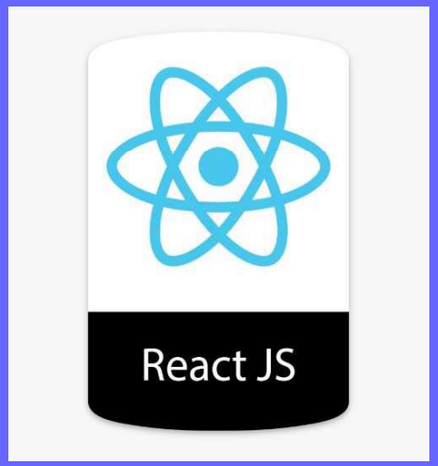
1

React JS



Frond End Applications Development

Brief history of Web Applications

- Initially: static HTML files only with HTML forms for input
- Common Gateway Interface (CGI)
  - Certain URLs map to executable programs that generate web page
  - Program exits after Web page complete
  - Introduced the notion of stateless servers: each request independent, no state carried over from previous requests. (Made scale-out architectures easier)
  - Perl typically used for writing CGI programs

First-generation web app frameworks

Examples: (PHP, ASP.net, Java servlets)

- Incorporate language runtime system directly into Web server
- **Templates**: mix code and HTML - HTML/CSS describes view
- Web-specific library packages:
  - URL handling
  - HTML generation
  - Sessions
  - Interfacing to databases

Second-generation frameworks

Examples: (Ruby on Rails, Django):

- **Model-view-controller**: stylized decomposition of applications
- Object-relational mapping (**ORM**): simplify the use of databases (make database tables and rows appear as classes and objects)
  - Easier fetching of dynamic data

Third-generation frameworks

Example: Angular

- JavaScript frameworks running in browser - More app-like web apps
  - Interactive, quick responding applications - Don't need server round-trip
- Frameworks not dependent on particular server-side capabilities
  - Node.js - Server side JavaScript
  - No-SQL database (e.g. MongoDB)
- Many of the concepts of previous generations carry forward
  - Model-view-controller
  - Templates - HTML/CSS view description

Model-View-Controller (MVC) Pattern

- **Model**: manages the application's data
  - JavaScript objects. Photo App: User names, pictures, comments, etc.
- **View**: what the web page looks like
  - HTML/CSS. Photo App: View Users, View photo with comments
- **Controller**: fetch models and control view, handle user interactions
  - JavaScript code. Photo App: DOM event handlers, web server communication

MVC pattern been around since the late 1970's

- Originally conceived in the Smalltalk project at Xerox PARC

View Generation

- Web App: Ultimately need to generate HTML and CSS
- **Templates** are commonly used technique. Basic ideas:
  - Write HTML document containing parts of the page that are always the same.
  - Add bits of code that generate the parts that are computed for each page.
  - The template is expanded by executing code snippets, substituting the results into the document.
- Benefits of templates (Compare with direct JavaScript to DOM programming)
  - Easy to visualize HTML structure
  - Easy to see how dynamic data fits in
  - Can do either on server or browser

AngularJS view template (HTML/CSS)

```
...
<body>
    <div class="greetings">
        Hello {{models.user.firstName}},
    </div>
    <div class="photocounts">
        You have {{models.photos.count}} photos to review.
    </div>
</body>
```

Angular has rich templating language (loops, conditions, subroutines, etc.). Later...

Controllers

- Third-generation: JavaScript running in browser Responsibilities:
- Connect models and views
  - Server communication: Fetch models, push updates
- Control view templates
  - Manage the view templates being shown
- Handle user interactions
  - Buttons, menus, and other interactive widgets

AngularJS controller (JavaScript function)

```
function userGreetingView ($scope, $modelService) {
    $scope.models = {};

    $scope.models.users = $modelService.fetch("users");
    $scope.models.photos = $modelService.fetch("photos");


    $scope.okPushed = function okPushed() {
        // Code for ok button pushing
    }
}
```

Angular creates $scope and calls controller function called when view is instantiated

Model Data

- All non-static information needed by the view templates or controllers
- Traditionally tied to application's database schema
  - Object Relational Mapping (ORM) - A model is a table row
- Web application's model data needs are specified by the view designers But need to be persisted by the database
- Conflict: Database Schemas don't like changing frequently but web application model data might (e.g. user will like this view better if we add … and lose ...)

Angular doesn't specify model data (JavaScript objs)

- Angular provides support for fetching data from a web server
  - REST APIs
  - JSON frequently used

On Server:

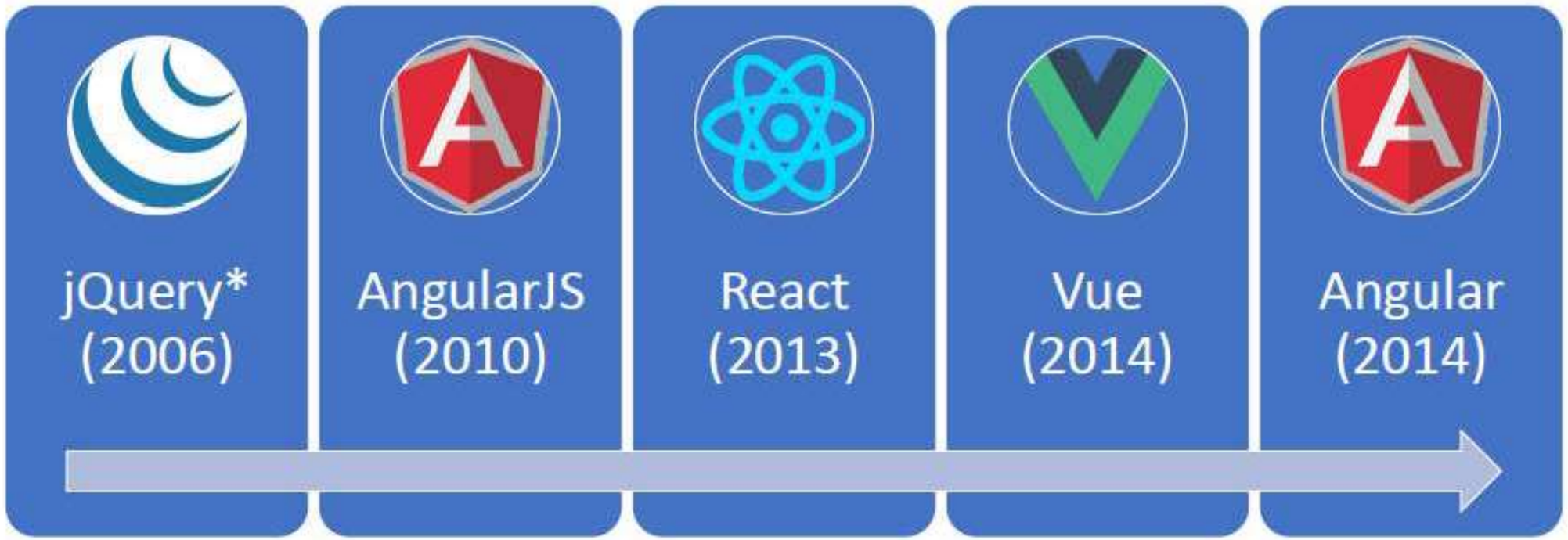
- Mongoose's Object Definition Language (ODL) has "models"

```
var userSchema = new Schema({
  firstName: String,
  lastName: String,
});
var User = mongoose.model('User', userSchema);
```

Fourth-generation frameworks

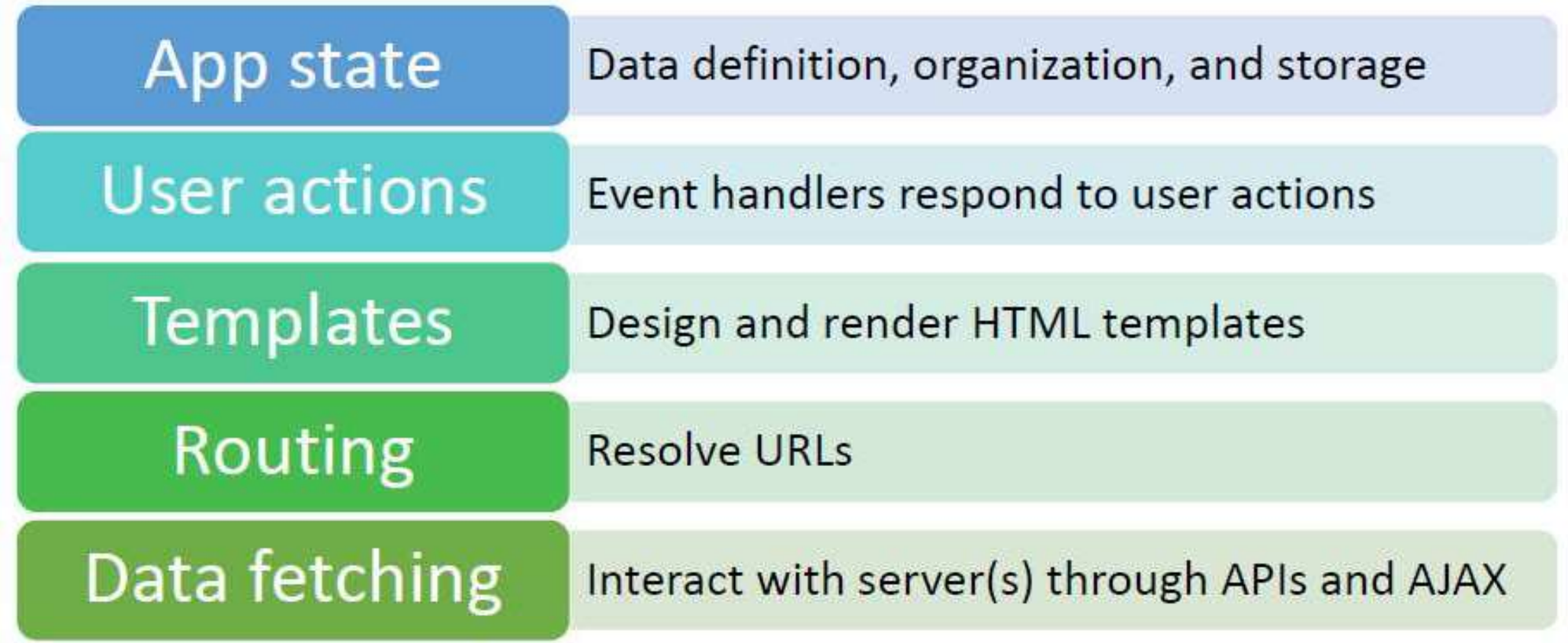
Examples: React.js, Vue.js, Angular(v2)

- Many of the concepts of previous generations carry forward
  - JavaScript in browser
  - Model-view-controllers
  - Templates
- Focus on JavaScript components rather than pages/HTML
  - Views apps as assembled reusable components rather than pages.
  - Software engineering focus: modular design, reusable components, testability, etc.
- Virtual DOM
  - Render view into DOM-like data structure (not real DOM)
  - Benefits: Performance, Server-side rendering, Native apps



* jQuery is more often considered a **library** than a **framework**

Common tasks in front-end development



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

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

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

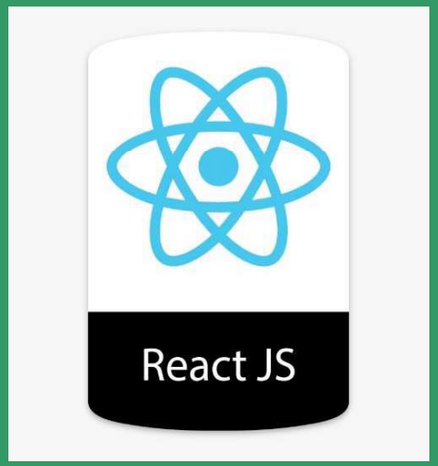
Cont …

Angular is what you get if you think the biggest problem with writing webapps is that Javascript isn't Java; Ember that it's not Ruby. (I kid. But I'm less familiar with those frameworks.) And so on. Everyone has their own ideas of what's hard to solve.

React + Redux is based on the idea that what's really hard with writing webapps is non-deterministic behaviour and unclear data flow. And if you've worked on a large Knockout or Backbone project, you're probably inclined to agree.

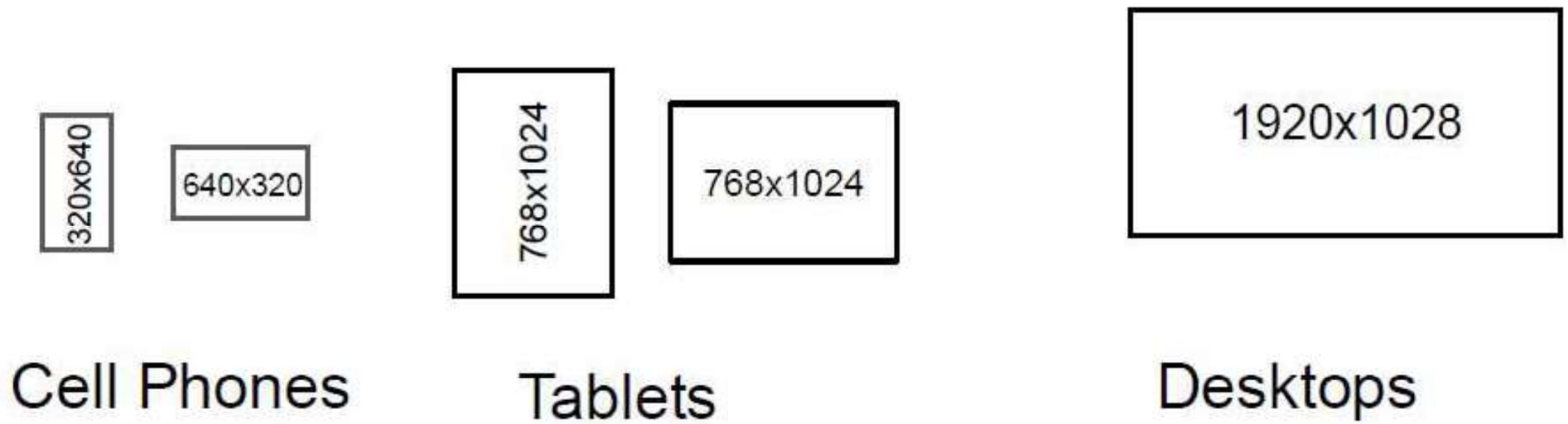
Cont …

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



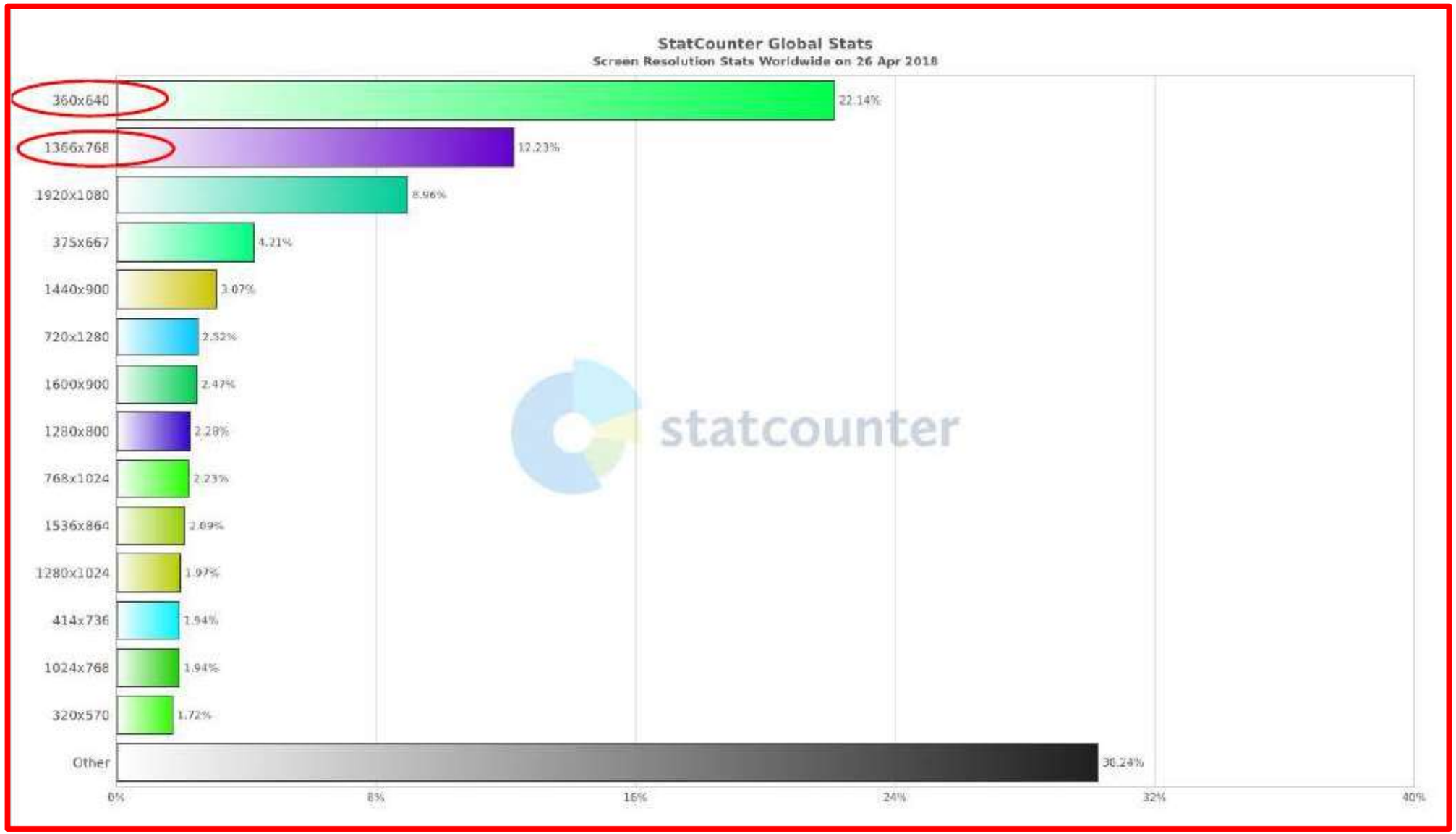
Reponsive Web Design

Web App Challenges: Screen real estate



- Do we need to build N versions of each web application?

Statistics

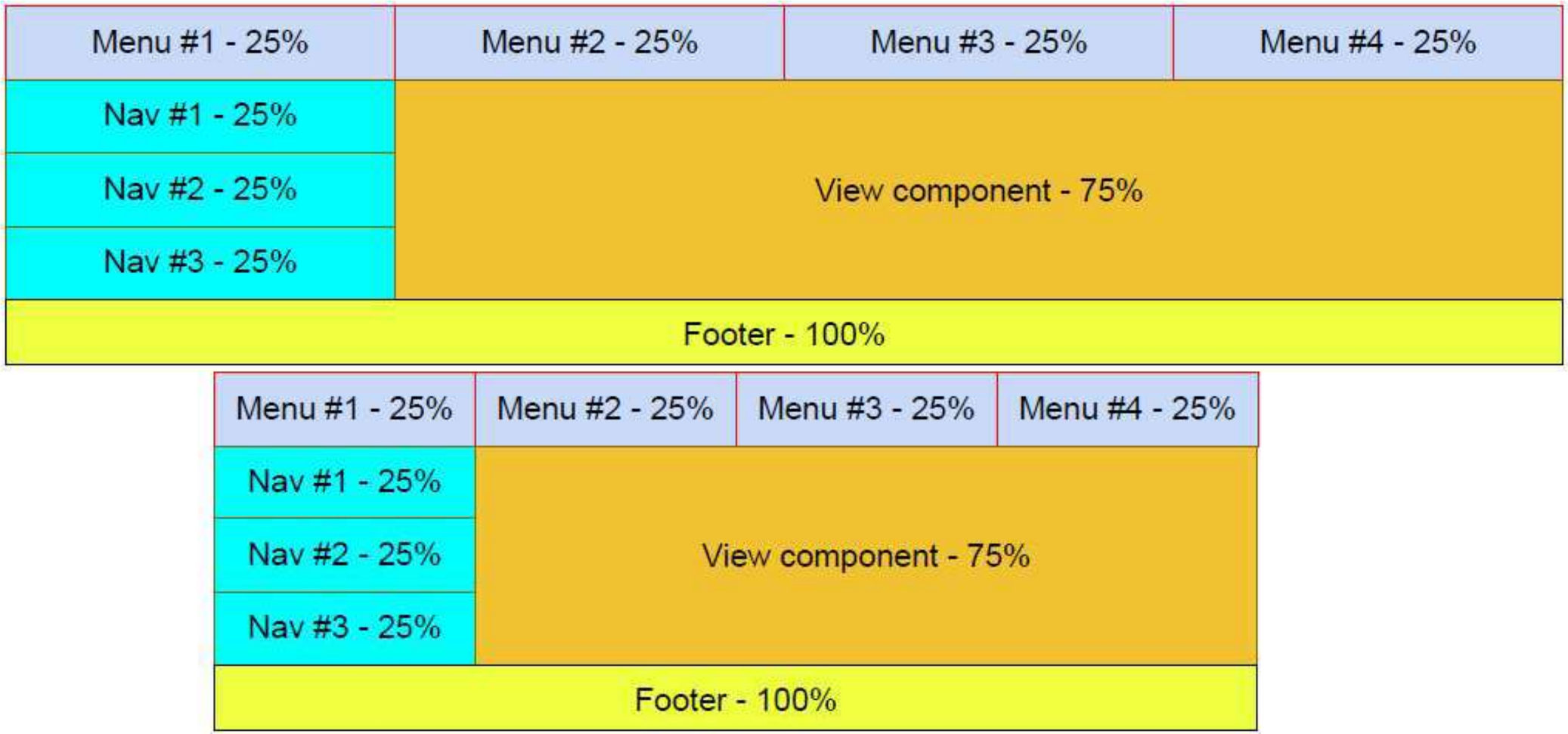

StatCounter Global Stats
Screen Resolution Stats Worldwide on 26 Apr 2018
360x640
1366x768
1920x1080
375x667
1440x900
720x1280
1600x900
1280x800
768x1024
1536x864
1280x1024
414x736
1024x768
320x570
Other
22.14%
12.23%
8.96%
4.21%
3.07%
2.52%
2.47%
2.28%
2.23%
2.09%
1.97%
1.94%
1.94%
1.72%
30.24%
0%
8%
16%
24%
32%
40%
statcounter

Responsive Web Design

- Content is like water!
  - The web app should flow into and fill whatever device you have.
- Possible with CSS extensions:
  - Add grid layout system with relative (e.g. 50%) rather than absolute (e.g. 50pt) measures
    - Specify element packing into columns and rows
  - Add `@media` rules based on screen sizes
    - Switch layout based on screen size
  - Made images support relative sizes
    - Autoscale image and videos to fit in screen region

```
img { width: 100%; height: auto; }
video { width: 100%; height: auto; }
```

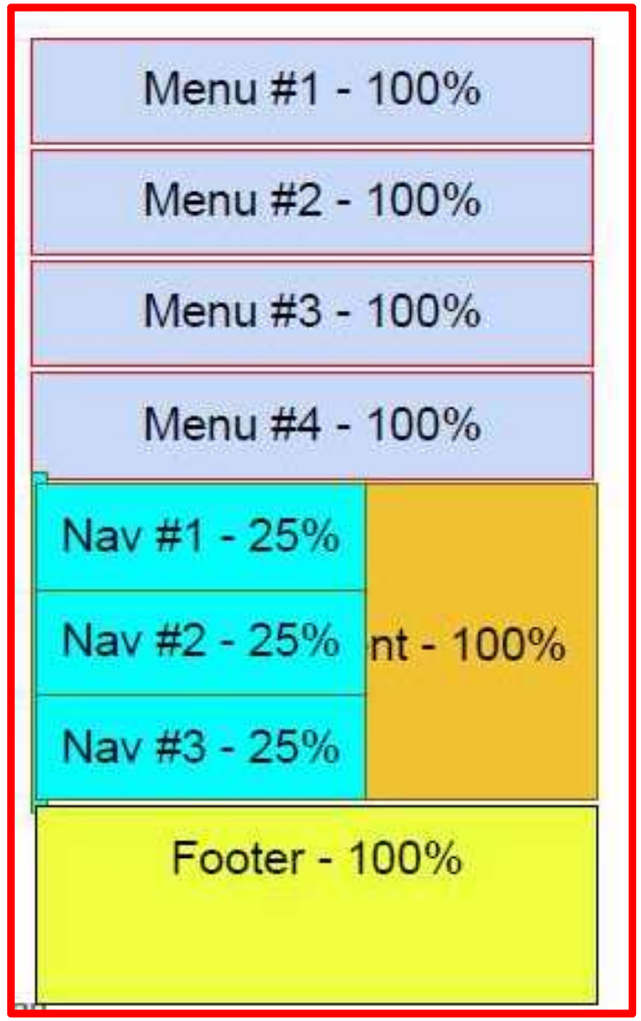
Example of Responsive Web Layout



CSS Breakpoints

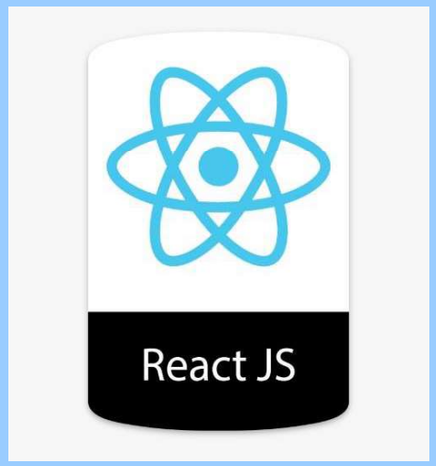
CSS Rules:

```
@media only screen and (min-width: 768px) {
     /* tablets and desktop layout */ }

@media only screen and (max-width: 767px) {
      /* phones */ }

@media only screen and (max-width: 767px)
     and (orientation: portrait) {
          /* portrait phones */ }
```



Responsive implementation

- Build components to operate at different screen sizes and densities
  - Use relative rather than absolute
  - Specify sizes in device independent units
- Use CSS breakpoints to control layout and functionality
  - Layout alternatives
  - App functionality conditional on available screen real estate
- Mobile first popular
  - Expand a good mobile design to use more real estate

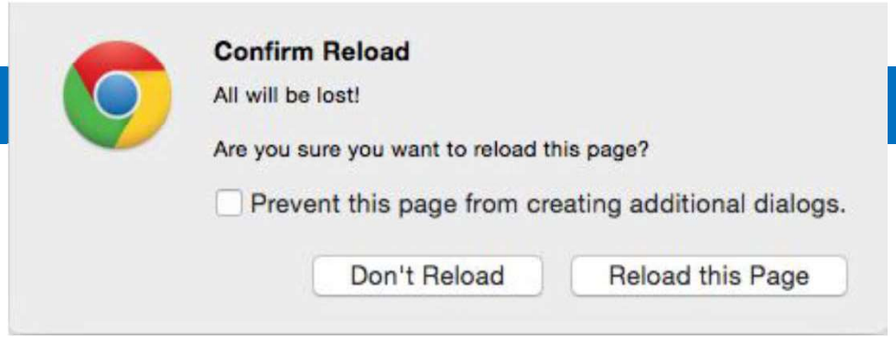


Single Page Applications (SPA)

Web Apps and Browsers

- Web apps run in browsers (by definition)
- Users are use to browsing in browsers
  - Browser maintains a history of URLs visited
    - Back button - Go back in history to previous URL
    - Forward button - Go forward in history to next URL
  - Can move to a different page
    - Typing into location bar or forward/back buttons
    - Selecting a bookmarked URL
    - Page refresh operation
- Browser tosses the current JavaScript environment when navigating to a different page
  - Problematic for JavaScript frameworks: URL (and cookies) are the only information preserved

Problem with some web apps



- Initial: pages served from web server
  - Each page had a URL and app switched between pages served by web server
- Early JavaScript apps: Website a single page/URL with the JavaScript

Problem: Restart web app on navigation (Can lose a lot of work!)

```
window.onbeforeunload = function(e) { return 'All will be lost!'; }
```

- Users expect app in browser to do the right thing:
  - Navigate with forward and back buttons, browser page history
  - Navigate away and come back to the app
  - Bookmark a place in the app
  - Copy the URL from the location bar and share it with someone
  - Push the page refresh button on the browser

Changing URL without page refresh

- Can change hash fragment in URL without reload

  http://example.com

  http://example.com#fragment

  http://example.com?id=3535

  http://example.com?id=3535#fragment

- HTML5 give JavaScript control of page reload
  - Browser History API - window.history - Change URL without reloading page

Deep linking

- Concept: the URL should capture the web app's context so that directing the browser to the URL will result the app's execution to that context
  - Bookmarks
  - Sharing
- Context is defined by the user interface designer!

  Consider: Viewing information of entity and have an edit dialog open

  Should the link point to the entity view or to the entity & dialog?

  Does it matter if I'm bookmarking for self or sharing with others?

  How about navigating away and back or browser refresh?

Deep linking in Single Page Apps

Two approaches:

1. Maintain the app's context state in the URL
   + Works for browser navigation and refresh
   + User can copy URL from location bar
2. Provide a share button to generate deep linking URL
   + Allows user to explicitly fetch a URL based on need
   + Can keep URL in location bar pretty

Either way web app needs to be able to initialize self from deep linked URL

Ugly URLs

```
http://www.example.org/dirmod?sid=789AB8&type=gen&mod=Core+Pag
es&gid=A6CD4967199
```

versus

```
http://www.example.org/show/A6CD4967199
```

What is that ugly thing in the location bar above my beautiful web application?

```
https://www.flickr.com/photos/jarnasen/24593000826/in/explore-2016-01-26/
```

ReactJS support for SPA

- ReactJS has no opinion! Need 3rd party module.
- Example: React Router Version 5 https://v5.reactrouter.com/
  - Idea: Use URL to control conditional rendering
  - Newer version 6 is available using same concepts as v5 but slightly different syntax
- Various ways of encoding information in URL
  - In fragment part of the URL: `HashRouter`
  - Use HTML5 URL handler: `BrowserRouter`
- Import as a module:

```
import {HashRouter, Route, Link, Redirect} from 'react-router-dom';
```

Example React Router V5

```
<HashRouter>
   <div>
     ...
     <Route path="/states" component={States} />
     ...
     <Link to="/states">States</Link>
      ...
    </div>
 </HashRouter>
```

- JSX block controlled by URL enclosed in `HashRouter`
- `Route` will render the component if URL matches.
- Use `Link` component to generated hyperlink:

```
<a href="#/states">
  States
</a>
```

Passing parameters with React Router

- Parameter passing in URL

```
<Route
    path="/Book/:book/ch/:chapter"
    component={BookChapterComponent}
/>
```

- Parameters put in prop.match of the component

```
function BookChapterComponent({ match }) {
  return ( <div>
             <h3>Book: {match.params.book}</h3>
             <h3>Chapter: {match.params.chapter}</h3>
           </div> );
}
```

```
<Link to="/Book/Moby/ch/1">
    Moby
</Link>
```

Book: Moby

Chapter: 1

Route: component=, render=, children=

- `component={BookChapterComponent}`
  - Mounts components on match (unmounts on URL change)
  - Passes `match` object with: `params`, `url`, `history`
- `render={props => <BookChapterComponent book={props.match.params.book} chapter={props.match.params.chapter} />}`
  - Calls function with props having `match` object from above.
  - Doesn't mount/unmount component (does update it)
- `children=` - Like render= except is called regardless of the match
  - match will be `null` if URL doesn't match
  - Useful if you want to have something always render but only active on matching URL.

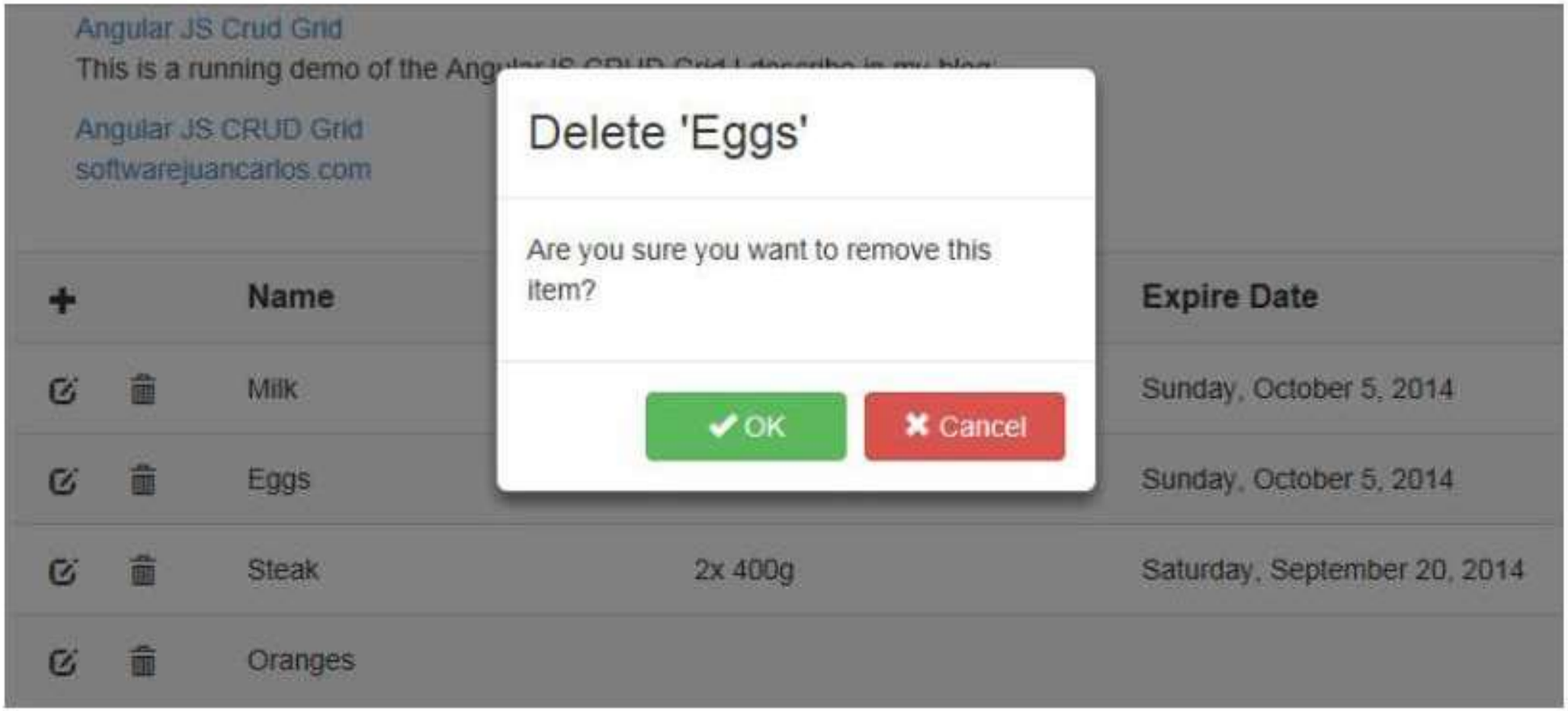
Multiple route matches have precedence order: component, render, children

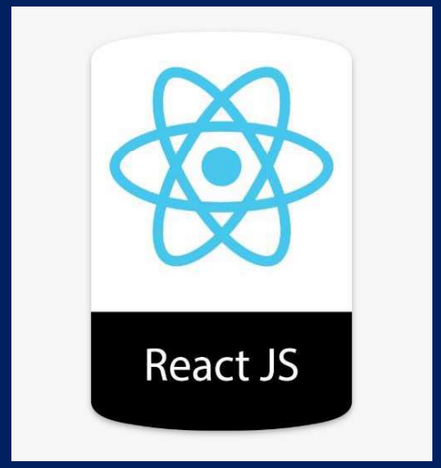
`Switch` is useful with multiple Route - Renders the first matching one

Example

- What to keep in URL: table length, viewport in table, search box, sort column, etc.
- Is it different for bookmark or share? Nav away and back?

Example: Not everything goes in URL



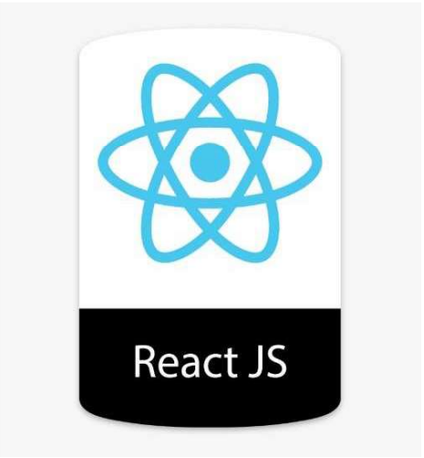


React JS Intro

What Is React ?

What is React ?

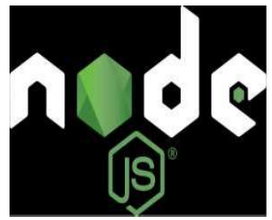
- React JS is a open source JavaScript libraries engineered to make it as simple and efficient as possible to build client-side web and hybrid mobile applications based on JavaScript, HTML5, and CSS.



- JavaScript framework for writing the web applications
  - Like AngularJS - Snappy response from running in browser
  - Less **opinionated**: only specifies rendering view and handling user interactions
- Uses Model-View-Controller pattern
  - View constructed from Components using pattern
  - Optional, but commonly used HTML templating
- Minimal server-side support dictated
- Focus on supporting for programming in the large and single page applications
  - Modules, reusable components, testing, etc.

Prerequisites

- HTML, CSS and JS
- Basics of Type Script

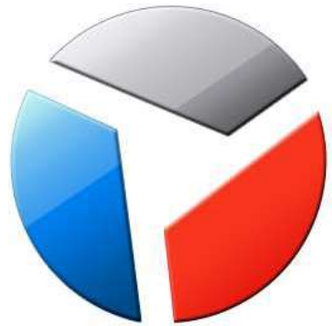


React is designed to meet the following application needs:

- Add interactivity to an existing page.
- Create a new end-to-end client-side web application using JavaScript, HTML5, CSS, and best practices for responsive design.
- Create a hybrid mobile application that looks and feels like a native iOS, Android or Windows application.
- It is a “ Structural Framework for Dynamic Web Apps “.
- Helps build interactive, modern Web Applications by increasing abstraction between the Developer and Common Web App Development Tasks.

Why React ?

- Modular Approach
- Re Usable Code
- Development Quicker and easier
- Unit Testable
- Many JS Providers



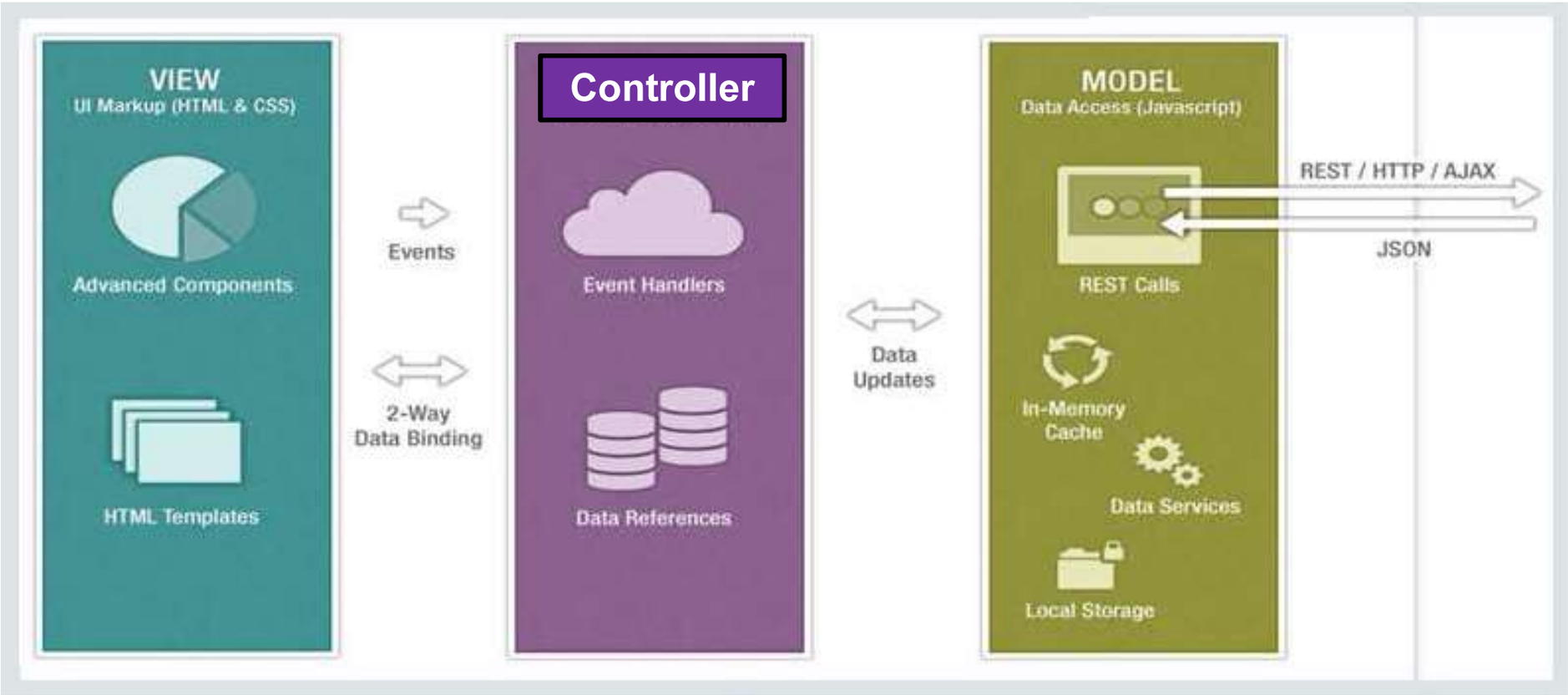
Cont …

- Declarative programming should be used for building UI and associating Software Components.
- “Imperative Programming is Very good for Business Logic”
- React encourages loose coupling between Presentation, Data and Logic Components.

React Architecture

- React supports the Model-View-Controller (MVC) architectural design pattern.

The MVC

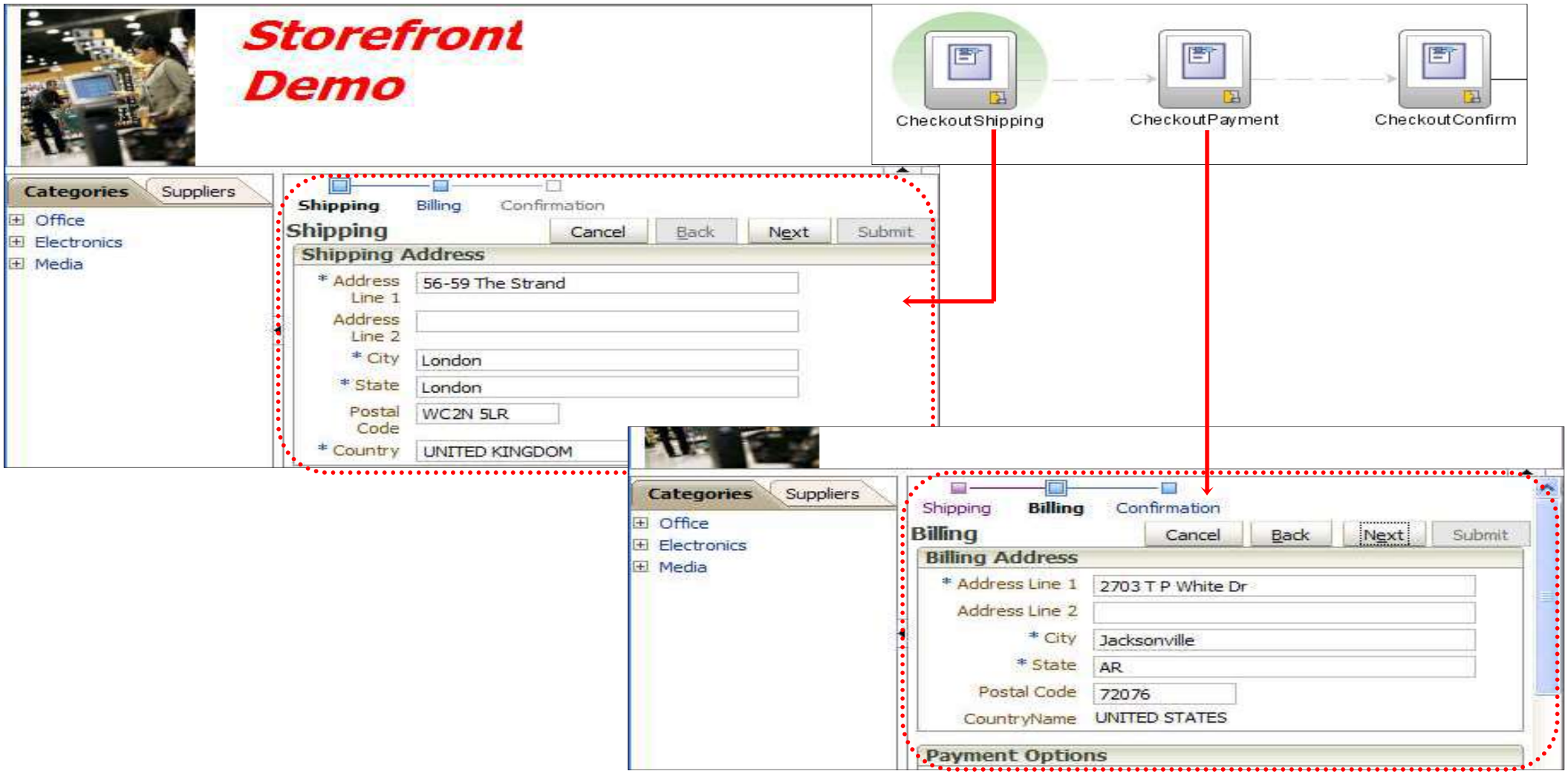

VIEW
UI Markup (HTML & CSS)
Advanced Components
HTML Templates
Events
2-Way
Data Binding
Controller
Event Handlers
Data References
Data
Updates
MODEL
Data Access (Javascript)
REST Calls
In-Memory
Cache
Data Services
Local Storage
REST / HTTP / AJAX
JSON

Why React is becoming Famous ?

1. One Way Binding
   - Binding Data to the UI Components.
   - Data Binding is Automatic Synchronization Between VIEW (HTML) and MODEL (Simple JS Variables).
   - MVC(Model View Controller)
2. Structure front End Code
   - A Pure Separation of Concern
   - HTML will be Focusing on UI (VIEW)
   - JavaScript Will Be Focusing on Model

3. Routing Support
   - Navigation Support
   - Focus is On SINGLE PAGE APPLICATION
   - Whole Application Will be Revolved Around A Single Page , But Navigation will Take Place not Between Pages but in turn Between Fragments.
   - Giving a Desktop Application Look and Feel.

Using a Bounded Task Flow



Cont …

4. Validation framework that provides UI element and component validation and data converters
5. Powered By React and There is a JS Community
6. Focus is on Developing Web Application Using Design Pattern
7. Messaging and event services for both Model and View layers
8. Caching services at the Model layer for performance optimization of pagination and virtual scrolling
9. Connection to data sources through Web services, such as Representational State Transfer (REST) or WebSocket
10. Support for Testing (Eg : KARMA)
11. Logging and Security Support

Hybrid Mobile Application Development Toolkit Features

- React includes support for hybrid mobile applications that run on iOS, Android, and Windows mobile devices within the Apache Cordova container.
- Apache Cordova enables you to use web technologies such as HTML5, CSS, and JavaScript to develop applications that you can deploy to mobile devices.
- Using the Cordova JavaScript APIs to access native device services, major mobile platforms such as Android, iOS, and Windows can be supported from a common code base.

Hybrid Mobile Application Development

1. JavaScript and HTML in the same file (JSX)

2. Embrace functional programming

3. Components everywhere

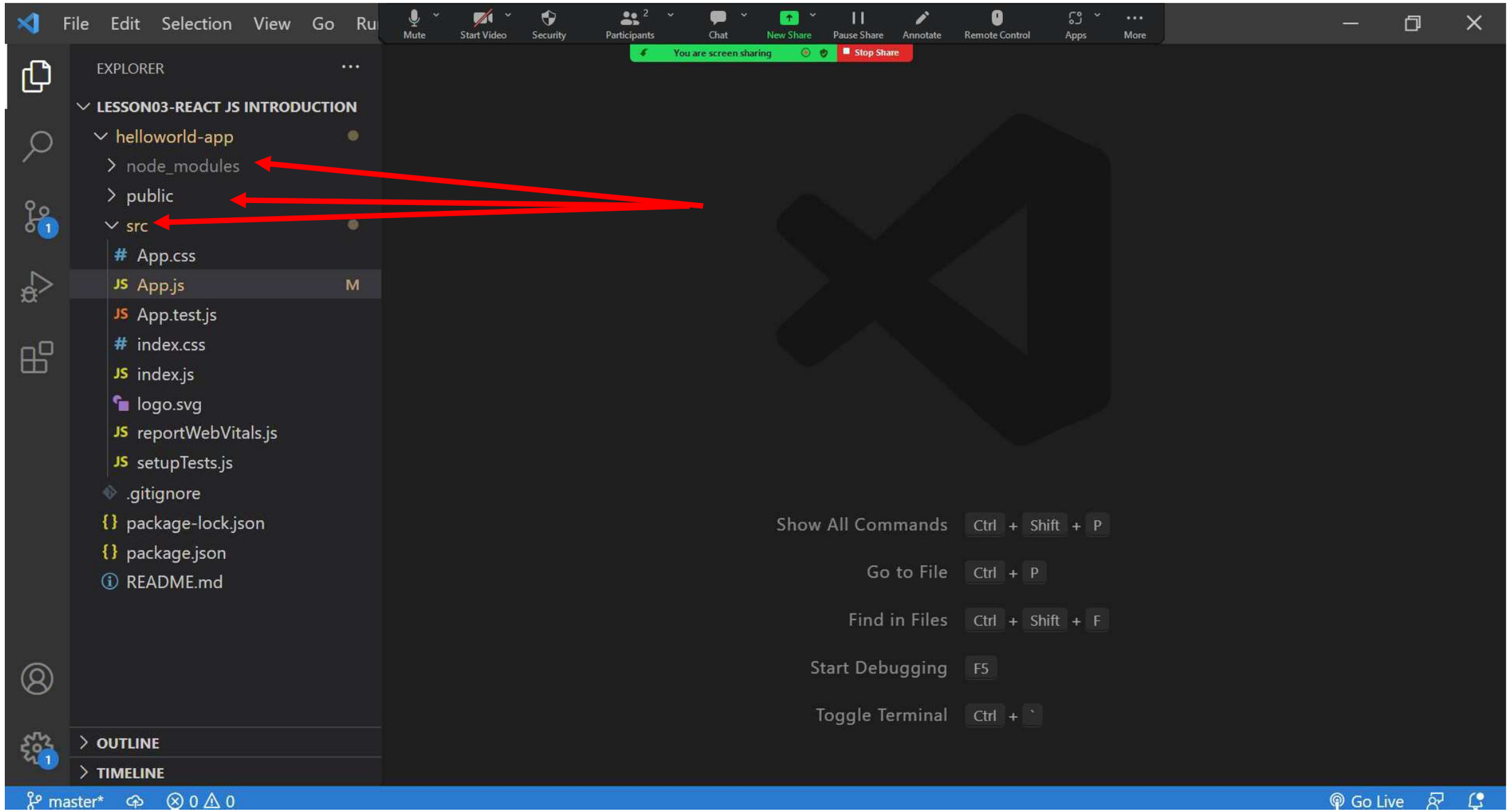
ReactJS Web Application Page

```
<!doctype html>
<html>
   <head>
      <title>CS142 Example</title>
   </head>
   <body>
      <div id="reactapp"></div>
      <script src="./webpackOutput/reactApp.bundle.js"></script>
   </body>
</html>
```

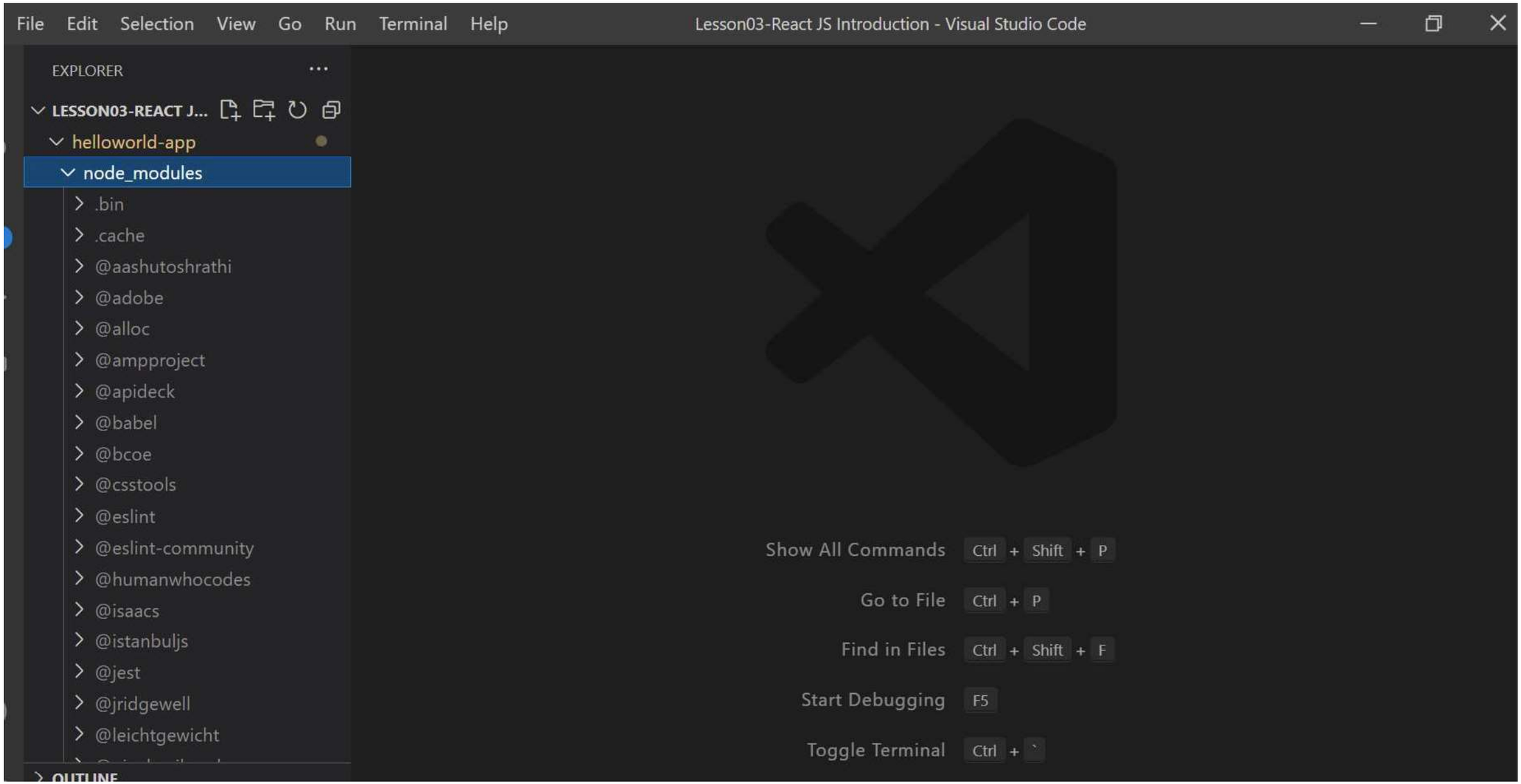
ReactJS applications come as a **JavaScript blob** that will use the DOM interface to write the view into the div.

Scaffolding an Application

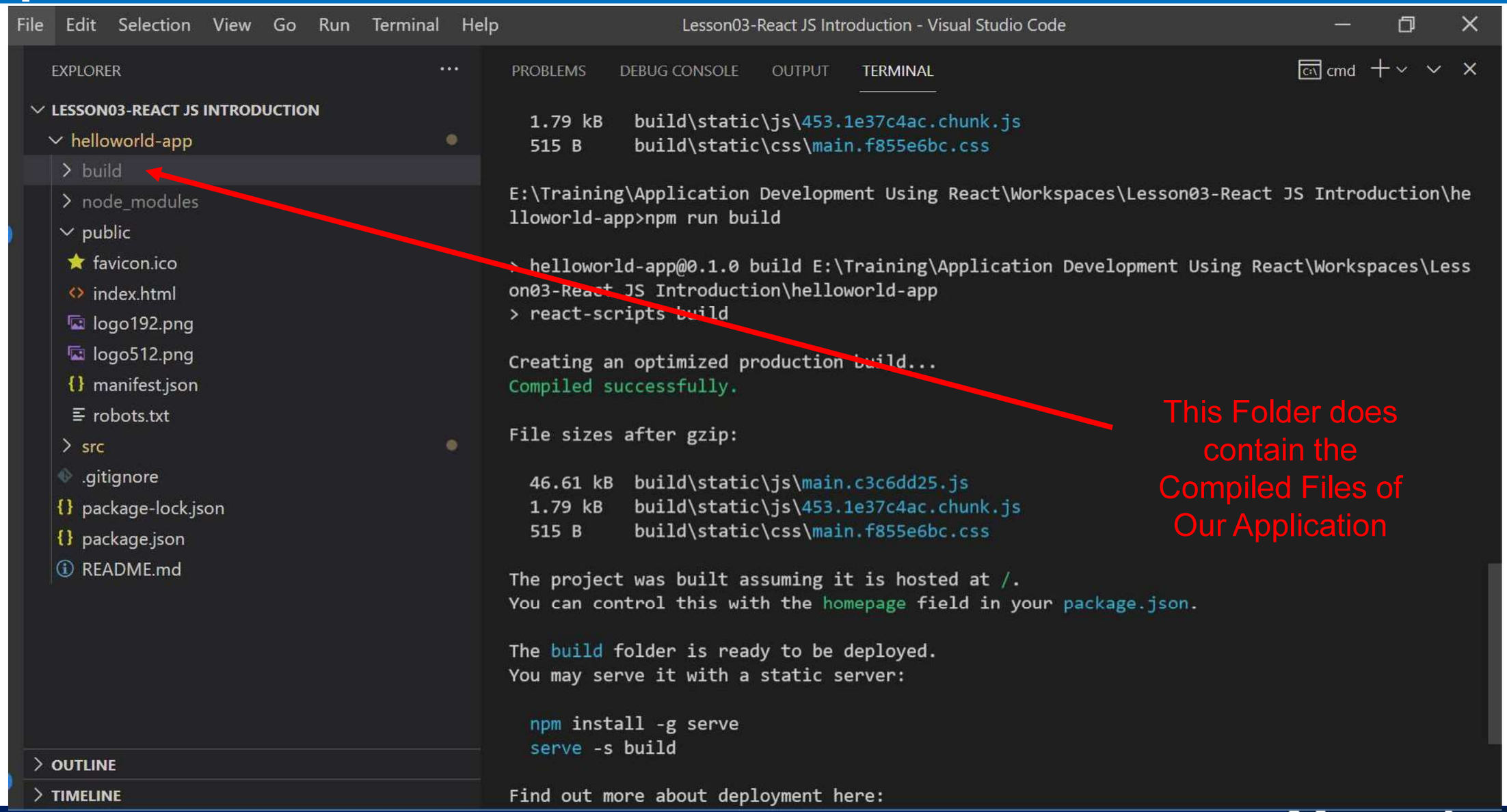
Project Structure [When React Project is Constructed]



Node_modules [Libs]

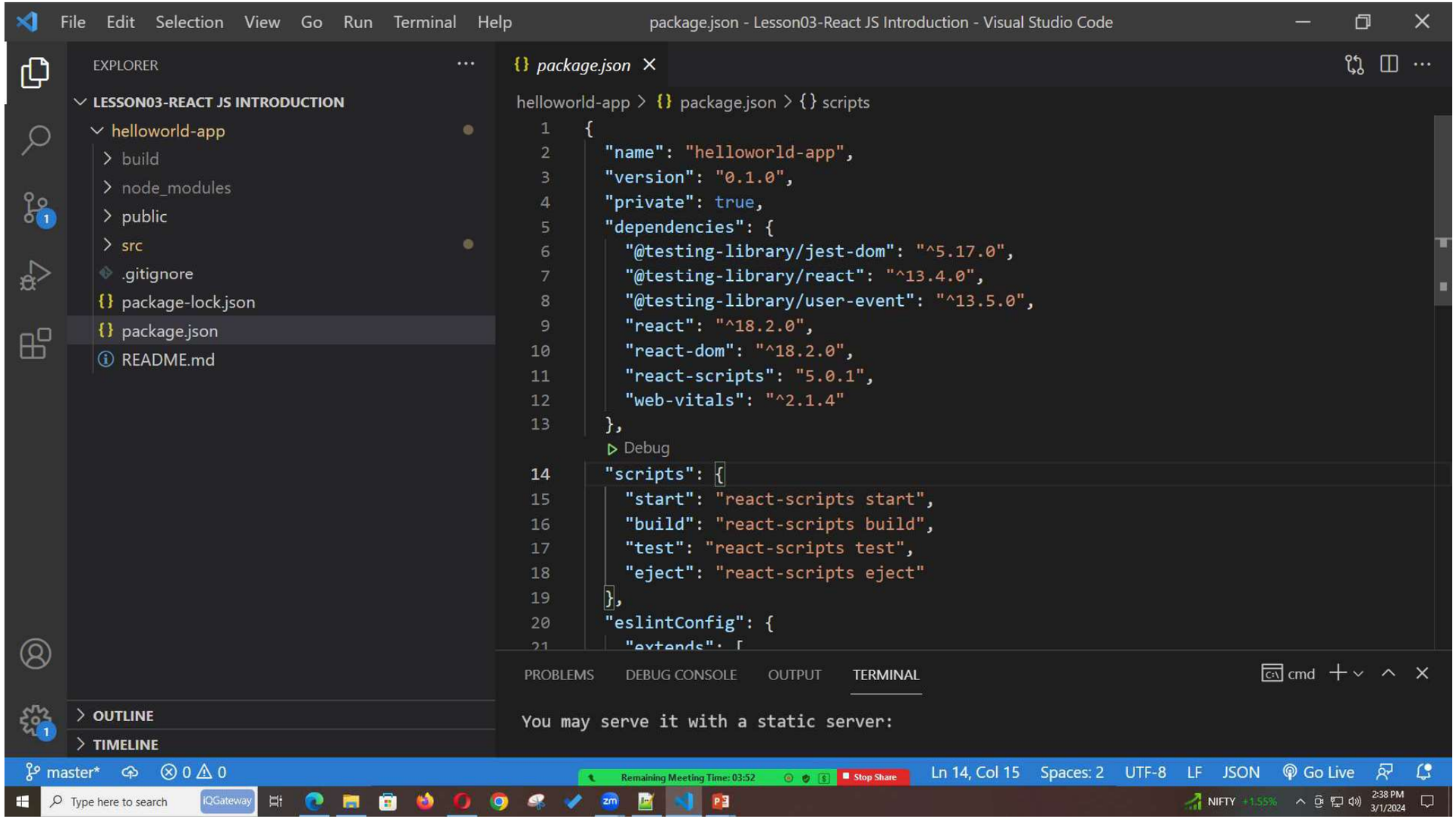


npm run build

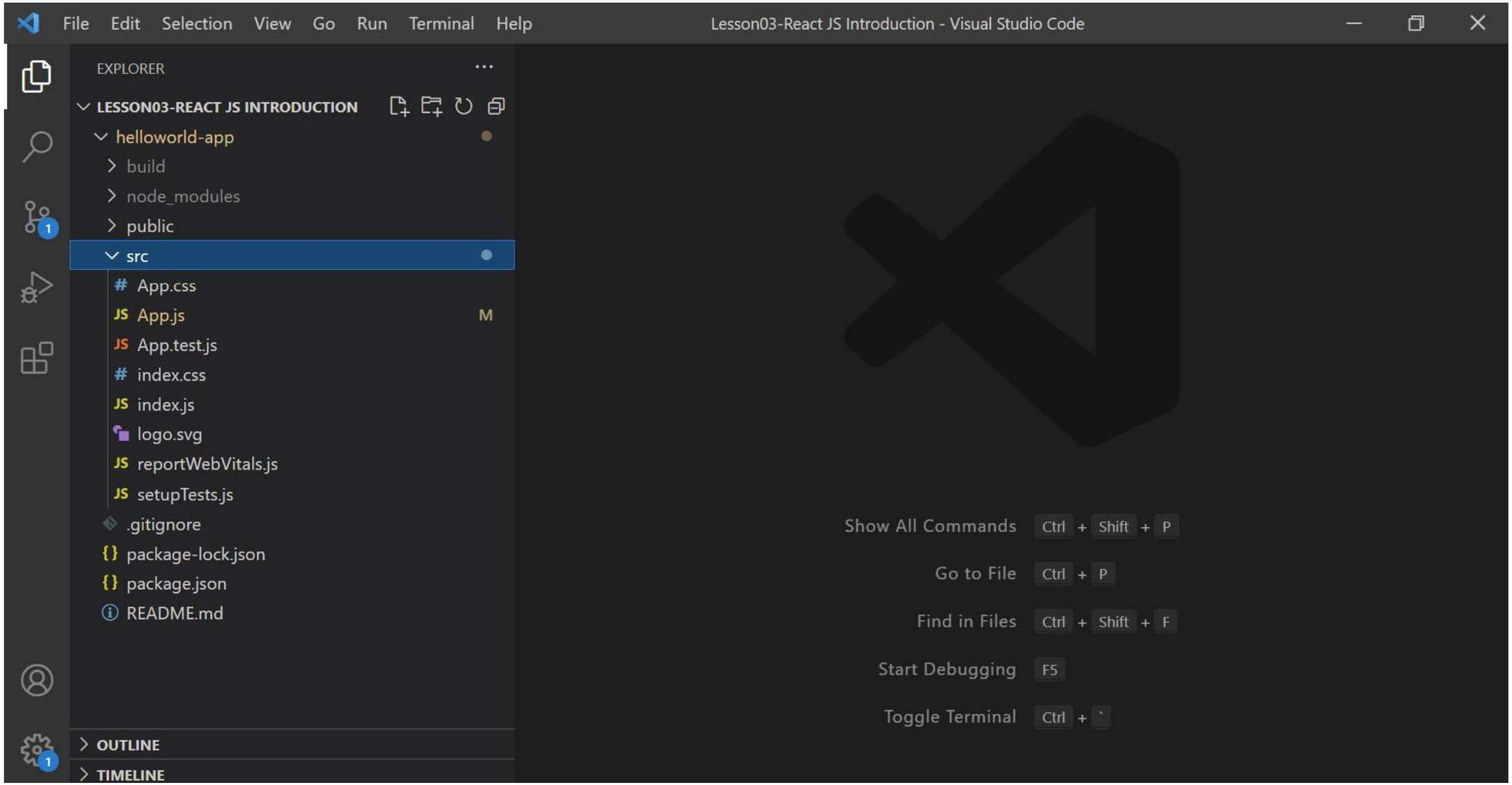


This Folder does contain the Compiled Files of Our Application

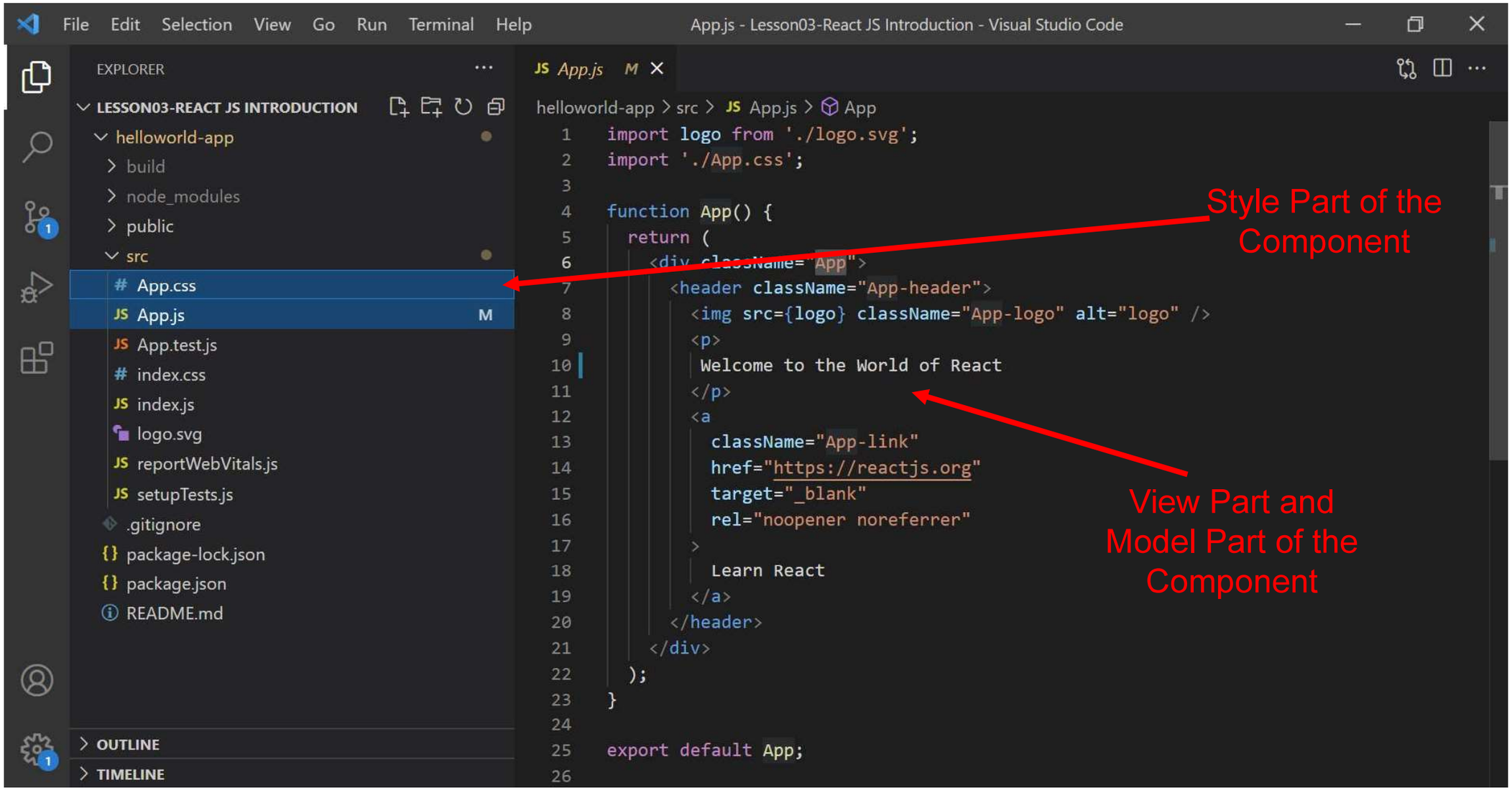
Package .json



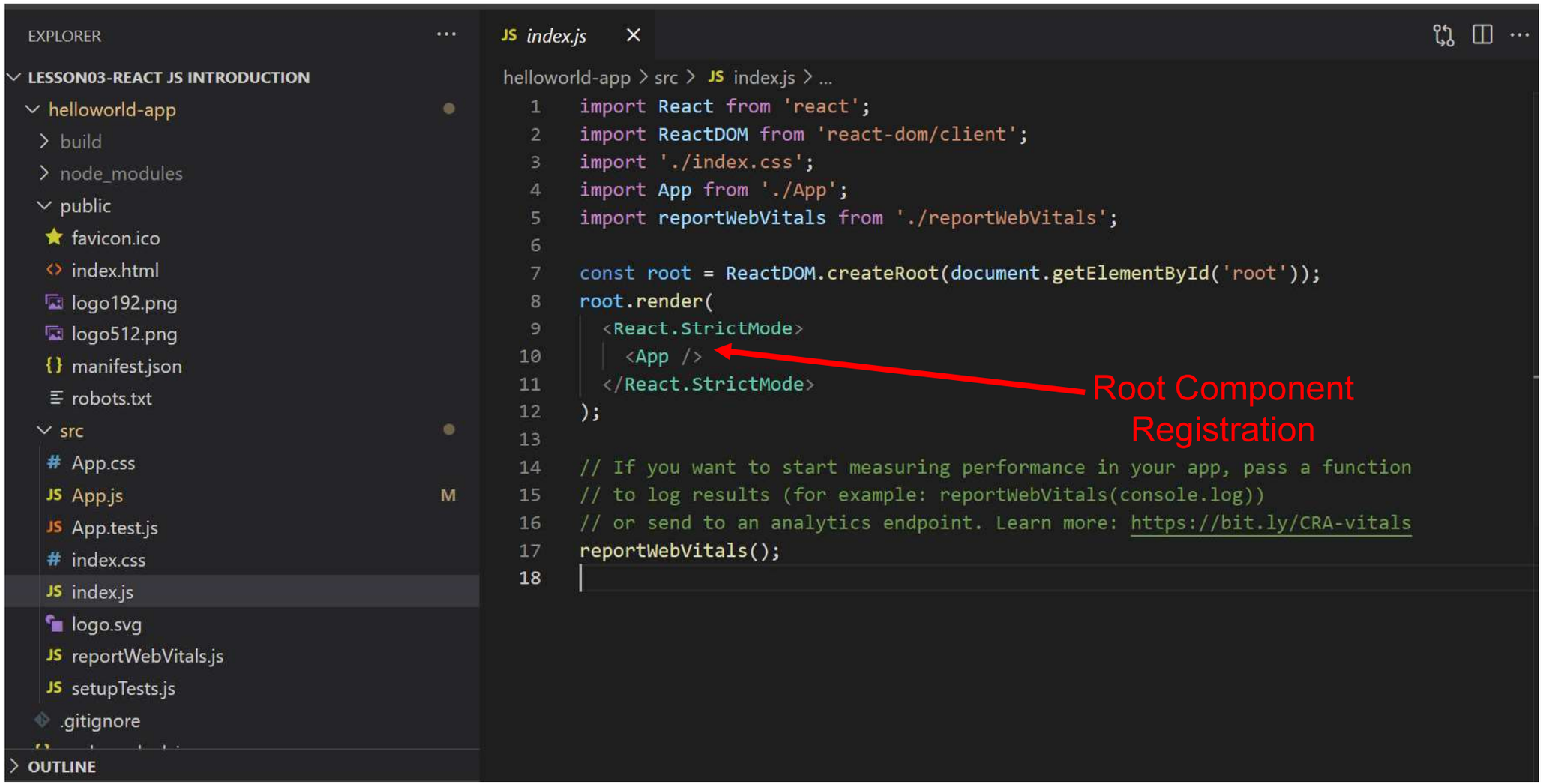
[Src – A Place Where Developer Writes the Code Functionality]



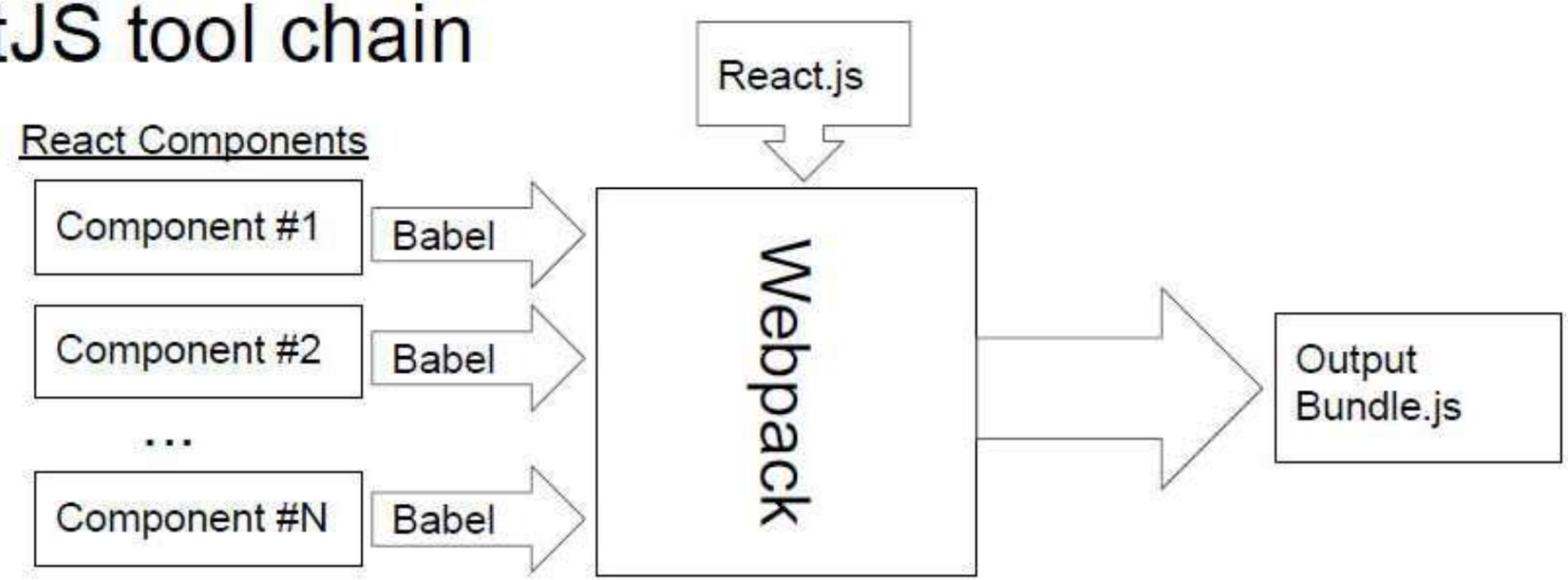
Root Component [App Component]



Registering the Root Component



ReactJS tool chain



Babel - Transpile language features (e.g. ECMAScript, JSX) to basic JavaScript

Webpack - Bundle modules and resources (CSS, images)

Output loadable with single `script` tag in any browser

JavaScript and HTML *in the same file*

Traditional approach

React approach

JSX: the React programming language

```
const first = "Aaron";
const last  = "Smith";

const name = <span>{first} {last}</span>;


const list = (
  <ul>
    <li>Dr. David Stotts</li>
    <li>{name}</li>
  </ul>
);
```

```
const listWithTitle = (
  <>
    <h1>COMP 523</h1>
    <ul>
      <li>Dr. David Stotts</li>
      <li>{name}</li>
    </ul>
  </>
);
```

"React is just JavaScript"

Functional Programming

1. Functions are "first class citizens"
2. Variables are immutable
3. Functions have no side effects

Functional programming

Functions are "first class citizens"

This means functions can be...

1. Saved as **variables**
2. Passed as **arguments**
3. **Returned** from functions

```
let add = function() {
  console.log('Now adding numbers');
  const five = 3 + 2;
};
```

```
function performTask(task) {
  task();
  console.log('Task performed!');
}

performTask(add);
```

```
function foo() {
  return function() {
    console.log('What gets printed?');
  };
}

foo
foo();
foo()();
```

Functional programming

Variables are immutable

```
let a = 4;

a = 2; // Mutates `a`
```

```
let b = [1, 2, 3];

b.push(4); // Mutates `b`

let c = [...b, 4]; // Does not mutate `b`
```

Tip: Use **const** instead of **let** to declare variables!

Functional programming

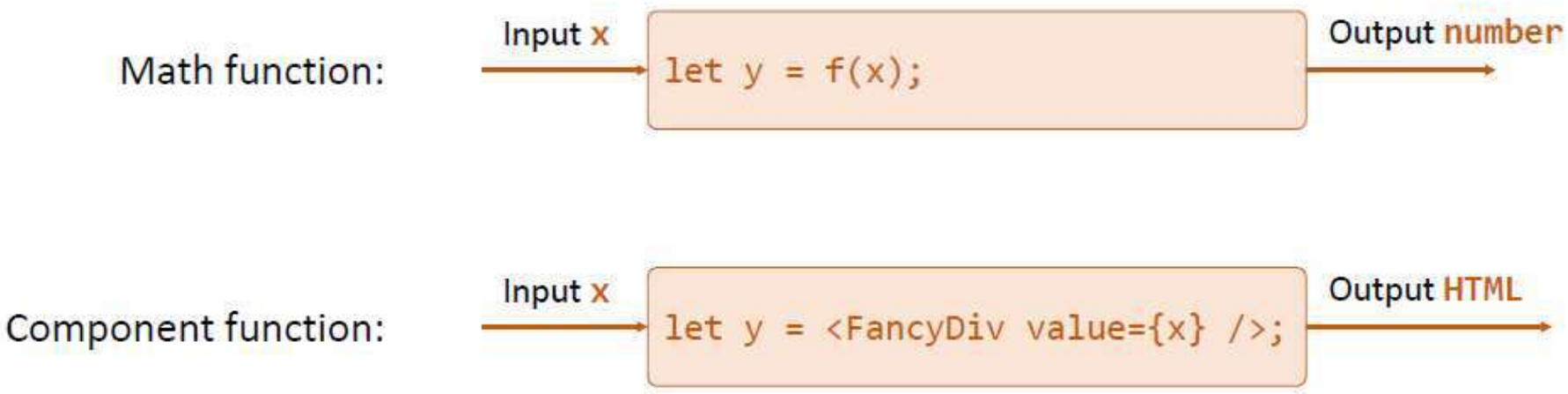
Functions have no side effects

```
const b = [];

function hasSideEffects() {
  b = [0];
}
```

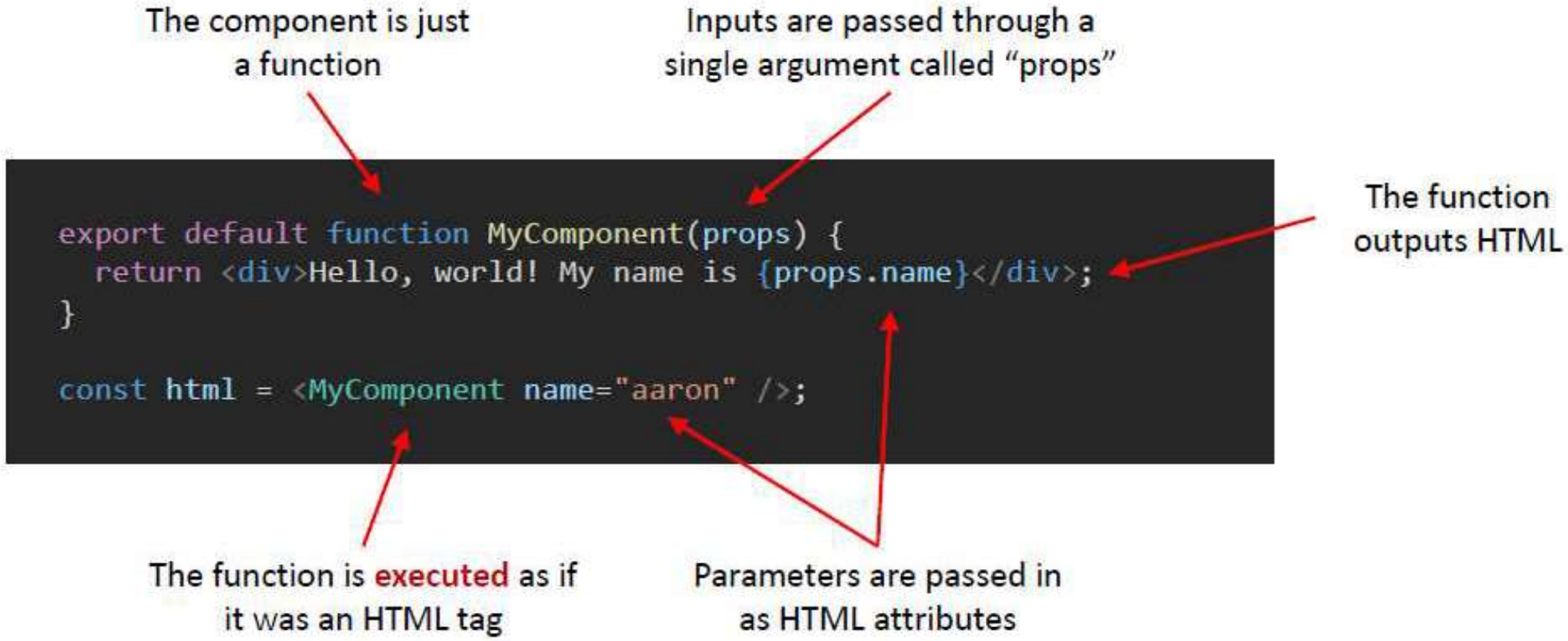
Components

Functions help break your code into small, reusable pieces

Components are functions for user interfaces



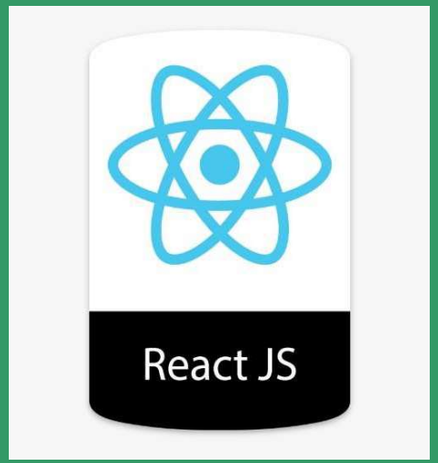
Anatomy of a React component



Component rendering

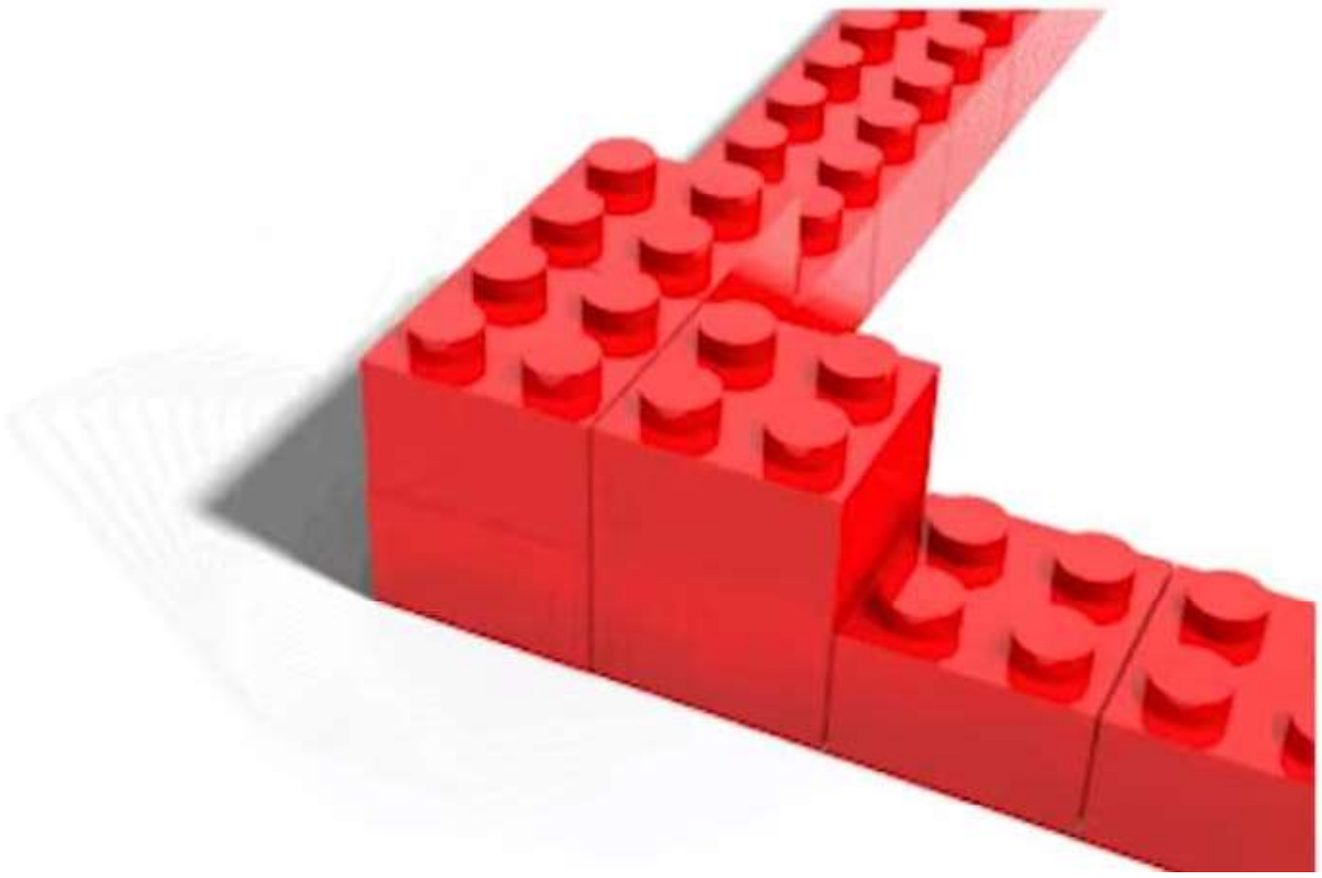
- When a component function **executes**, we say it "**renders**"
- Assume components may re-render at any time

Our job is to ensure that
every time the component re-renders,
the correct output is produced



The Components

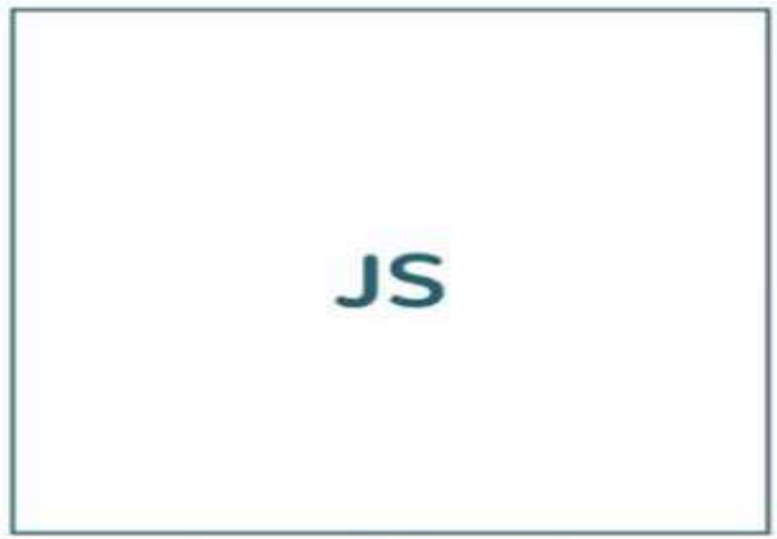
Angular Components are the Building Blocks Like



Cont …

Traditional Developments

- HTML which is the static portion of your application and then you write your JavaScript which is the dynamic portion of the application.
- The JavaScript is kind of embedded into your HTML so when the HTML loads the JavaScript also gets loaded



Show Current Date and Time

This is how We Do in Traditional Application

Add a div and paragraph

Code to get date/time

Get the paragraph DOM element

Update value

How Do We Make it Dynamic

Add a div and paragraph

Add button

Code to get date/time

Get the paragraph DOM element

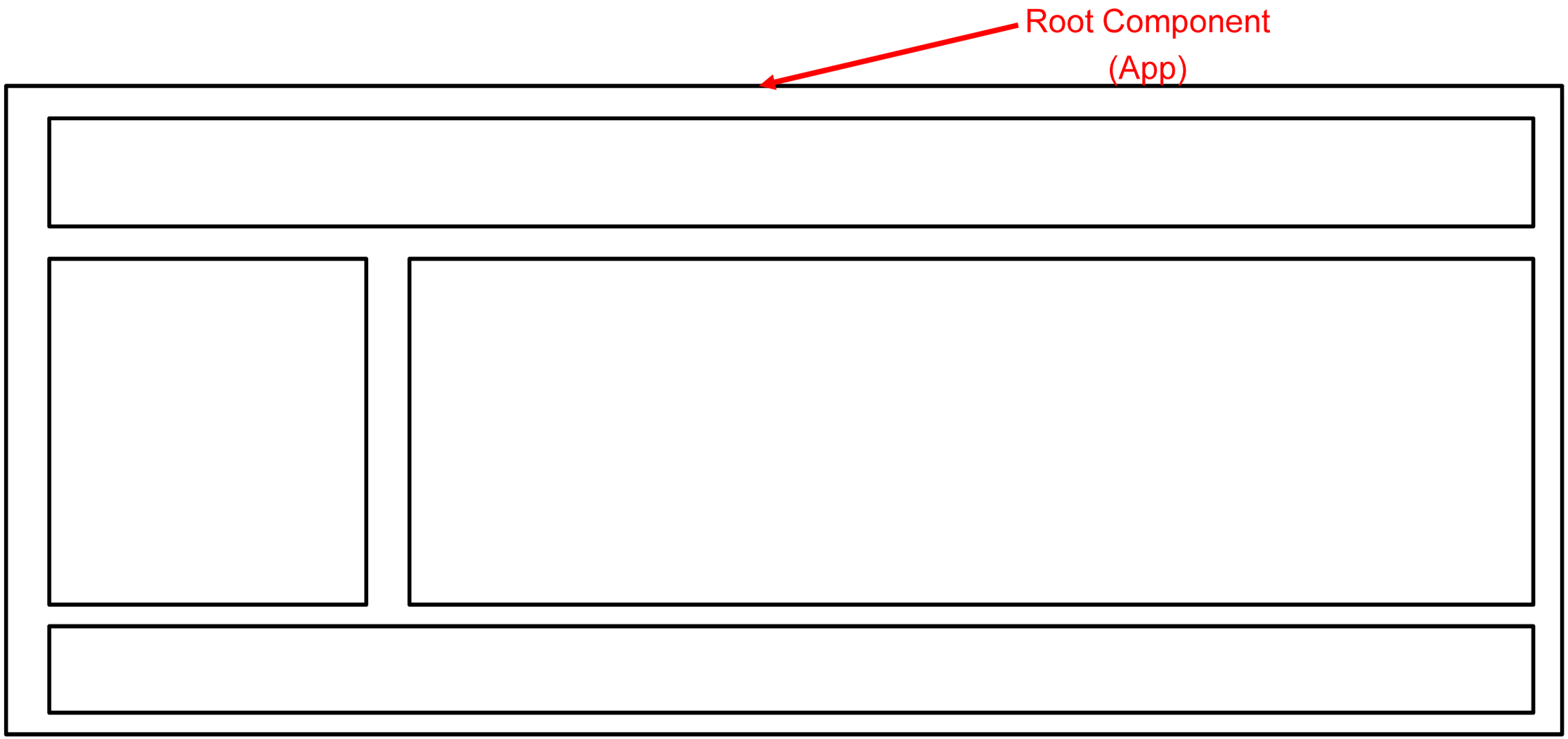
Update value

Code function to handle button click

Component Based Approach

In React

- In React applications we do not have this implicit divide at least in the mentioned module we don't have to think about it when we creating something
  - What is the HTML side
  - What is the JavaScript side
- In React We think about is components
- React has a component based approach to developing web applications


Root Component
(App)

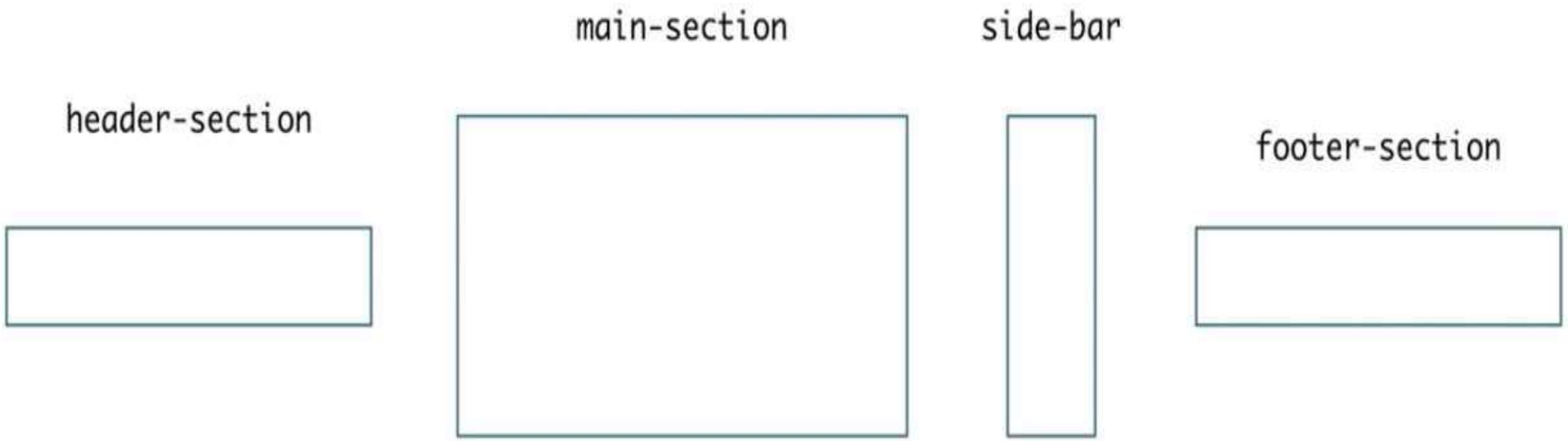
- So every significant portion of real estate on our page which can be self-sufficient which knows what to do with that area can be split and created as a component
- This is the approach we use in developing an React Application.
- We are going to create this entity which contains HTML and JavaScript together in it right so this is a self-sufficient piece of web application that can be plugged in somewhere else and that knows what to do.

- Creating a component assign a name to that component write assign a selector that selector is what a consumer can use to call and render that component


```
header-section
```

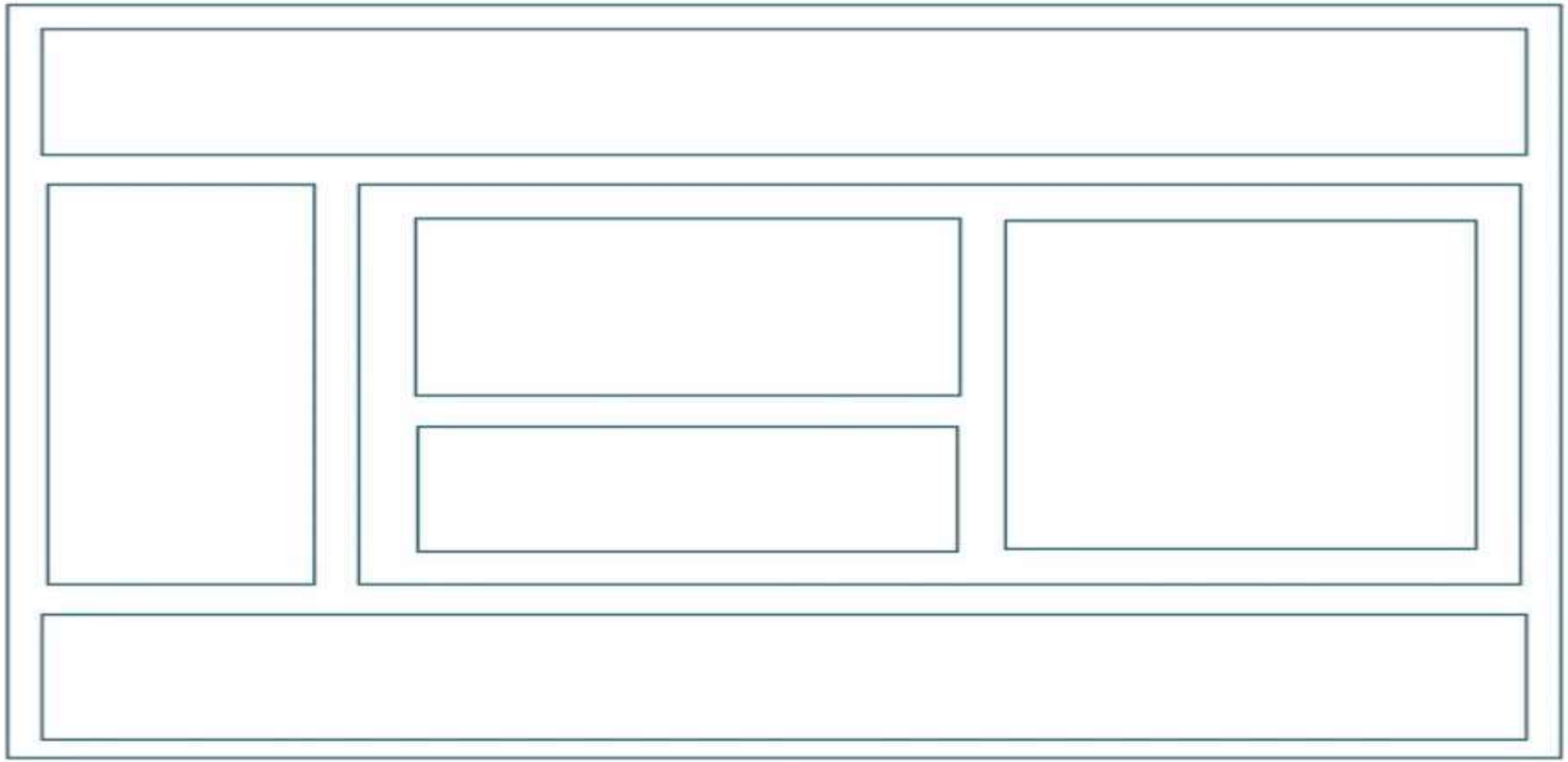
```
<header-section></header-section>
```

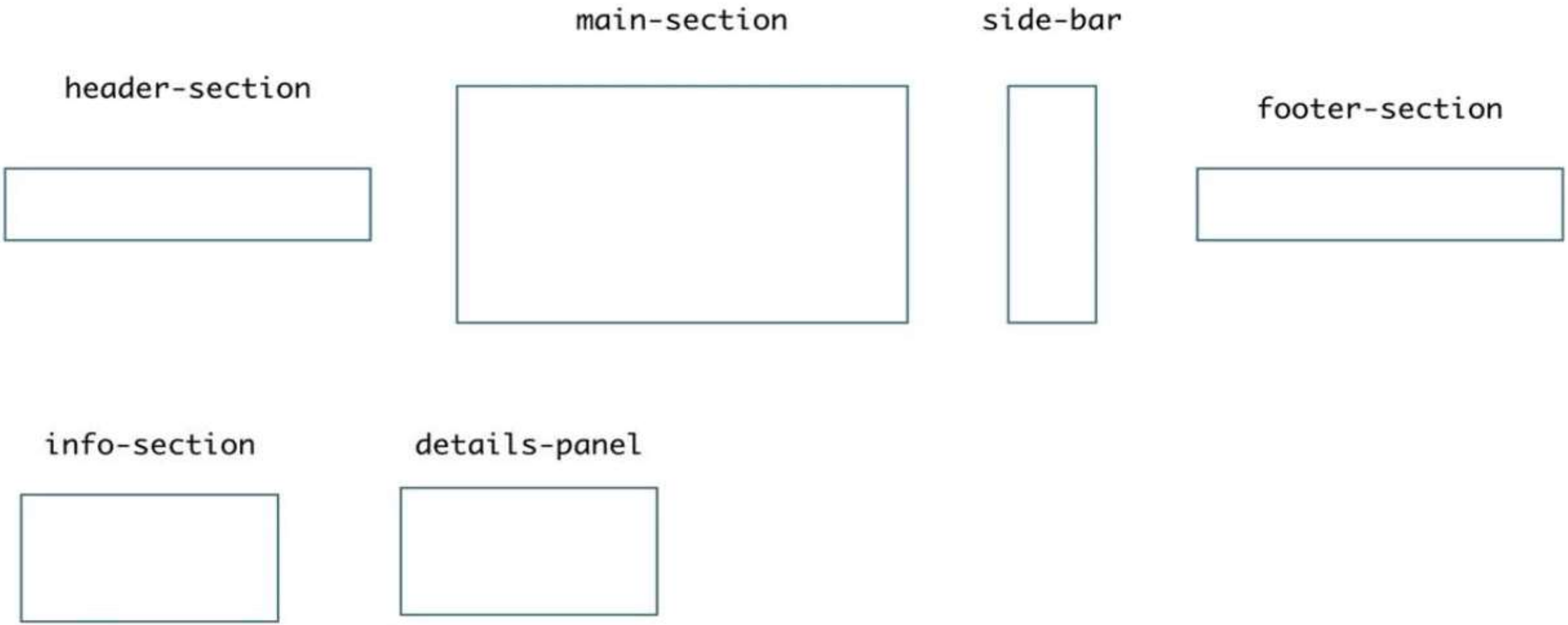
Component Based Development



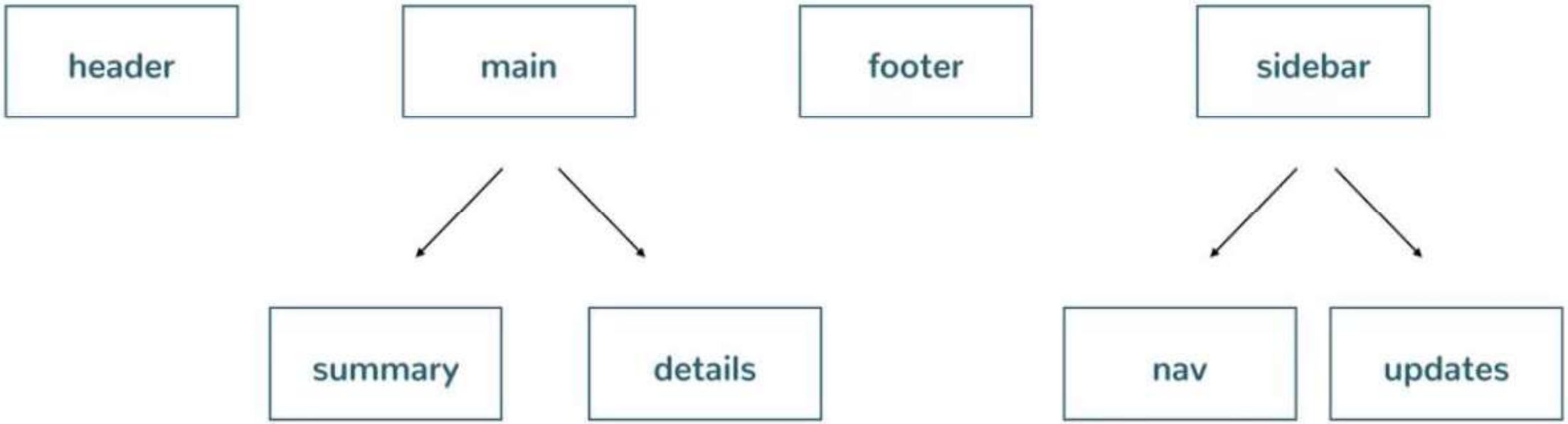
Components on a Page

```
<header-section></header-section>

<main-section></main-section>

<side-bar></side-bar>

<footer-section></footer-section>
```

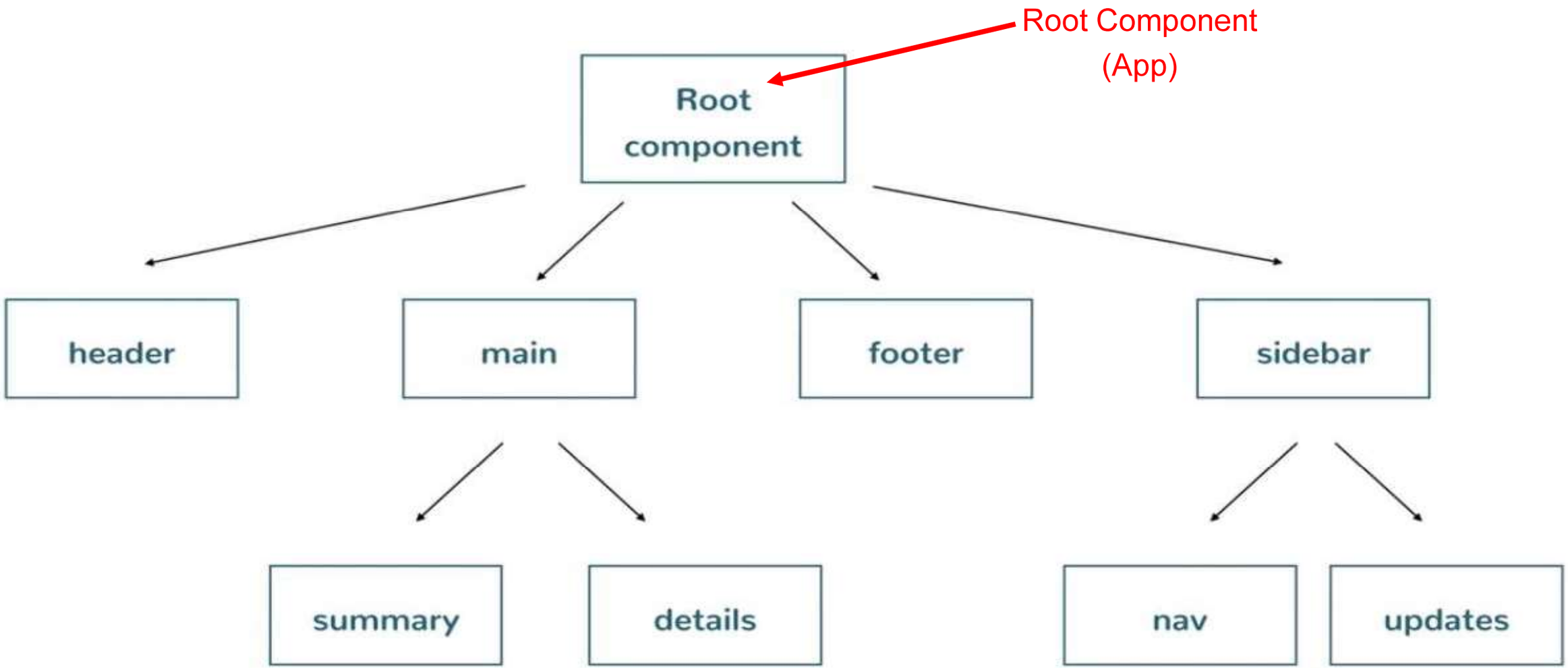



main-section
side-bar
header-section
footer-section
info-section
details-panel

Component Tree Structure



Every React Application as a Root Component Which Holds Other Comps



Component in Code

JavaScript File

Component Code

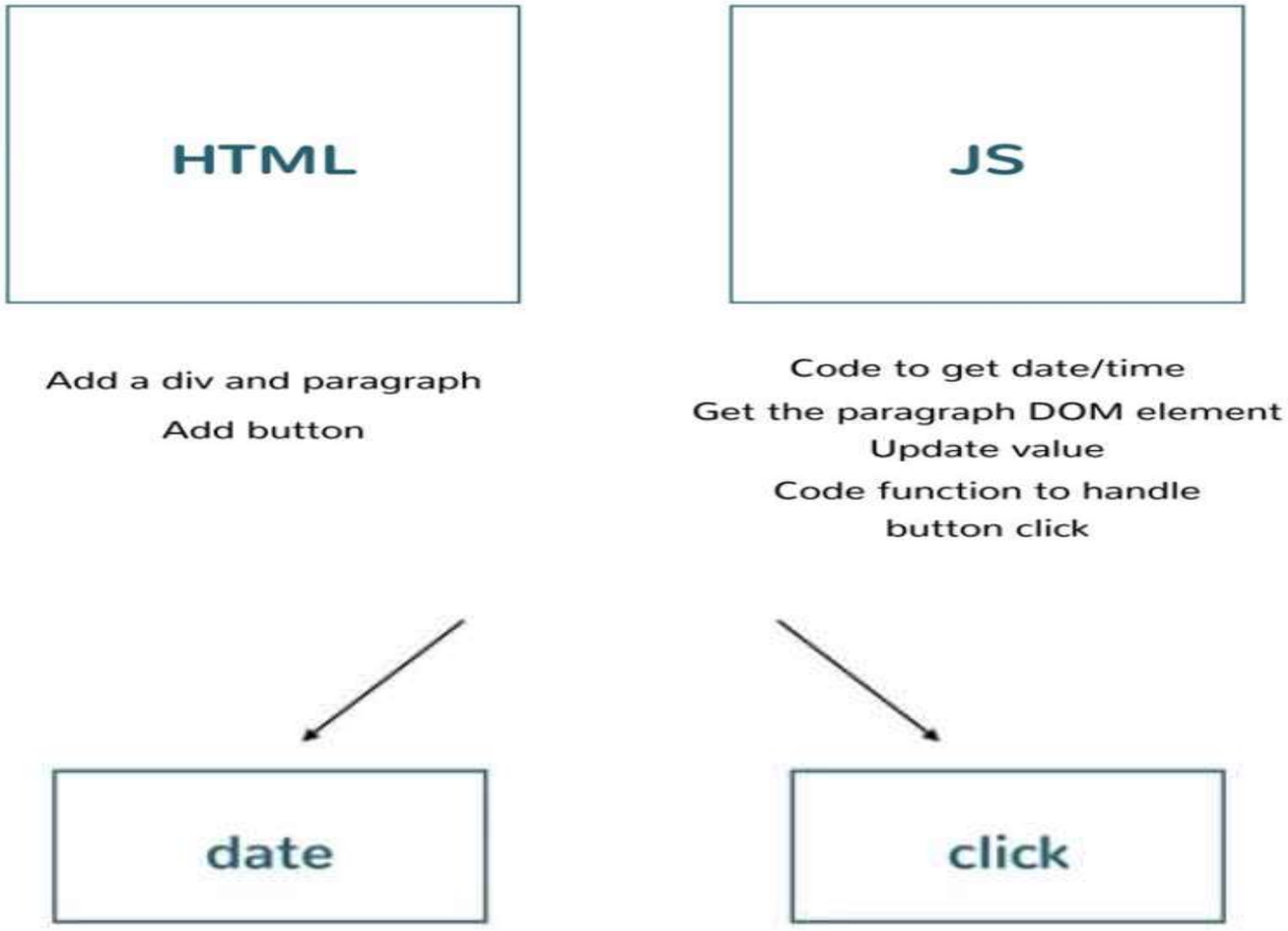
App.js

App Component Code

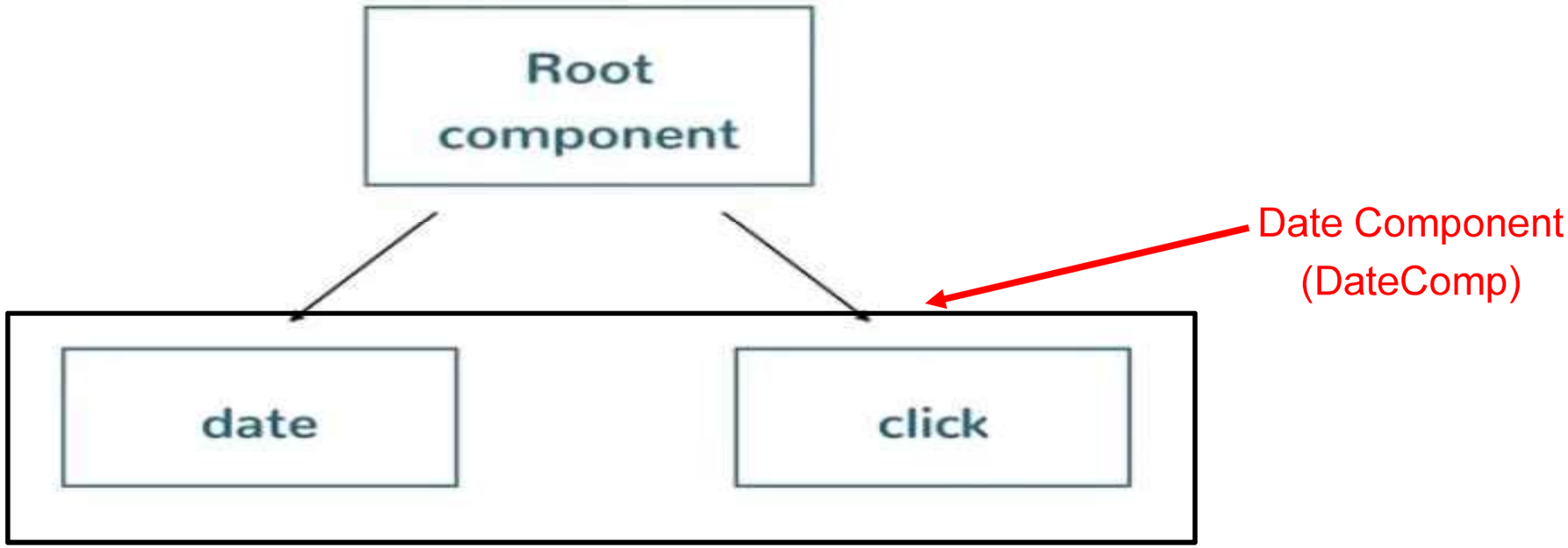
Date.js

Date Component Code

How it Alters Our Application



Finally Our React Application Looks Like this



"In React, everything is a component"

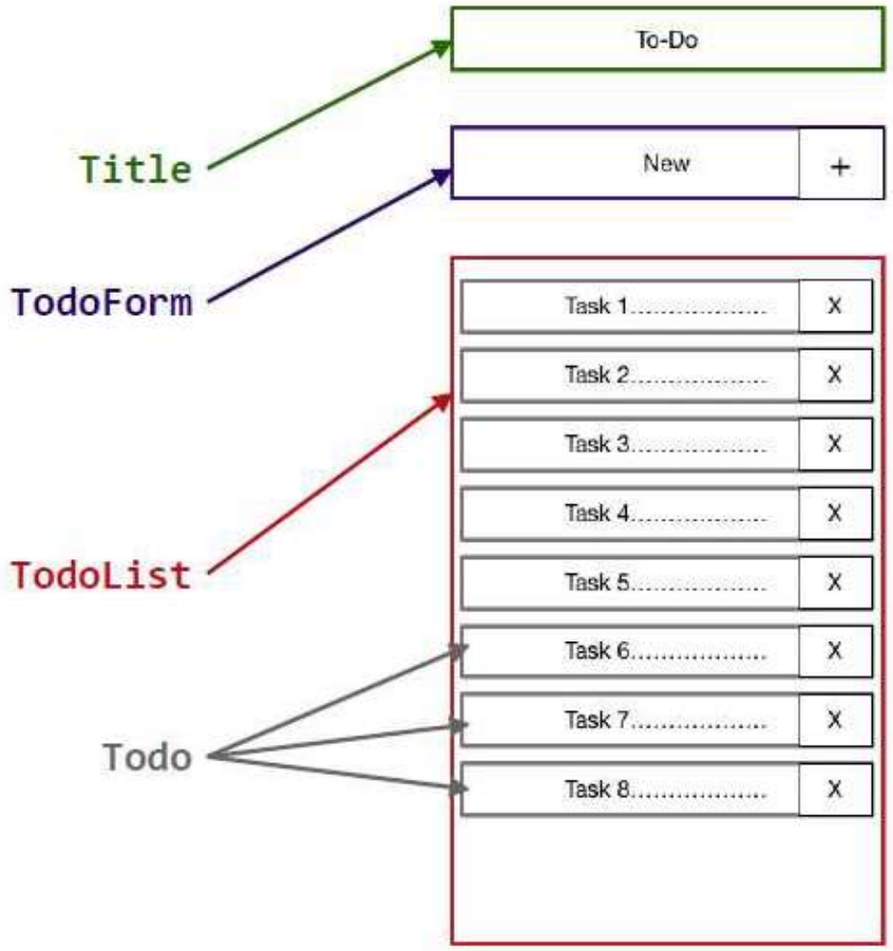
Todo application

Big idea:

- A digital to-do list

First step:

- mockup / wireframe



Creating a new React app

Creating a new React app is simple!

1. Install Node.js
2. Run: `npx create-react-app app-name`
3. New app created in folder: `./app-name`

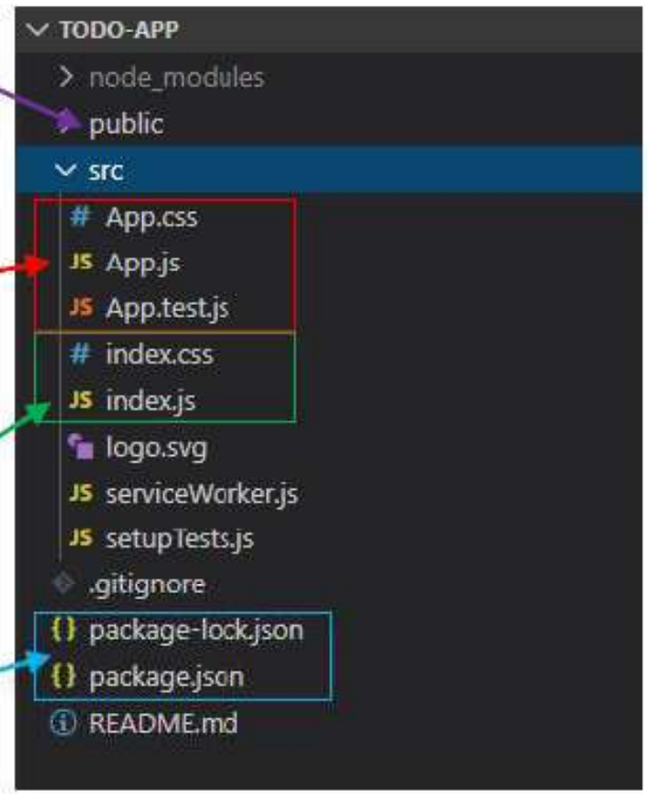
Anatomy of a new React app

public holds the initial html document and other static assets

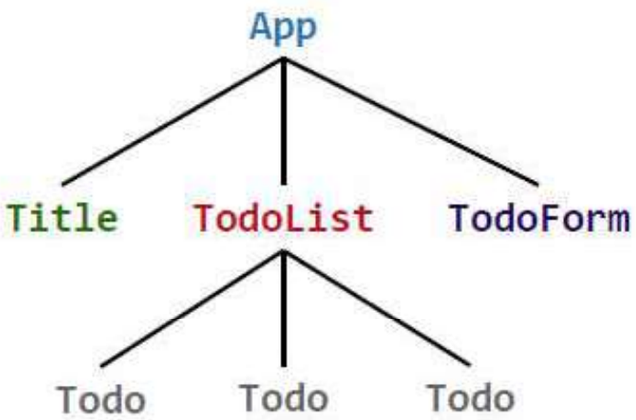
App is a boilerplate starter component

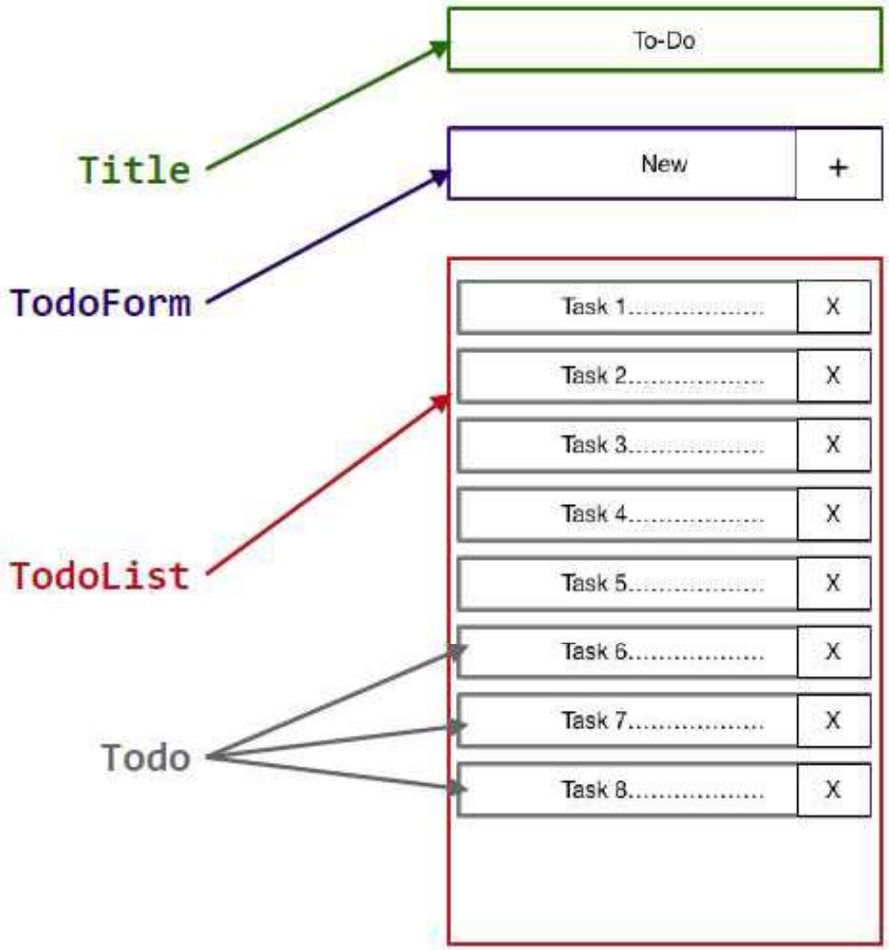
index.js binds React to the DOM

package.json configures npm dependencies



Component Hierarchy





reactApp.js - Render element into browser DOM

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

import ReactAppView from './components/ReactAppView';

let viewTree = React.createElement(ReactAppView, null);
let where = document.getElementById('reactapp');

ReactDOM.render(viewTree, where);
```

ES6 Modules - Bring in React and web app React components.

Renders the tree of React elements (single component named **`ReactAppView`**) into the browser's DOM at the div with id=reactapp.

Component Types

Stateless Functional Component

JavaScript Functions

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

Stateful Class Component

Class extending Component Class

```
class Welcome extends React.Component {

  render() {
    return <h1> Welcome , {this.props.name} </h1>;
  }

}
```

components/ReactAppView.js - ES6 class definition

```
import React from 'react';

class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    ...
  }
  render() { ...
};

export default ReactAppView;
```

Inherits from React.Component. props is set to the attributes passed to the component.

Require method render() - returns React element tree of the Component's view.

ReactAppView render() method

```
<div>
   <label>Name: </label>
   <input type="text" ... />
   <h1>Hello {this.state.yourName}!</h1>
</div>
```

```
render() {
    let label = React.createElement('label', null,'Name: ');
    let input =  React.createElement('input',
           { type: 'text', value: this.state.yourName,
               onChange: (event) => this.handleChange(event) });
    let h1 = React.createElement('h1',  null,
                        'Hello ', this.state.yourName, '!');
    return React.createElement('div', null, label, input, h1);
}
```

Returns element tree with `div` (`label`, `input`, and `h1`) elements

ReactAppView render() method w/o variables

```
render() {
    return React.createElement('div', null,
        React.createElement('label', null,'Name: '),
        React.createElement('input',
           { type: 'text', value: this.state.yourName,
             onChange: (event) => this.handleChange(event) }),
        React.createElement('h1',  null,
                'Hello ', this.state.yourName, '!')
      );
}
```

JSX templates must return a valid `children` param

- Templates can have JavaScript scope variables and expressions
  - `<div>{foo}</div>`
    - Valid if `foo` is in scope (i.e. if foo would have been a valid function call parameter)
  - `<div>{foo + 'S' + computeEndingString()}</div>`
    - Valid if `foo` & `computeEndString` in scope
- Template must evaluate to a value
  - `<div>{if (useSpanish) { ... } }</div>` - Doesn't work: `if` isn't an expression
  - Same problem with "for loops" and other JavaScript statements that don't return values
- Leads to contorted looking JSX: Example: Anonymous immediate functions
  - `<div>{ (function() { if ...; for ..; return val;})() }</div>`

Use JSX to generate calls to createElement

```
render() {
   return (
      <div>
        <label>Name: </label>
        <input
              type="text"
              value={this.state.yourName}
              onChange={(event) => this.handleChange(event)}
          />
        <h1>Hello {this.state.yourName}!</h1>
     </div>
    );
}
```

- JSX makes building tree look like templated HTML embedded in JavaScript.

Conditional render in JSX

- Use JavaScript Ternary operator (?:)

```
<div>{this.state.useSpanish ? <b>Hola</b> : "Hello"}</div>
```

- Use JavaScript variables

```
let greeting;
const en = "Hello";  const sp = <b>Hola</b>;
let {useSpanish} = this.prop;
if (useSpanish) {greeting = sp} else {greeting = en};
```

```
<div>{greeting}</div>
```

Iteration in JSX

- Use JavaScript array variables

```
let listItems = [];
for (let i = 0; i < data.length; i++) {
   listItems.push(<li key={data[i]}>Data Value {data[i]}</li>);
}
return <ul>{listItems}</ul>;
```

- Functional programming

```
<ul>{data.map((d) => <li key={d}>Data Value {d}</li>)}</ul>
```

key= attribute improves efficiency of rendering on data change

Styling with React/JSX - lots of different ways

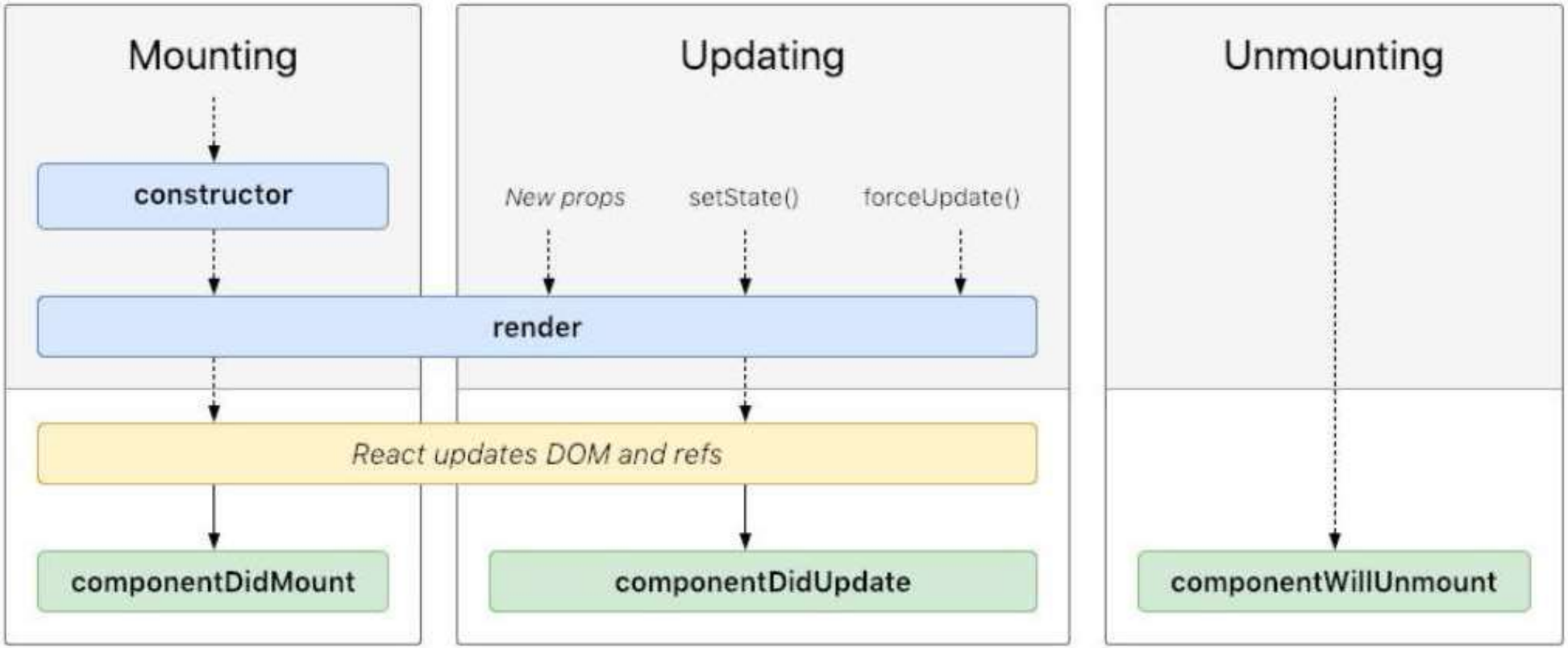
```
import React from 'react';
import './ReactAppView.css';
class ReactAppView extends React.Component {
  ...
render() {
   return (
      <span className="cs142-code-name">
        ...
      </span>
    );
```

Webpack can import CSS style sheets:

```
.cs142-code-name {
  font-family: Courier New, monospace;
}
```

Must use `className=` for HTML `class=` attribute (JS keyword conflict)

Component lifecycle and methods



Example of lifecycle methods - update UI every 2s

```
class Example extends React.Component {
     ...
    componentDidMount() {    // Start 2 sec counter
       const incFunc =
         () => this.setState({ counter: this.state.counter + 1 });
       this.timerID = setInterval(incFunc, 2 * 1000);
    }

    componentWillUnmount() {  // Shutdown timer
       clearInterval(this.timerID);
    }
    ...
```

Stateless Components

- React Component can be function (not a class) if it only depends on props

```
function MyComponent(props) {
  return <div>My name is {props.name}</div>;
}
```

Or using destructuring...

```
function MyComponent({name}) {
  return <div>My name is {name}</div>;
}
```

- Much more concise than a class with `render` method
  - But what if you have one bit of state…

Component state and input handling

```
import React from 'react';

class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
  }
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
  ....
```

Make `<h1>Hello {this.state.yourName}!</h1>` work

- Input calls to `setState` which causes React to call render() again

One way binding: Type 'D' Character in input box

- JSX statement: `<input type="text" value={this.state.yourName}`
  `onChange={(event) => this.handleChange(event)} />`

  Triggers `handleChange` call with `event.target.value` == "D"

- handleChange - `this.setState({yourName: event.target.value});`

  `this.state.yourName` is changed to "D"

- React sees state change and calls render again:
- Feature of React - highly efficient re-rendering

Calling React Components from events: A problem

```
class ReactAppView extends React.Component {
    ...
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
  ...
}
```

Understand why:

```
<input type="text" value={this.state.yourName} onChange={this.handleChange} />
```

Doesn't work!

Calling React Components from events workaround

- Create instance function bound to instance

```
class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
```

Calling React Components from events workaround

- Using public fields of classes with arrow functions

```
class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
  }
  handleChange = (event) => {
    this.setState({ yourName: event.target.value });
  }
  ...
```

Calling React Components from events workaround

- Using arrow functions in JSX

```
class ReactAppView extends React.Component {
  ...
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
  render() {
    return (
        <input type="text" value={this.state.yourName}
            onChange={(event) => this.handleChange(event)} />
     );
  }
```

A digression: `camelCase` vs `dash-case`

Word separator in multiword variable name

- Use dashes: `active-buffer-entry`
- Capitalize first letter of each word: `activeBufferEntry`

Issue: HTML is case-insensitive but JavaScript is not.
ReactJS's JSX has HTML-like stuff embedded in JavaScript.

ReactJS: Use camelCase for attributes

AngularJS: Used both: dashes in HTML and camelCase in JavaScript!

Special list key property

- **Situation:** Display a **dynamic array of elements**
- Must specify a special "**key**" property for each element
- The key of an item **uniquely identifies it**
- Used by React internally for **render optimization**
- Can be any unique value (string or number)

What are hooks?

Hooks: Special functions that allow developers to hook into **state** and **lifecycle** of React components.

State: One or more data values associated with a React component instance.

Lifecycle: The events associated with a React component instance (create, render, destroy, etc).

Built-in hooks:

- `useState`
- `useEffect`
- `useReducer`
- `useMemo`
- `useRef`
- `useCallback`

React Hooks - Add state to stateless components

- Inside of a "stateless" component add state: `useState(initialStateValue)`
  - `useState` parameter: `initialStateValue` - the initial value of the state
  - `useState` return value: An two element polymorphic array
    - 0th element - The current value of the state
    - 1st element - A set function to call (like `this.setState`)
- Example: a bit of state:

```
const [bit, setBit] = useState(0);
```

- How about lifecycle functions (e.g. componentDidUpdate, etc.)?
  - `useEffect(lifeCycleFunction, dependency array)`
    - `useEffect` parameter `lifeCycleFunction` - function to call when something changes

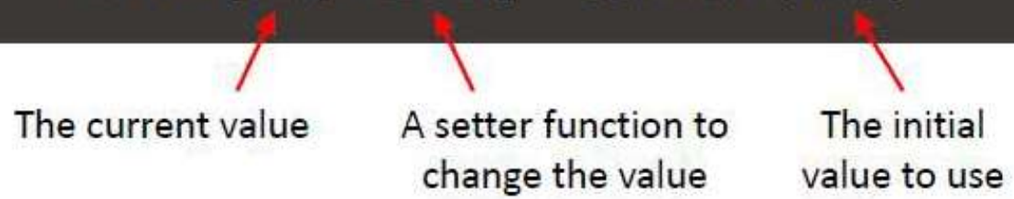
First React hook: useState

Purpose:

1. Remember values internally when the component re-renders
2. Tell React to re-render the component when the value changes

Syntax:

```
const [val, setVal] = useState(100);
```



React Hooks Example - useState

```
import React, { 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>
  );
}
```

React Hooks Example - useEffect Model fetching

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

Predicting component re-rendering

A component will only re-render when…

1. A value inside **`props`** changes

   – or –

2. A **`useState`** setter is called

This means all data values displayed in the HTML should depend on either **`props`** or **`useState`**

Second React hook: `useEffect`

Purpose:

Act as an **observer**, running code in response to value changes

Syntax:

```
useEffect(() => {
  console.log(`myValue was changed! New value: ${myValue}`);
}, [myValue]);
```

A list of values such that changes should trigger this code to run

The code to run when values change

Communicating between React components

- Passing information from parent to child: Use props (attributes)

```
<ChildComponent param={infoForChildComponent} />
```

- Passing information from child to parent: Callbacks

```
this.parentCallback = (infoFromChild) =>
        { /* processInfoFromChild */};

<ChildComponent callback={this.parentCallback}> />
```

- React Context (https://reactjs.org/docs/context.html)
  - Global variables for subtree of components

Building a React project

- When you're ready to launch your app, run this command:

```
npm run build
```

- This bundles your app into CSS/JS/HTML files and puts them in the **/build** folder
- These files can be served from an AWS S3 bucket

3rd party components and libraries

- React-Router
- Redux
- Material-UI
- Bootstrap
- Font-Awesome
- SWR