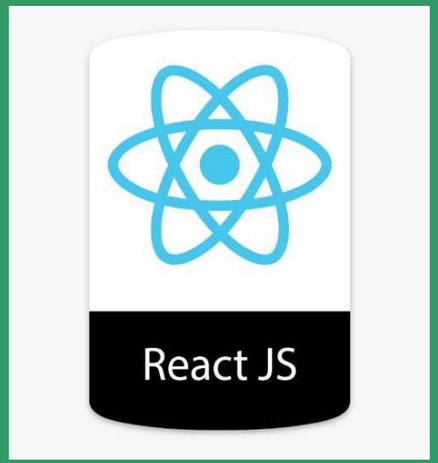
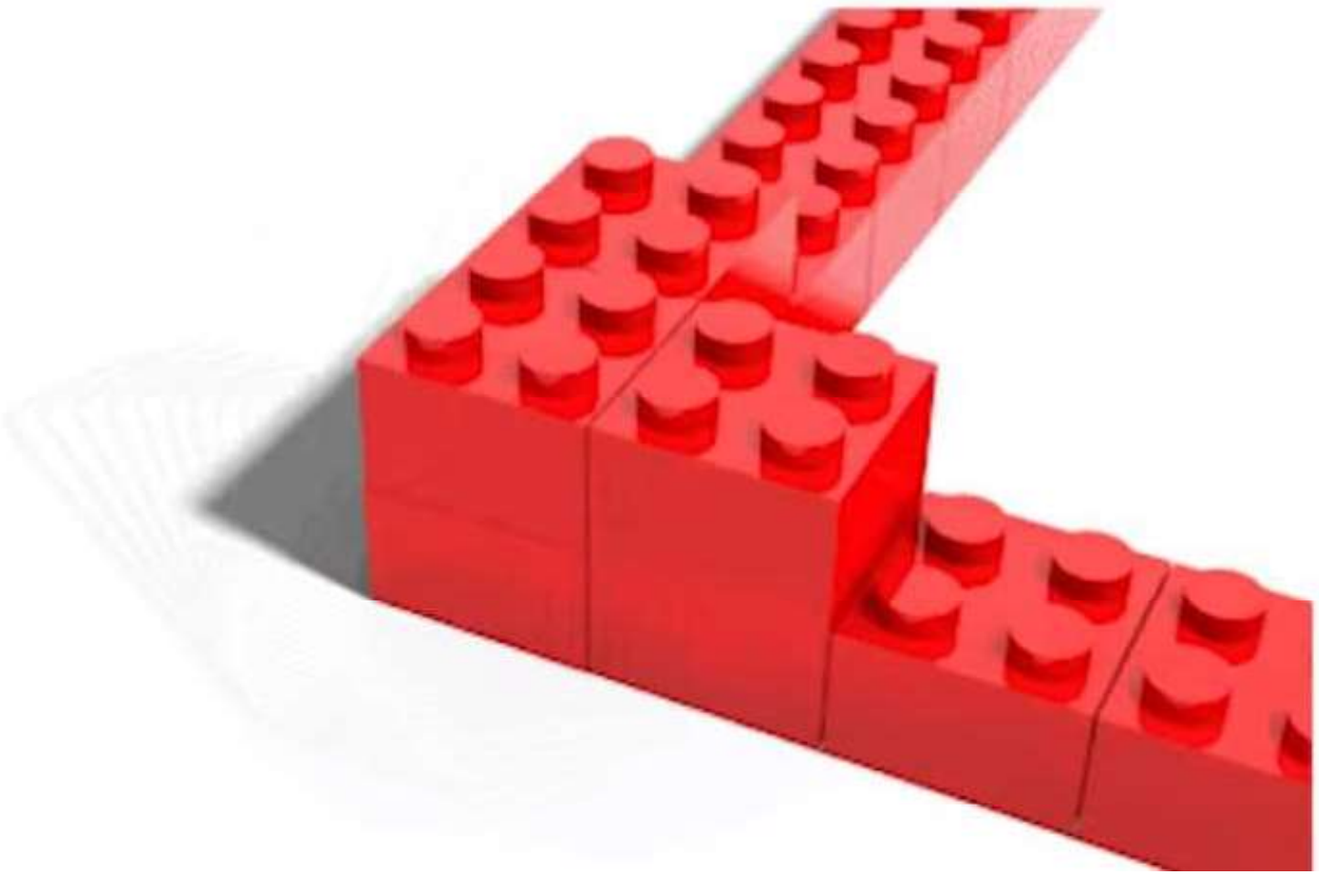
2

React JS



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

HTML

Add a div and paragraph

JS

Code to get date/time

Get the paragraph DOM element

Update value

How Do We Make it Dynamic

HTML

Add a div and paragraph

Add button

JS

Code to get date/time

Get the paragraph DOM element

Update value

Code function to handle button click

Component Based Approach

In Angular

- In angular 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 Angular We think about is components
- Angular has a component based approach to developing web applications

header

- 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 angular 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.

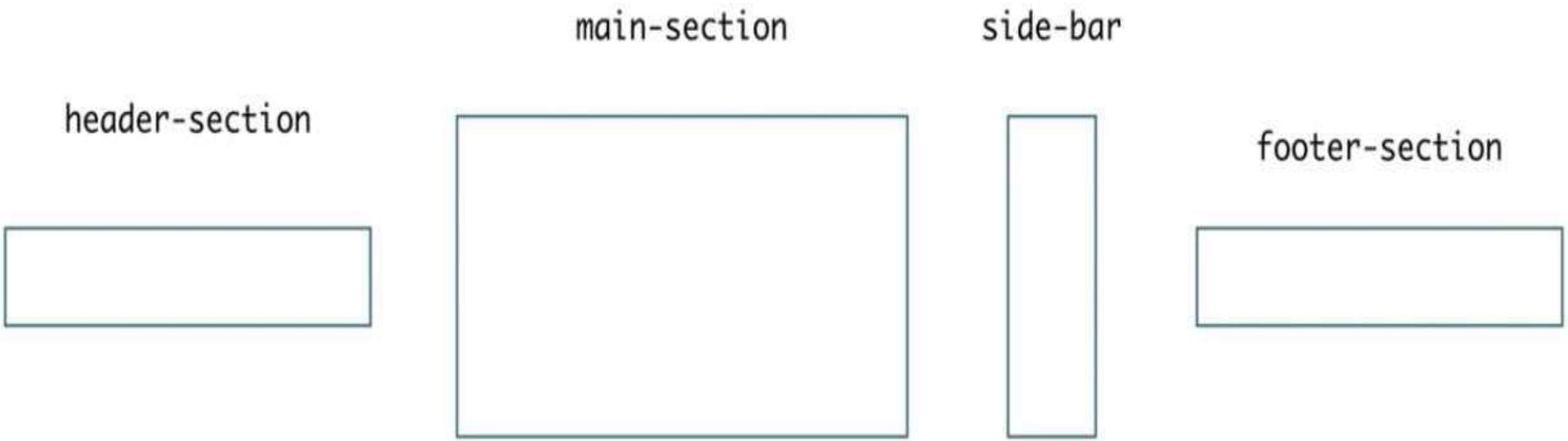
- Creating a component assign a name to that component write assign a selector that selector is what a consumer can use to call and render that component

header

header-section

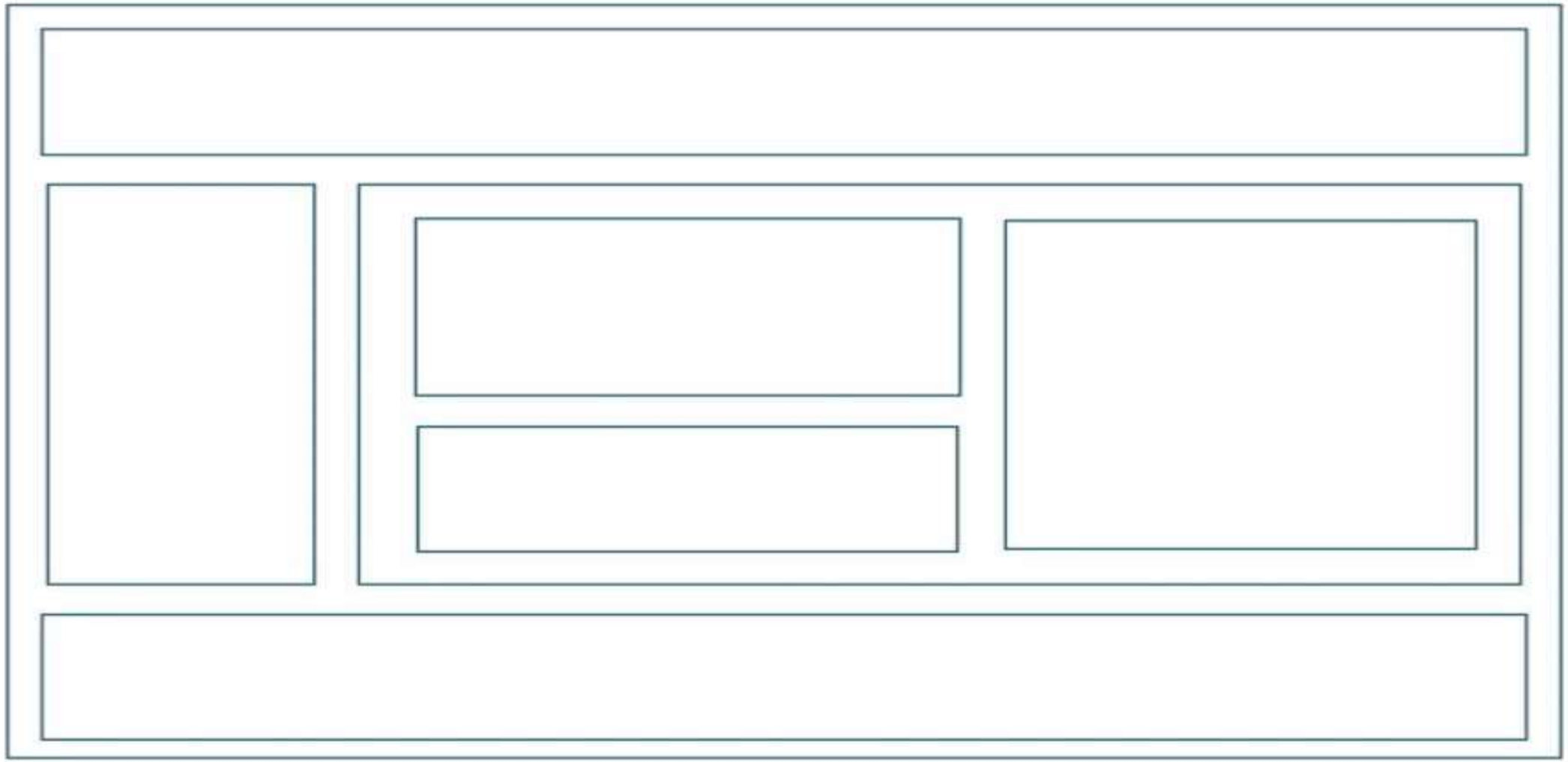
```
<header-section></header-section>
```

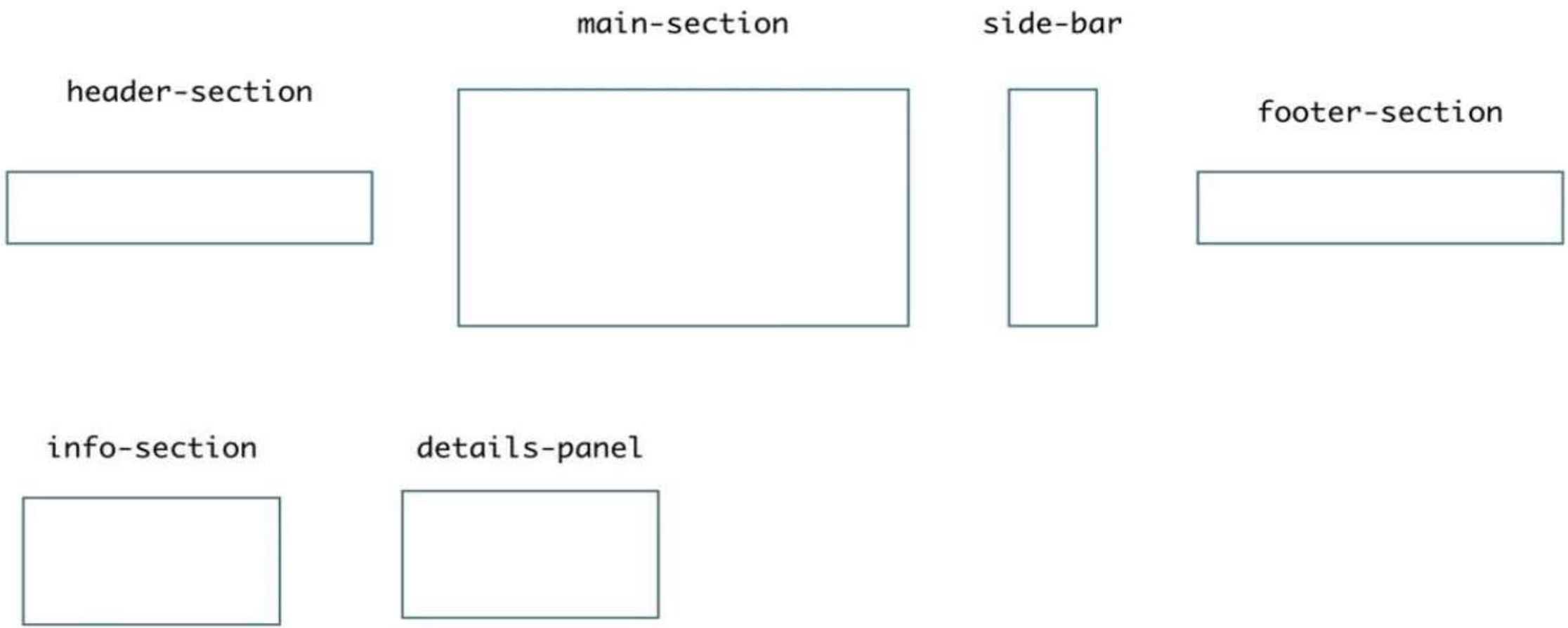
Component Based Development



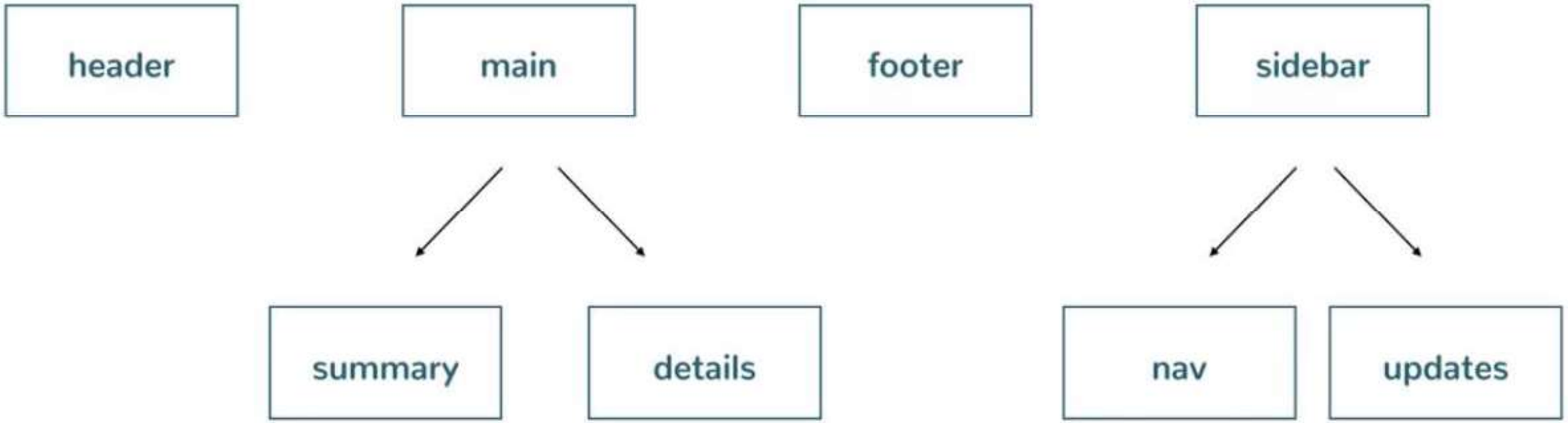
Components on a Page

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

<main-section></main-section>

<side-bar></side-bar>

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

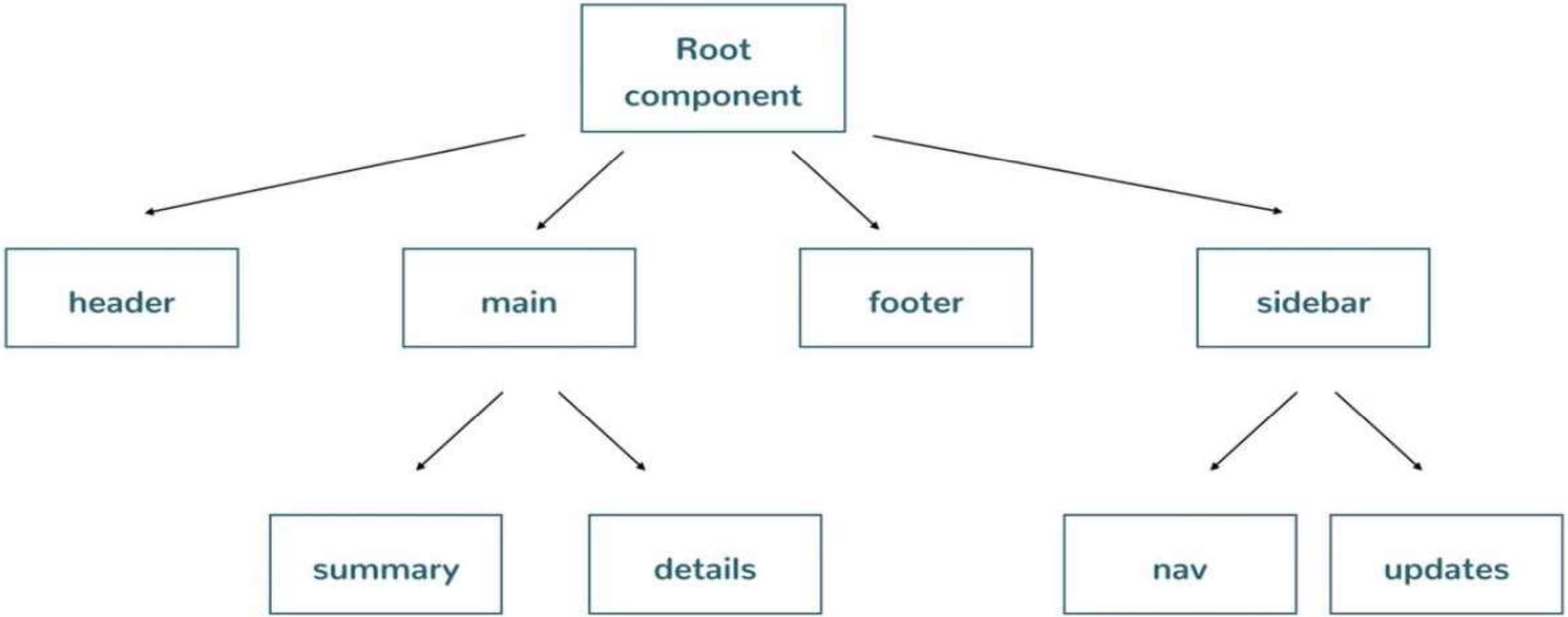



main-section
side-bar
header-section
footer-section
info-section
details-panel

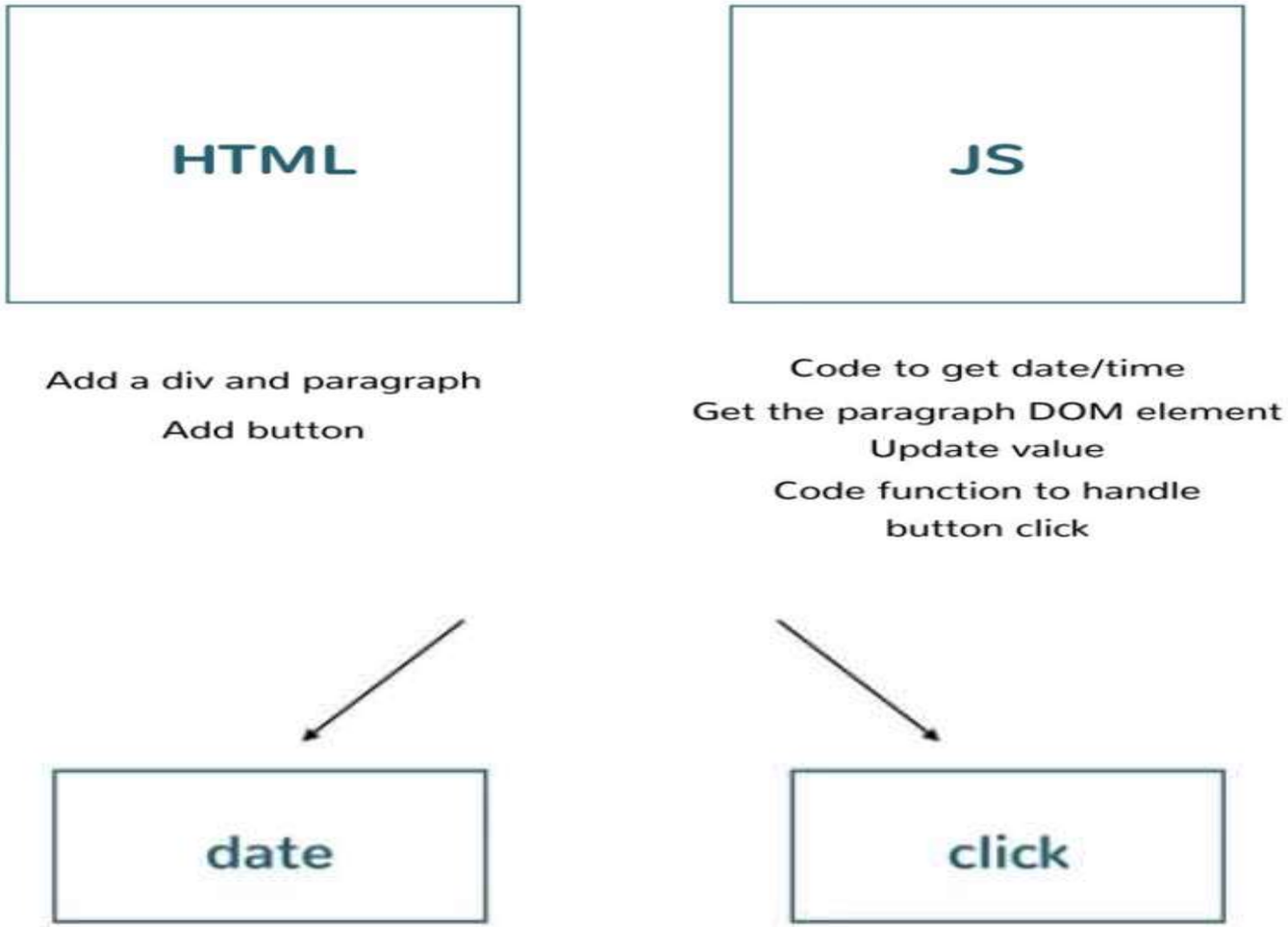
Component Tree Structure



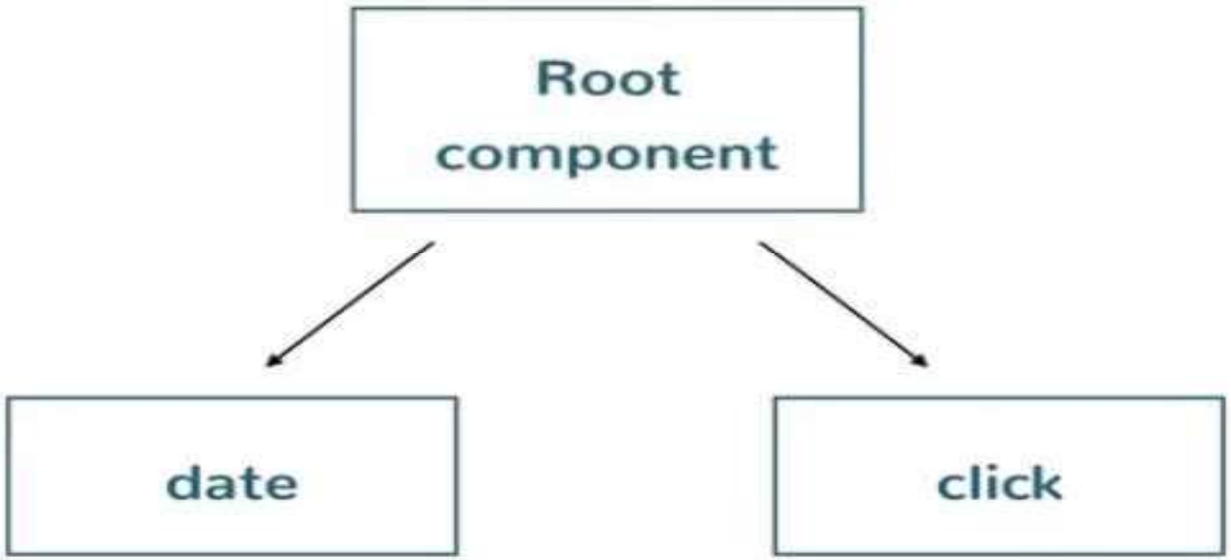
Every Angular Application as a Root Component Which Holds Other Comps



How it Alters Our Application



Finally Our Angular Application Looks Like this



"In React, **everything is a component**"

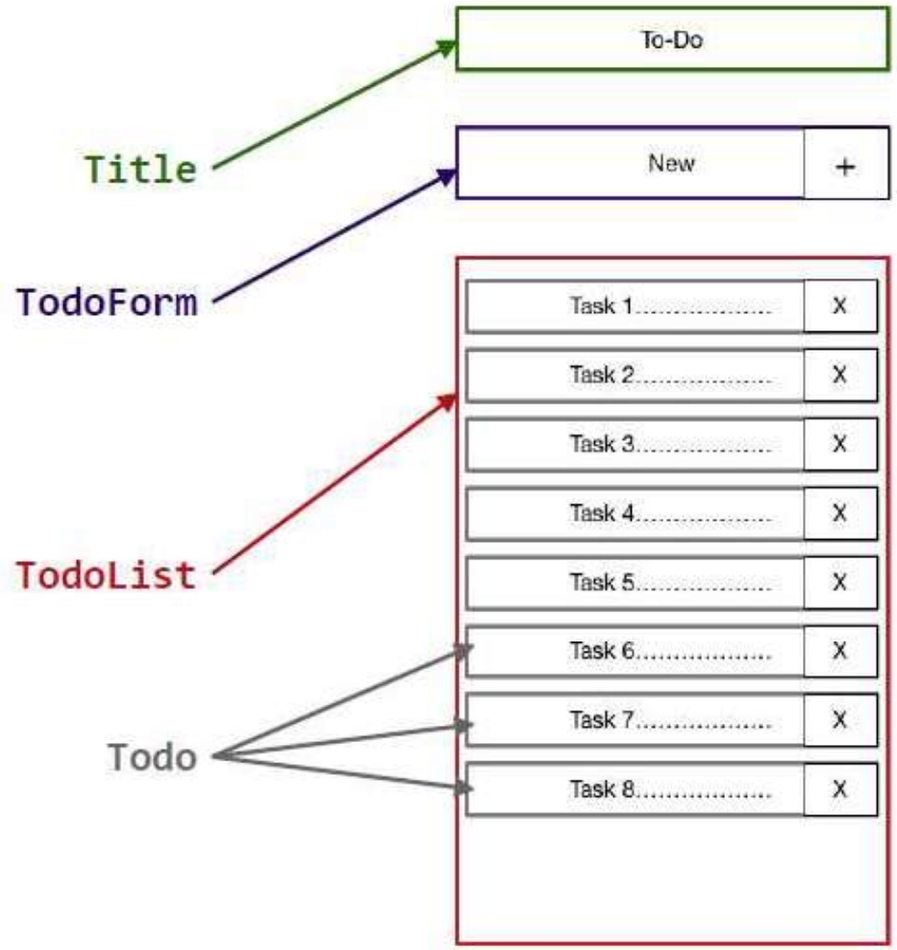
Todo application

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`

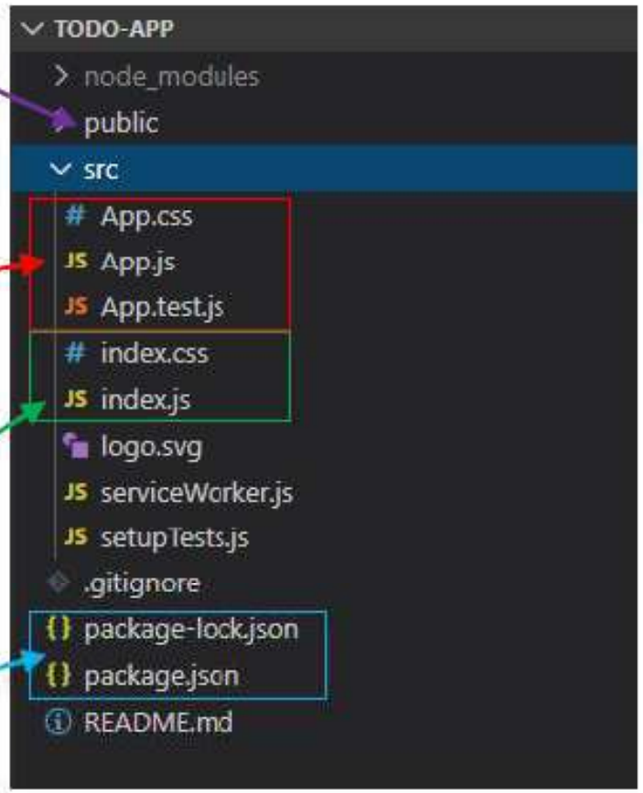
Anatomy of a new React app

public holds the initial html document and other static assets

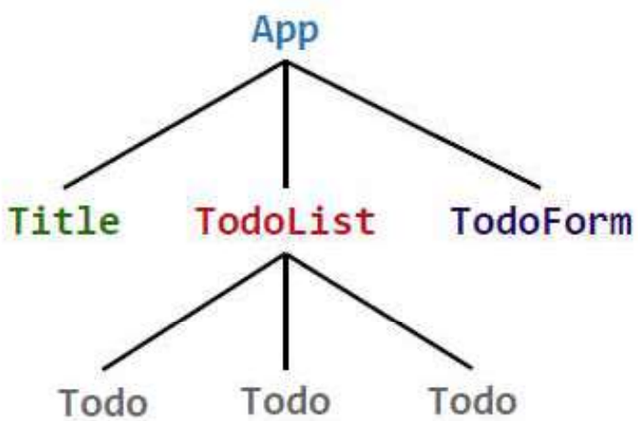
App is a boilerplate starter component

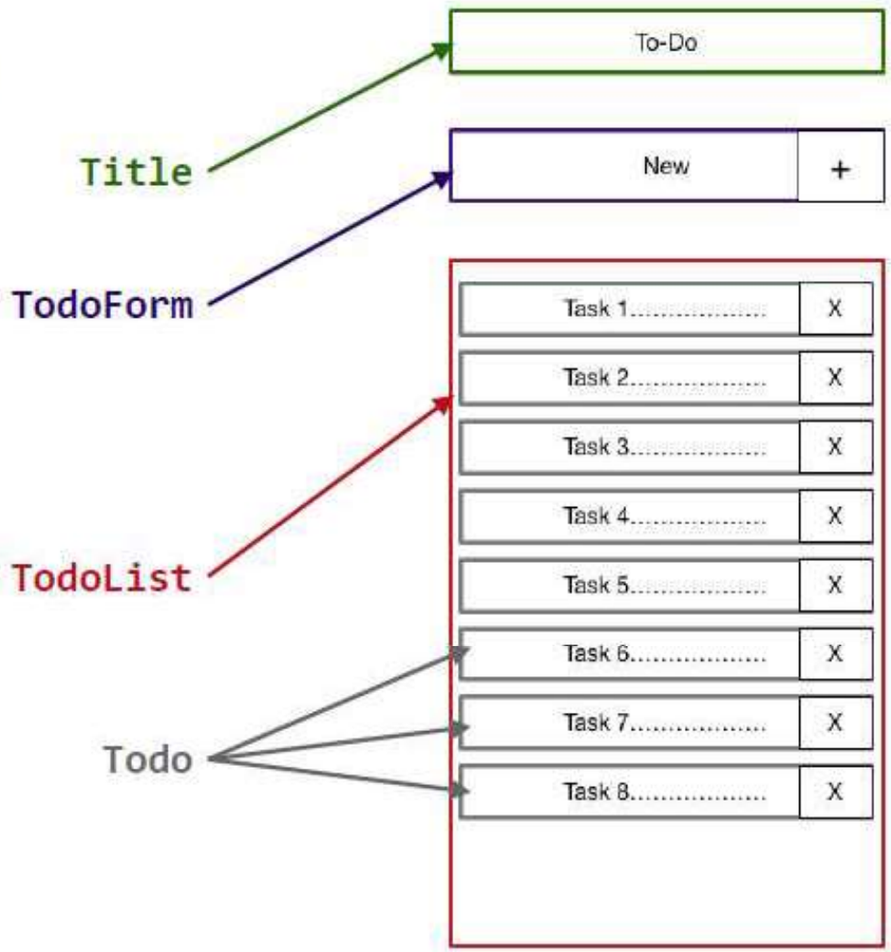
index.js binds React to the DOM

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.

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

Name: Enter a name here

Hello !

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

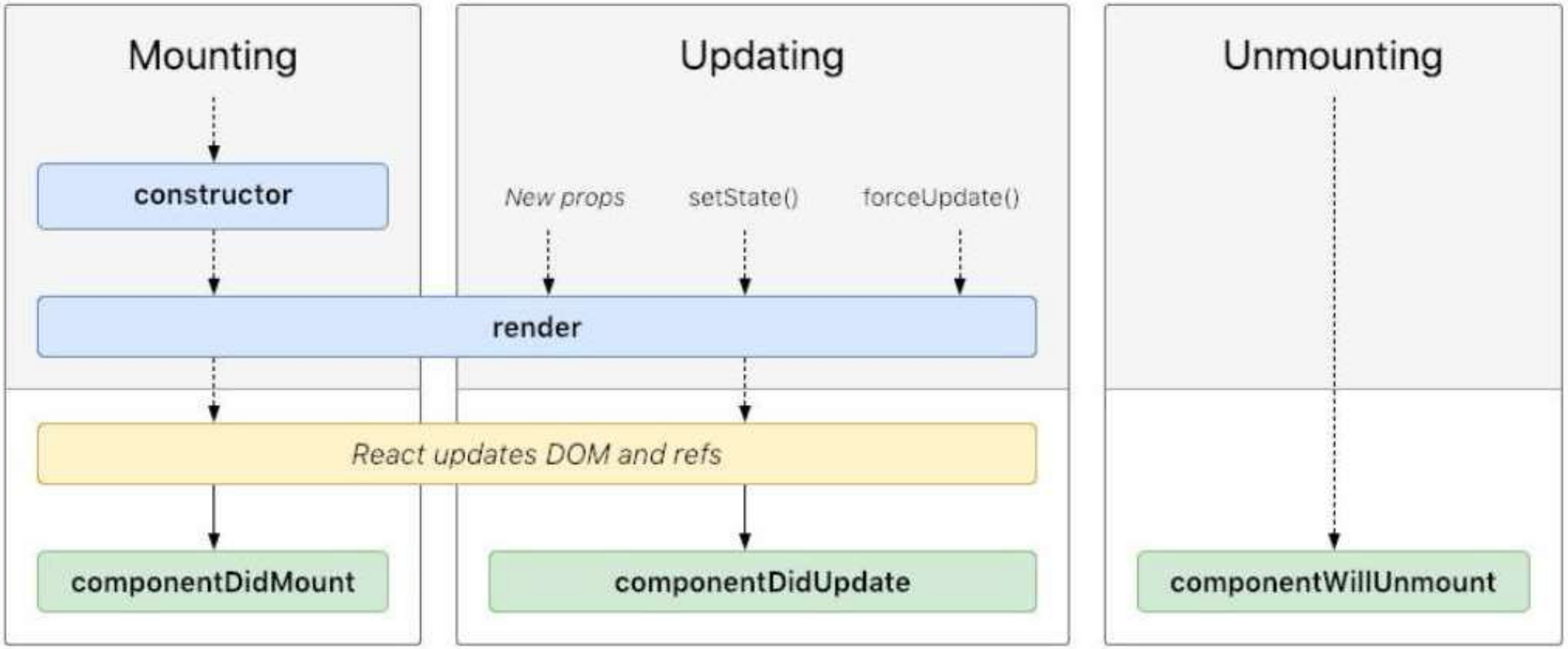
```
import React from 'react';
import ./ReactAppView.css';
class ReactAppView extends React.Component {
  ...
render() {
   return (
      <span className="cs142-code-name">
        ...
      </span>
    );
```

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

Name: D

Hello D!

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);
```

The current value

A setter function to change the value

The initial value to use

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