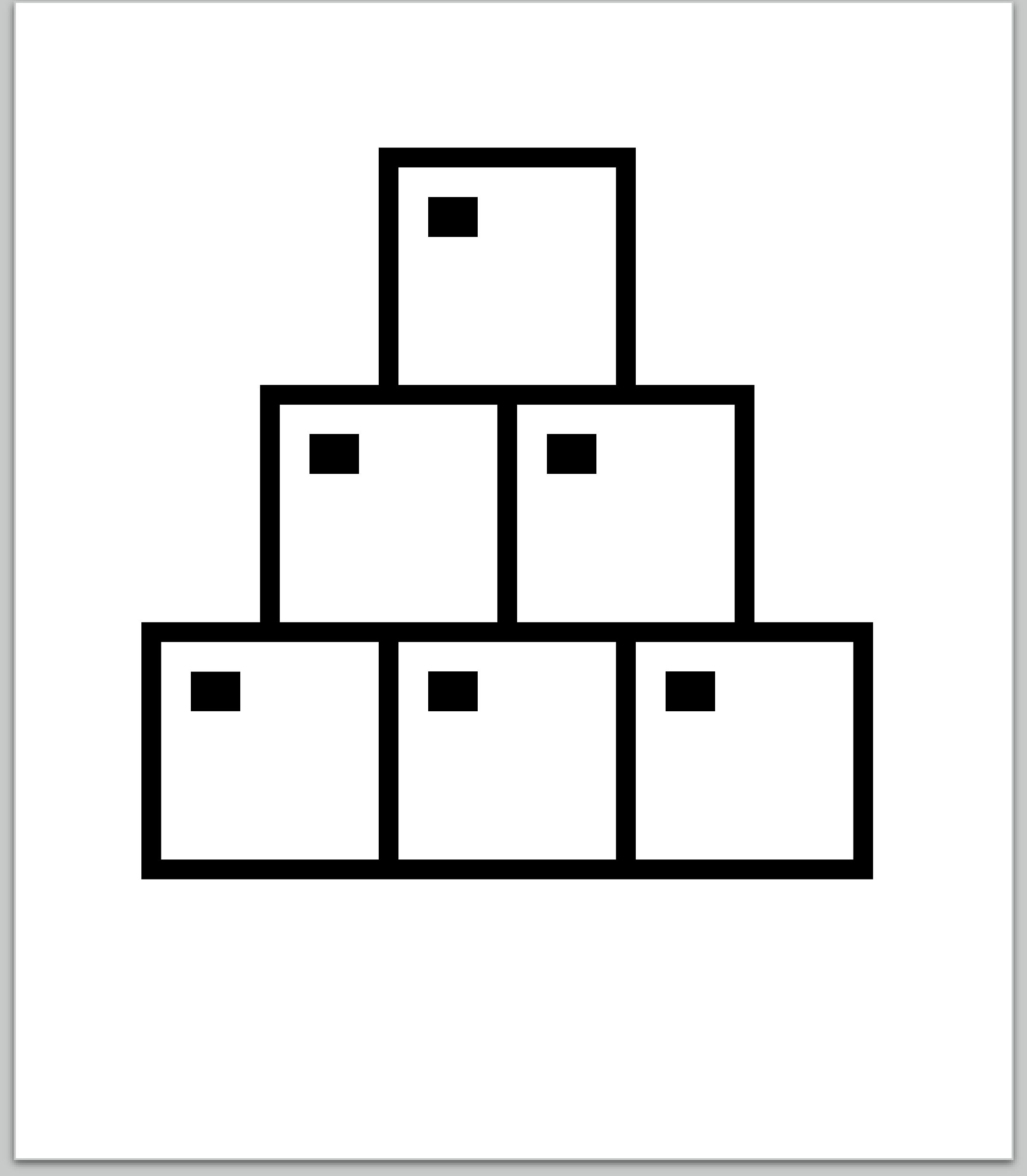
Common strategies for React Micro frontends

- Use a container application to handle rendering and routing between the micro frontends.
- Use a CDN to host shared frontend libraries and resources.
- Make use of React Portals to render components from an application outside of its root DOM node.

Shared State in React Micro Frontends

- Four strategies
  - Web workers
  - Props and callbacks
  - Custom events
  - Pub Sub library

Lab 19: Creating Micro Frontends



Thank You

chris@minnick.com

https://chrisminnick.com

Disclaimer and Copyright