React

Yaniv Arad

ECMAScript

ES = ECMAScript

A standardized specification of a scripting language

Like Java Script , JScript & others (less common..)

React can be written in ES5 as well as ES6 (last version)

Ecma Script 6

ES6 – What's new ?

- Default Parameters
- Template Literals
- Multi-line Strings
- Destructuring Assignment
- Enhanced Object Literals
- Arrow Functions
- Promises

Constants

```
const PI = 3.141593
PI > 3.0
```

Template Literals

Intuitive expression interpolation for single-line and multi-line strings

```
var customer = { name: "Foo" }
var card = { amount: 7, product: "Bar", unitprice: 42 }
var message = `Hello ${customer.name},
want to buy ${card.amount} ${card.product} for
a total of ${card.amount * card.unitprice} bucks?`
```

Default Parameter

```
function f (x, y = 7, z = 42) {
    return x + y + z
}
f(1) === 50
```

Spread Operator

```
var params = [ "hello", true, 7 ]
var other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]
f(1, 2, ...params) === 9

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

Export/Import

```
//  lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

//  someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))

//  otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

Object Oriented - Classes

```
class Shape {
    constructor (id, x, y) {
        this.id = id
        this.move(x, y)
    }
    move (x, y) {
        this.x = x
        this.y = y
    }
}
```

Object Oriented - 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
    }
}
```

Object Oriented - Static Members

```
class Rectangle extends Shape {
    ...
    static defaultRectangle () {
        return new Rectangle("default", 0, 0, 100, 100)
    }
}
class Circle extends Shape {
    ...
    static defaultCircle () {
        return new Circle("default", 0, 0, 100)
    }
}
var defRectangle = Rectangle.defaultRectangle()
var defCircle    = Circle.defaultCircle()
```

New data structures -Set

```
let s = new Set()
s.add("hello").add("goodbye").add("hello")
s.size === 2
s.has("hello") === true
for (let key of s.values()) // insertion order
    console.log(key)
```

New data structures - Map

```
let m = new Map()
m.set("hello", 42)
m.set(s, 34)
m.get(s) === 34
m.size === 2
for (let [ key, val ] of m.entries())
    console.log(key + " = " + val)
```

Arrow Functions - Map Function

```
odds  = evens.map(v => v + 1)
pairs = evens.map(v => ({ even: v, odd: v + 1 }))
nums  = evens.map((v, i) => v + i)
```

React – What is It ?

- A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES

CREATED BY FACEBOOK (AND USED IN INSTAGRAM ALSO)

-The “View” in the Application…A FAST one !

React Basics

JS Programming “Patterns”

Imperative vs Declarative

Declarative style - allows you to control flow and state in your application by saying "It should look like this“ (The “What?”)

Imperative style - allows you to control your application by saying "This is what you should do". (The “How?”)

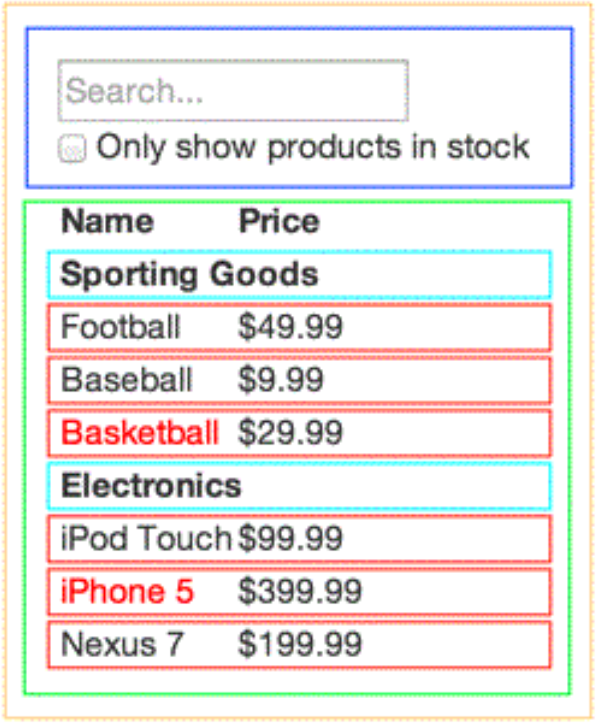
Components – React's Bread & Butter.

When you're in React's world you are just building **components** that fit into other components.

Everything is a component.

Components Driven Application

- FilterableProductTable
  - SearchBar
  - ProductTable
    - ProductCategoryRow
    - ProductRow



DOM – A Reminder

DOM stands for Document Object Model and is an abstraction of a structured text.

For web developers, this text is an HTML code, and the DOM is simply called HTML DOM. Elements of HTML become nodes in the DOM.

So, while HTML is a text, the DOM is an in-memory representation of this text.

DOM – Cont..

whenever we want to dynamically change the content of the web page,
we modify the DOM…

```
var item = document.getElementById("myLI");
item.parentNode.removeChild(item);
```

DOM - Issues

The DOM trees are huge nowadays ..

SPA Oriented - need to modify the DOM tree **incessantly** and **a lot**.

Problems :

- Hard to manange : Lost the context ? dive really deep into the code to even know what's going on .. **time-consuming and bug-risky.**

- Inefficient - Do we really need to do all this findings **manually**?

Solutions ?

Declarativeness - Instead of low-level techniques like traversing the DOM tree manually, you simple *declare* how a component should look like.

React does the low-level job for **you**

But..What about the performance issue ???

Virtual DOM

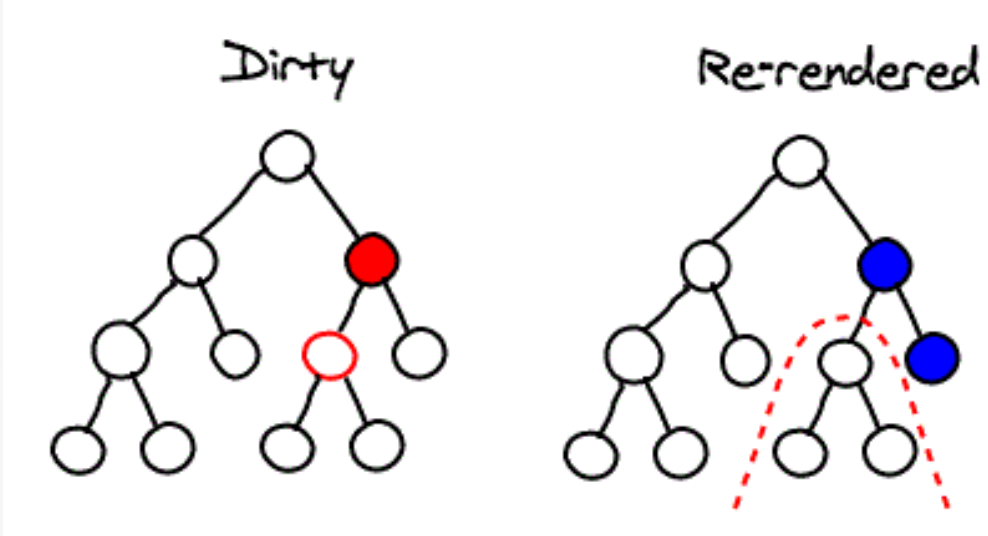
The *Virtual DOM* is an abstraction of the HTML DOM.

lightweight and detached from the browser-specific implementation details

Virtual DOM – How it works ?

Builds the tree representation of the DOM in the memory and calculates which DOM element should change

React's diffing algorithm uses the tree representation of the DOM and re-calculates all subtrees when its' parent got modified



React JS – Lets Begin…

Use Node & NPM for Dev life cycle :

>npm install –g create-react-app

>create-react-app myapp
For Type script :
>create-react-app –scripts-version=react-scripts-ts

>cd myapp

>npm start

Create Elements

ReactDOM - The primary API for rendering into the DOM :

```
ReactDOM.render(reactElement, domContainerNode)
```

Create Elements

```
<body>
  <div id="example"></div>
  <script type="text/babel">
    ReactDOM.render(
      <h1>Hello, world!</h1>,
      document.getElementById('example')
    );
  </script>
</body>
```

JSX

JSX = Java Script Extension

Combining JS and HTML (inline)

```
render : function()
{
    return <h1>Hello From Component !</h1>;
}
```

The idea ?

Have everything is one place

-The “Component style” – single responsibility !

Components

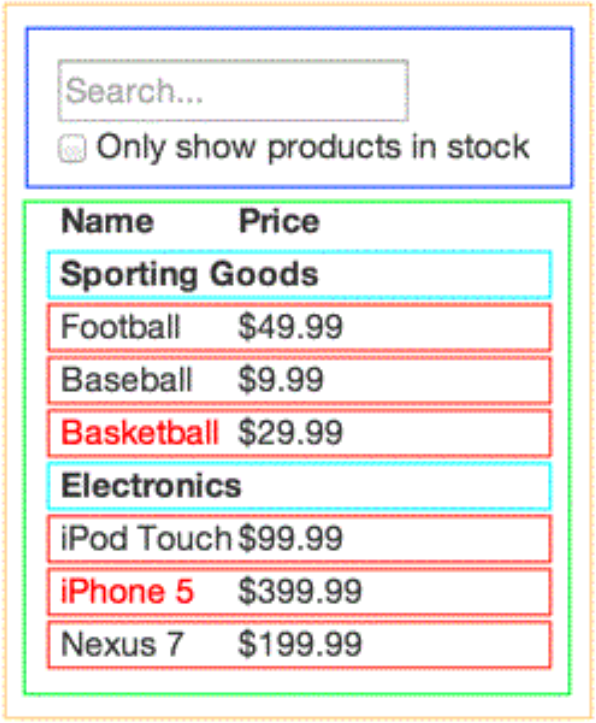
Components

- The “Bread & Butter” of React fundamentals…

- Let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Everything is a COMPONENT

- FilterableProductTable
  - SearchBar
  - ProductTable
    - ProductCategoryRow
    - ProductRow



Create Component

```
 <div id="myDiv"></div>
<script type="text/babel">

    class MyComponent extends React.Component
    {
        render()
        {
            return (<div>Hello from Component</div>);
        }
    };
    ReactDOM.render(<MyComponent />, document.getElementById('myDiv'));
 </script>
```

Set Style

Style - must be an object !

```
<script type="text/babel">

    class MyComponent extends React.Component
    {
        render()
        {
        return (<div style={{backgroundColor:"#FF0000", width:"400px"}}>
            Hello from Component</div>);
        }
    };
    ReactDOM.render(<MyComponent />, document.getElementById('myDiv'));
 </script>
```

Props

A Way to pass IMMUTABBLE data into a component

```
<script type="text/babel">
      class Book extends React.Component
      {
          render()
          {
              return (<div>
                          The Title is {this.props.Title}
                          The Proce is {this.props.Price}
                      </div>);
          }
      };
      ReactDOM.render(<Book Title="Harry Potter" Price="50" />,
       document.getElementById('myDiv'));
```

Props

Can also be used in the constructor

```
<script type="text/babel">
      class Book extends React.Component
      {
          constructor(props)
          {
              super(props);
              // Do Something with this.props.Title
          }
          render()
          {
              return (<div>
                          The Title is {this.props.Title}
                          The Proce is {this.props.Price}
                      </div>);
          }
      };
      ReactDOM.render(<Book Title="Harry Potter" Price="50" />,
```

Event Handling

```
class MyComp extends React.Component
{
    constructor()
    {
        super();
        this.setText = this.setText.bind(this);
    }
    setText(e)
    {
        alert(e.target.value);
    }
    render()
    {
        return (<div><input type="text" onChange={this.setText} /> </div>);
    }
};
```

State

props are set by the parent and they are fixed throughout the lifetime of a component.

For data that is going to change, we have to use state.

State is :

- Initialized in the constructor
- Changed over lifecycle by setState method

State

```
class Book extends React.Component
{
    constructor()
    {
        super();
        this.state = { Data:''};

        this.setText = this.setText.bind(this);
    }
    setText(e)
    {
        this.setState({Data : e.target.value});
    }
    render()
    {
        var data = this.state.Data
        return (<div>
                    <input type="text" onChange={this.setText}  />
                    Data is : {data}
                </div>);
    }
};
```

State - The problem

State Updates May Be **Asynchronous** !!!

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment
});
```

```
// Correct
this.setState(function(prevState, props) {
  return {
    counter: prevState.counter + props.increment
  };
});
```

Repeater

Rendering dynamic collection

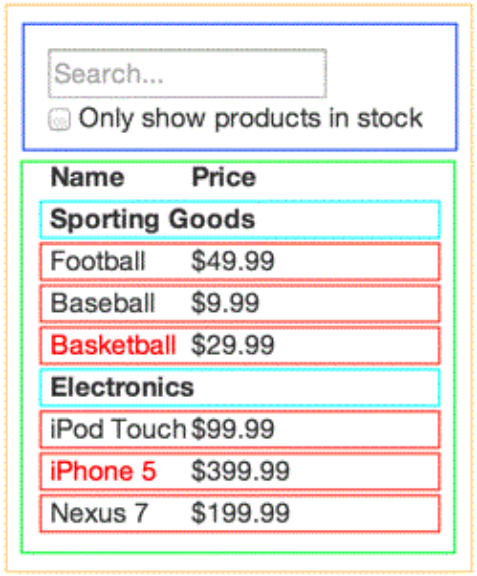
```
class MYComp extends React.Component{
   constructor()
   {
      super();
      this.state =  { items: ['One','Two','Three']}
    }
  render()
  {
    var listItems = this.state.items.map(function(item) {
      return (<h1>item</h1>);
    });

    return (<div>{listItems></div> );
  }
```

Nested Components

Child Components

Components WITHIN OTHER Components

- FilterableProductTable
  - SearchBar
  - ProductTable
    - ProductCategoryRow
    - ProductRow

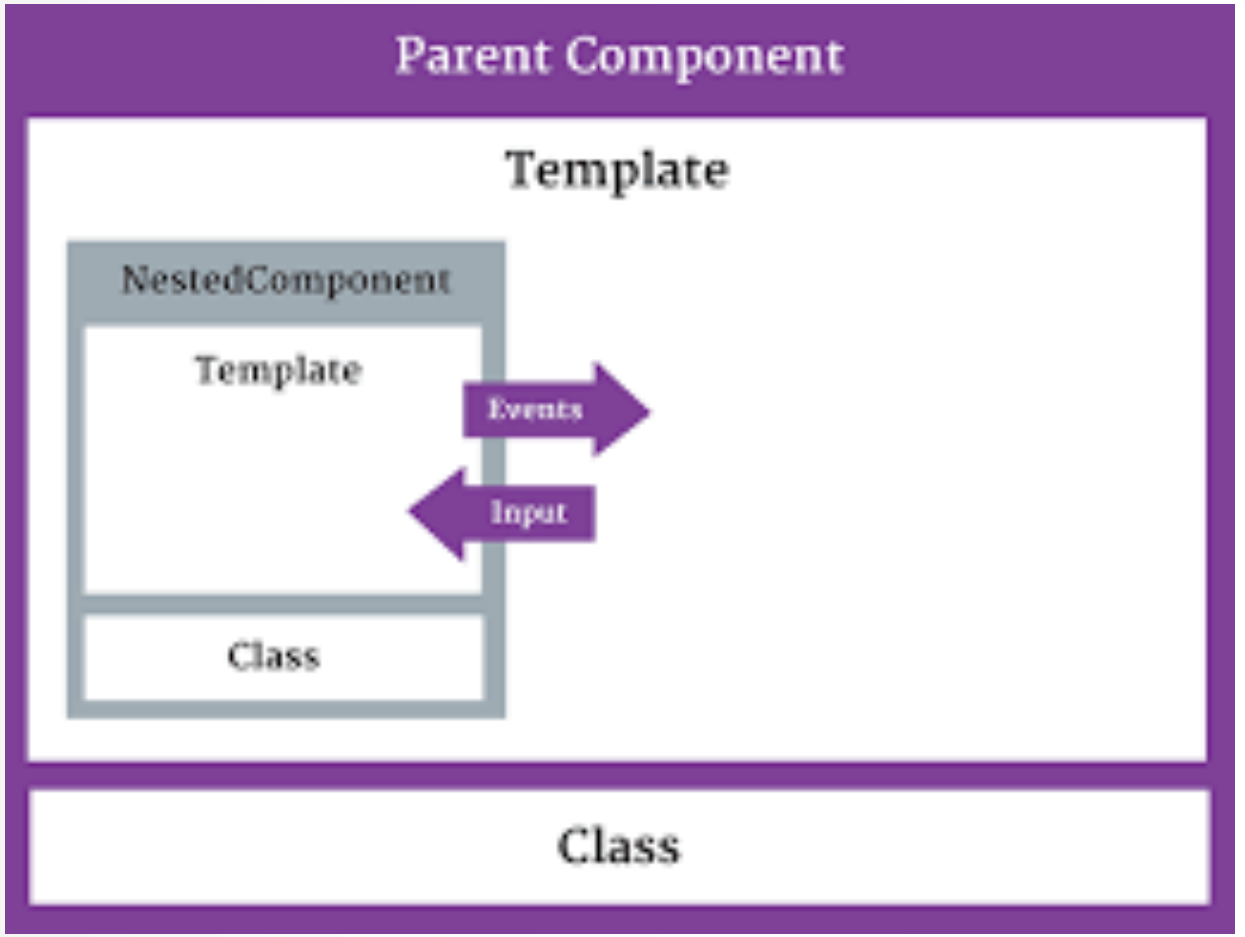


Nested Components

```
class ChidComp extends React.Component
{
    render()
    {
        return (<h1>Hello from Child Comp</h1>);
    }
};

class ParentComp extends React.Component
{
    render()
    {
        return (<div><h1>Hello from Parent Comp</h1>
        <ChidComp /></div>);
    }
};
ReactDOM.render(<div><ParentComp /></div>, document.getElementById('myDiv'));
```

Passing Data

Also called Bi-Dirctional Data Flow...



Passing Data from Parent to Child

Use Props !

```
class ChidComp extends React.Component
    {
        render()
        {
            return (<div><h2>{this.props.Data}</h2></div>);
        }
    };

     class ParentComp extends React.Component
    {
            return (<div><ChidComp Data={"Hello From Parent"} /></div>);
    };
```

Passing Data from Child to Parent

A Parent always know his childs components but...

Child CAN NOT know it's parent!

We need a callback function !

Passing Data from Child to Parent

The Parent Component...

```
class ParentComp extends React.Component
{
    constructor()
    {
        super();
        this.state = { passedData: '' }
    }
    onChildChanged(newData)
    {
        this.setState({ passedData: newData })
    }
    render()
    {
        return <div><input type="text" value={this.state.passedData} /><ChildComp
        callbackParent={(data) => this.onChildChanged(data) } /></div>
    }
```

Passing Data from Child to Parent

The Child Component...

```
class ChildComp extends React.Component
{
  onTextChanged(e)
  {
    this.props.callbackParent(e.target.value); // we notify our parent
  }
  render()
  {
    return <div><input type="text" onChange={(d) => this.onTextChanged(d)} /></div>
  }
}
```

Access a RESTfull service server

REST API

HTTP Method	Action	Examples
GET	Obtain information about a resource	http://example.com/api/orders (retrieve order list)
GET	Obtain information about a resource	http://example.com/api/orders/123 (retrieve order #123)
POST	Create a new resource	http://example.com/api/orders (create a new order, from data provided with the request)
PUT	Update a resource	http://example.com/api/orders/123 (update order #123, from data provided with the request)
DELETE	Delete a resource	http://example.com/api/orders/123 (delete order #123)

Access REST-full Service

Use **Axios** package for async requests to a REST-full service

Axios is a promise based !

>npm install axios –save

For using : import axios from ‘axios’

Access REST-full Service - GET

```
public getUsers()
{
    axios.get(`https://jsonplaceholder.typicode.com/users`)
     .then((res : any) => {
       const persons = res.data;
       this.setState({ users : persons });
     })
}
```

Access REST-full Service - POST

```
public addUser()
{
  const user =
  {
    name : 'Avi'
  };
    axios.post(`https://jsonplaceholder.typicode.com/users`,user)
     .then((res : any) => {
       console.log(res.data);
     })
}
```

Access REST-full Service - PUT

```
 public updateUser()
{
  const user =
  {
    name : 'Avi'
  };
    axios.put(`https://jsonplaceholder.typicode.com/users/${this.state.id}`,user)
     .then((res : any) => {
       console.log(res.data);
     })
}
```

Access REST-full Service - DELETE

```
public deleteUser()
{
    axios.delete(`https://jsonplaceholder.typicode.com/users/${this.state.id}`)
     .then((res : any) => {
       console.log(res.data);
     })
}
```

Forms

Forms

React has NOT built-in forms mechanism (like Angular..)

Some external known libraries : Formik, Redux Forms..

Forms

React has NOT built-in forms mechanism (like Angular..)

Some external known libraries : Formik, Redux Forms..

Forms

The big difference from “regular” forms : Instead of sending the data from the DOM to the server directly, send it to the component for validations, data shaping and sending to the server

Forms

```
handleSubmit(e)
{
  e.preventDefault(); // prevent submitting the page
  //Do some logic/validation before send it to the server
}
 render() {
   return (
     <div className="App">
       <form onSubmit={ e => this.handleSubmit(e) }>
         User : <input type="text" onChange={ e => this.setName(e)} /><br/>
         Password : <input type="password" onChange={ e => this.setPwd(e)}
         <input type="submit" />
       </form>
     </div>
   );
 }
}
```

Forms

The errors messages using..binding !

```
constructor()
{
  super();
  this.state = { 'user' : '', 'pwd' :'', 'nameHasValue' : false}
}
```

```
setName(e)
{
  var name = e.target.value;
  this.setState( prevState => {
    return {'name' : name};
  }, () => {
   if (this.state.name == null || this.state.name.length == 0)
   {
     this.setState({'nameHasValue' : false})
   }
  else
   {
      this.setState({'nameHasValue' : true})
   }});
}
```

Forms

The errors messages using..binding !

```
render() {
  var mandatoryError;
  if (this.state.nameHasValue == false)
  {
    mandatoryError = <div>User name ia mandatory !!</div>;
  }
  return (
    <div className="App">aa
      <form noValidate onSubmit={ e => this.handleSubmit(e) }>
        User : <input required type="text" onChange={ e => this.setName(e)}
        Password : <input type="password" onChange={ e => this.setPwd(e)} />
        <input type="submit" />
        {mandatoryError}
      </form>
    </div>
  );
```

Component Life Cycle

Component Life Cycle

Component Initialization – Events Lifecycle

1 – Constructor
2 – getDerivedStateFromProps
3 – render
4 - componentDidMount

Constructor()

- Called **BEFORE** component is mounted

- The right place to inisialize state

- NO place for business logic

static getDerivedStateFromProps(props,state)

- invoked immediately before mounting occurs

- called before render(), therefore setting state synchronously in this method will not trigger a re-rendering.

-the only lifecycle hook called on server rendering

-The best place to initialize state based on props !

componentDidMount()

- invoked immediately after a component is mounted

- Do DOM intercations here

-Do AJAX calles here

- Setting state in this method will trigger a re-rendering.

Component Life Cycle

Component Update – Events Life cycle

1 – getDerivedStateFromProps
2 - shouldComponetUpdate
3 – render
4 – getSnapshotBeforeRender
5 - componentDidUpdate

static getDerivedStateFromProps(props,state)

- In every state/props update

- Returns a new state (as a result of a possible props change) or returns null

shouldComponentUpdate(nextProps, nextState)

- Let React know if a component's output is not affected by the current change in state or props

- The default behavior is to re-render on every state change

- Invoked before rendering when new props or state are being received

-If returns false, then getSnapshotBeforeUpdate(), render(), and componentDidUpdate() will not be invoked

getSnapshotBeforeUpdate(props,state)

Invoked right before the most recently rendered output is committed from the VDOM to the DOM. It enables your component to capture some information

from the DOM (e.g. scroll position) before it is potentially changed.

-The function should return a value (based on current ui elements). This value is will be passed to componentDidUpdate as the third parameter

componentDidUpdate(prevProps, prevState)

- Invoked immediately after updating occurs.

- Use this as an opportunity to operate on the DOM when the component has been updated

- A good place to do AJAX requests as long as you compare the current props to previous props

componentWillUnmount()

- Is invoked immediately before a component is unmounted and destroyed

- Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any DOM elements that were created in componentDidMount

"Render Phase"

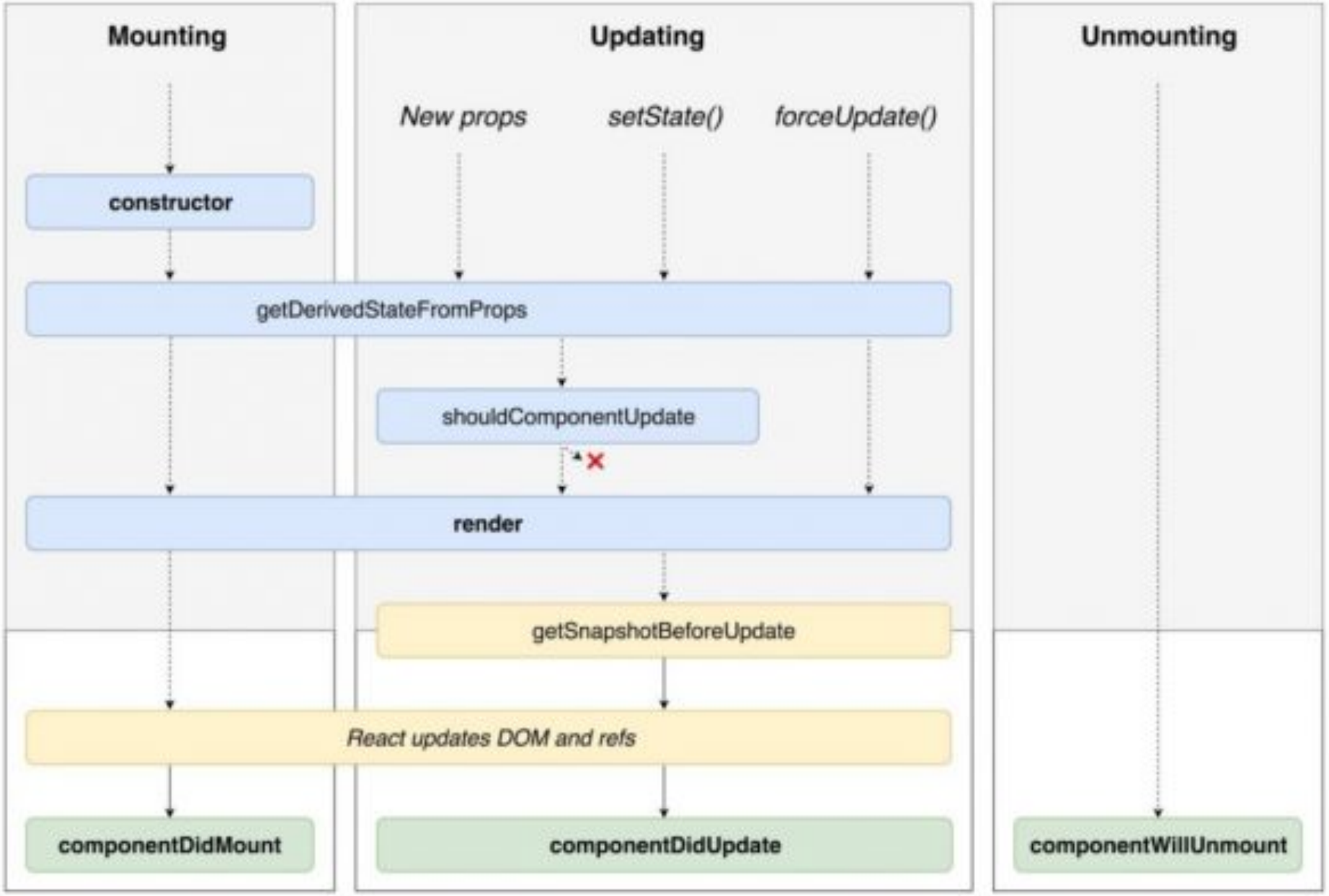
Pure and has no side effects. May be paused, aborted or restarted by React.

"Pre-Commit Phase"

Can read the DOM.

"Commit Phase"

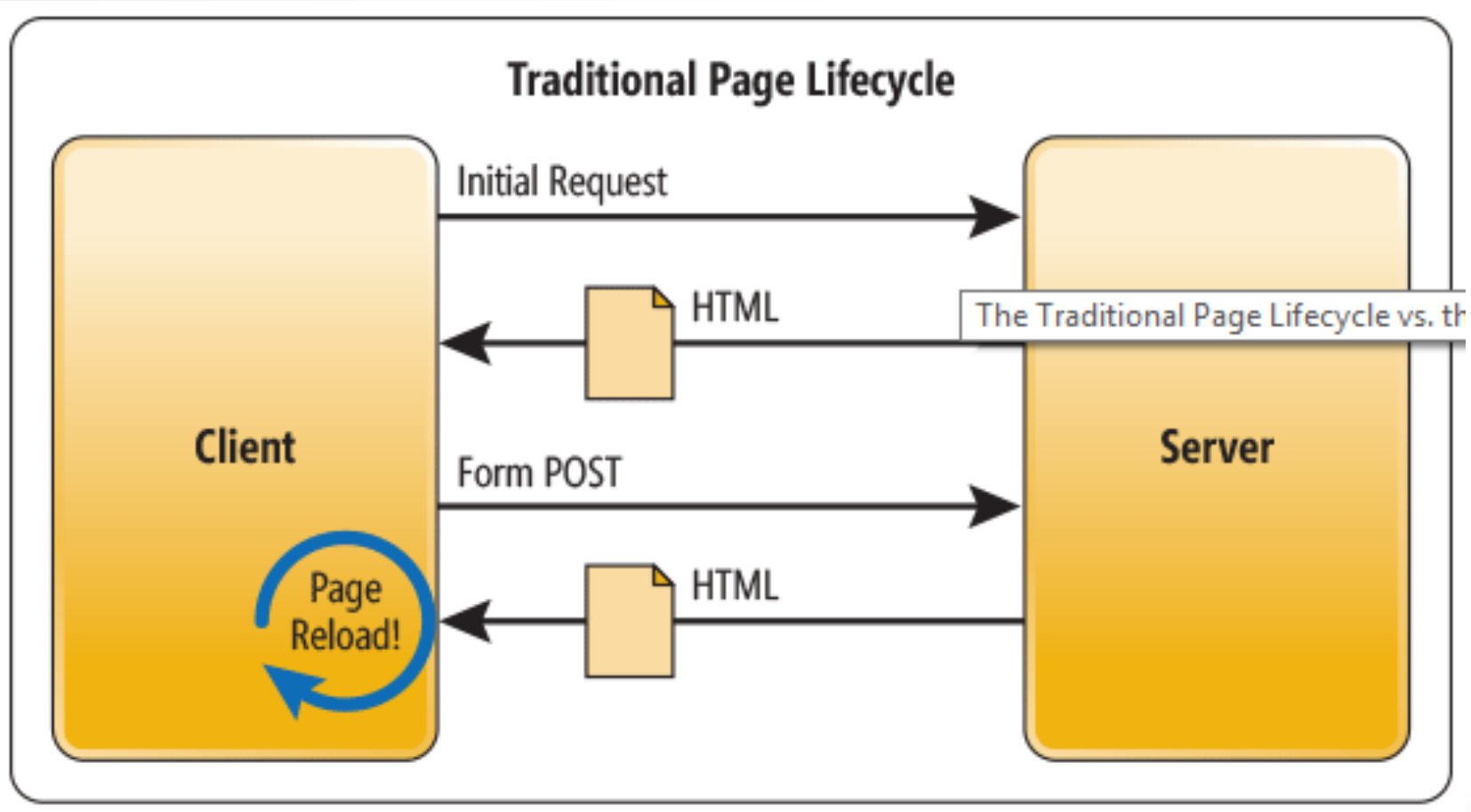
Can work with DOM, run side effects, schedule updates.



Single Page Application

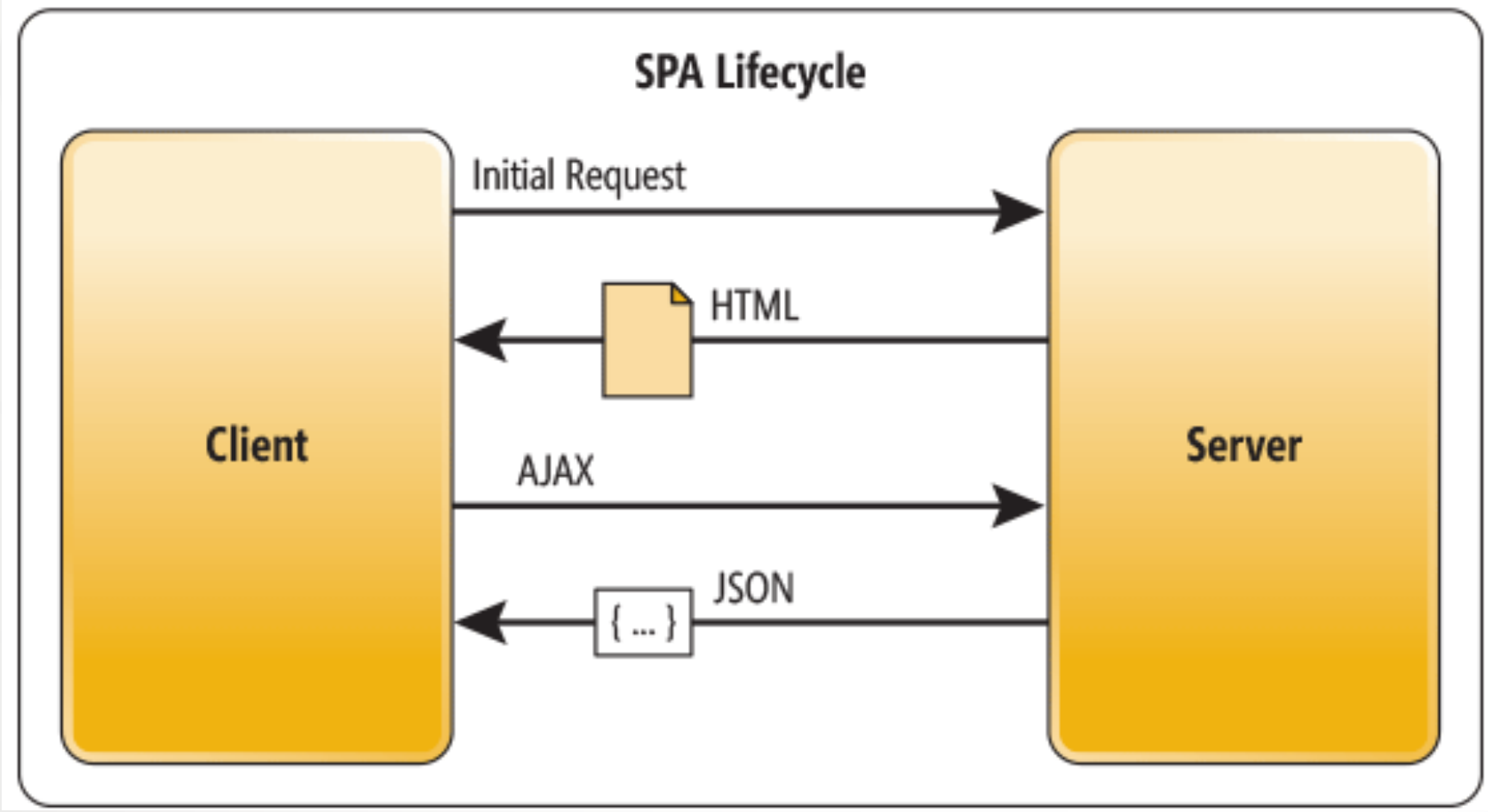
SPA - Single Page Application

Traditional Web Applications



SPA - Single Page Application

Modern Web Application



SPA Benefits

- Applications are more fluid and responsive

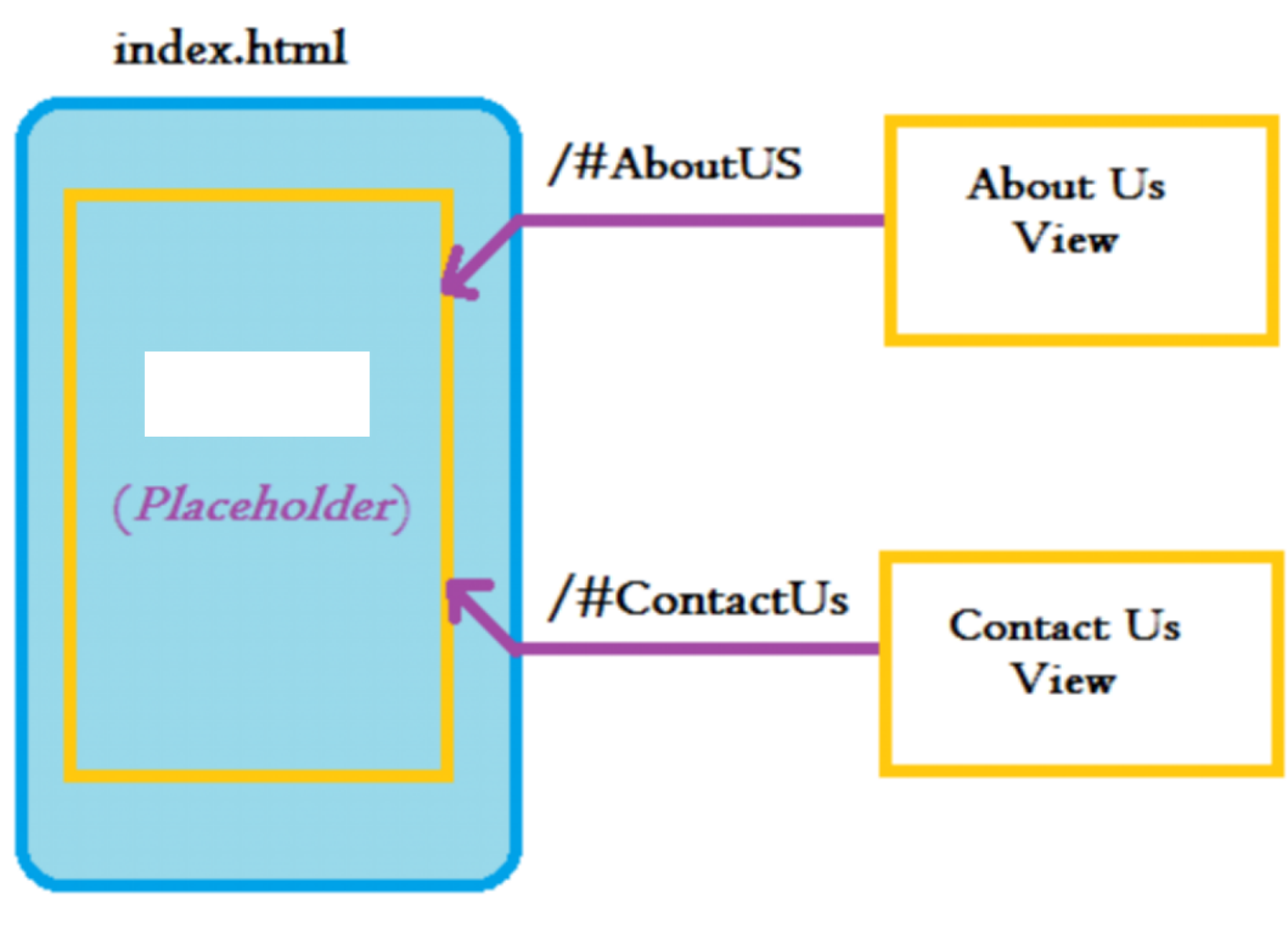
- (Sending the app data as JSON) Creates a separation between the presentation (HTML markup) and application logic

- “Mobile alike” User Experience

Routing = Navigation

- Routing is a core concept in single page applications(SPA).
- React introduced a new router that was built from scratch to integrate with the the concepts of component compositions.

Routing = Navigation



Routing – Let's begin

Need to install the React Router Dom package

> npm install react-router-dom --save

Routing

"Wrapping" the application with the proper Router. The default is BrowserRouter

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

/* App is the entry point to the React code.*/
import App from './SPA/MainPage';

/* import BrowserRouter from 'react-router-dom' */
import { BrowserRouter } from 'react-router-dom';

ReactDOM.render(
    <BrowserRouter>
        <App />
    </BrowserRouter>
    , document.getElementById('root'));

export default App;
```

Routing

Main (and only page) with Components Container

```
import React, { Component } from 'react';
import { Link, Route, Switch} from 'react-router-dom';
import Contact from './Contact';
import About from './About';
```

```
  render() {
    return (
      <div>

        <nav>
          <ul>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/Contact">Contact</Link></li>
          </ul>
        </nav>

        <Switch>
          <Route  path="/contact" component={Contact}/>
          <Route path="/about" component={About}/>
        </Switch>
      </div>
```

Routing

Main (and only page) with Components Container

```
<li><Link to="/">Home</Link></li>
<li><Link to="/About">About</Link></li>
<li><Link to="/contact">Contact</Link></li>
```

Link - The Router API for (instead of traditional “a” tag) specifies the location we will be telling React Router we are virtually navigating to

Routing with Params - Pass Data

Need to be set in the routing configuration

```
<Route  path ="Home" component={Home}/>
<Route path="about/:ID" component={About} />
<Route path="contact" component={Contact} />
```

Routing with Params - Get Data

Get the data in the target component via props.match

```
 static getDerivedStateFromProps(props,state)
 {
  return {'id' : props.match.params.ID};
 }

render() {
    console.log('Rendered');
    return (
      <div className="App">
        The id is : {this.state.id}
      </div>
    );
  }
```

Routing from code

Using a special object, the “history” for queueing navigations

```
navigate()
{
    this.props.history.push(`/Contact/${this.state.id}`)
}
render() {

  return (
    <div className="App">
     <input type="button" onClick={this.navigate}/>
    </div>
  );
}
}
```

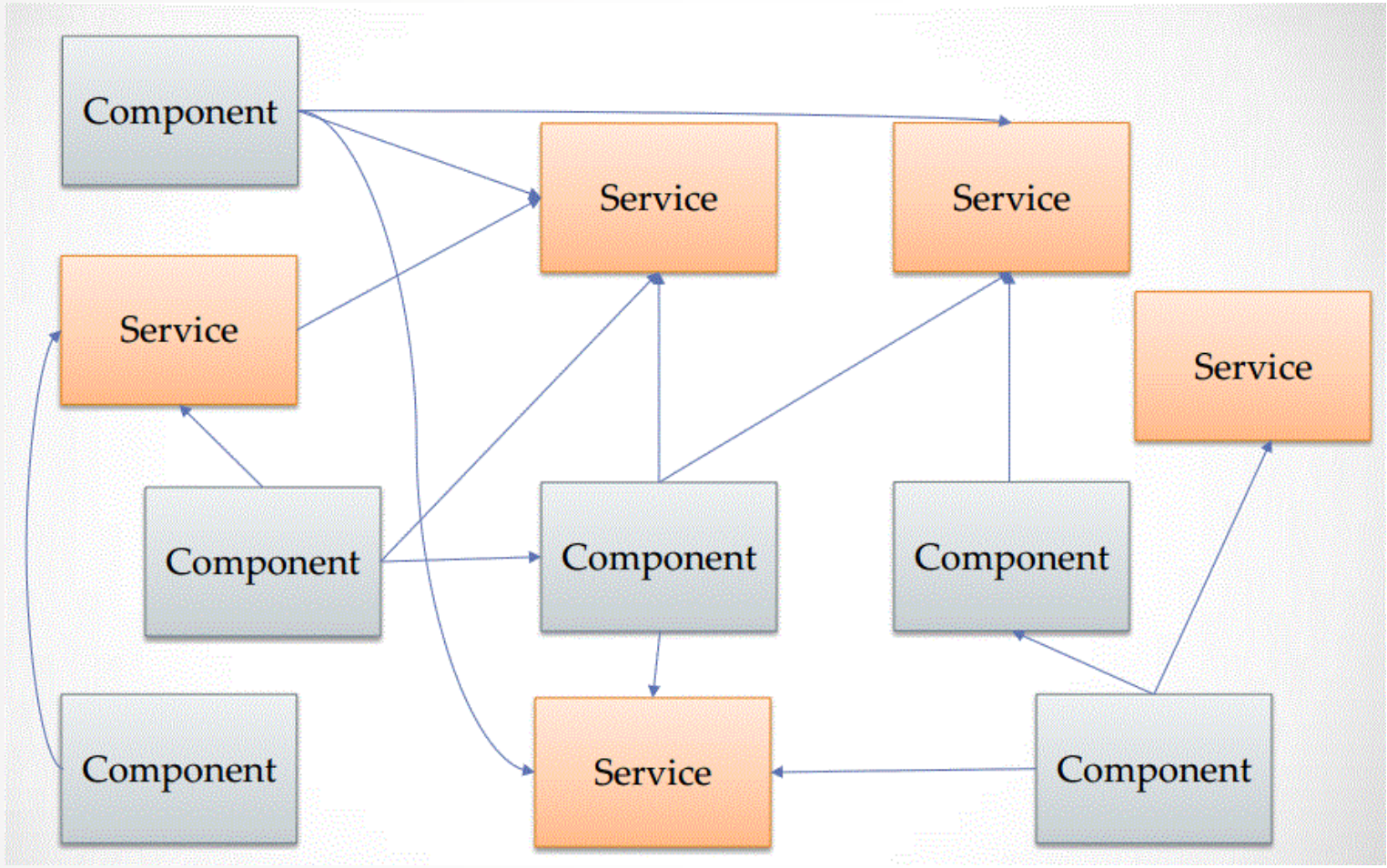
Nested Routing

Created Routes in the sub-component itself

```
return (<div>

  <h3>Products Page</h3>
   <ul>
    <li><Link to={`${this.props.match.url}/details/{this.state.id}`}>Get Details</Linl
    <li><Link to={`${this.props.match.url}/add`}>Add New</Link></li>
  </ul>
  <Switch>
    <Route path={`${this.props.match.url}/details/:id`} component={Details}/>
    <Route path={`${this.props.match.url}/add`} component={AddNew}/>
  </Switch>
</div>)
```

Redux

SPA's can be complex



Redux

- Design Pattern based on FLUX
- Predictable state container for javaScript apps

Redux key principles

Single source of truth :

The application state is stored in an object tree

Within a single store

Redux key principles

State is read only

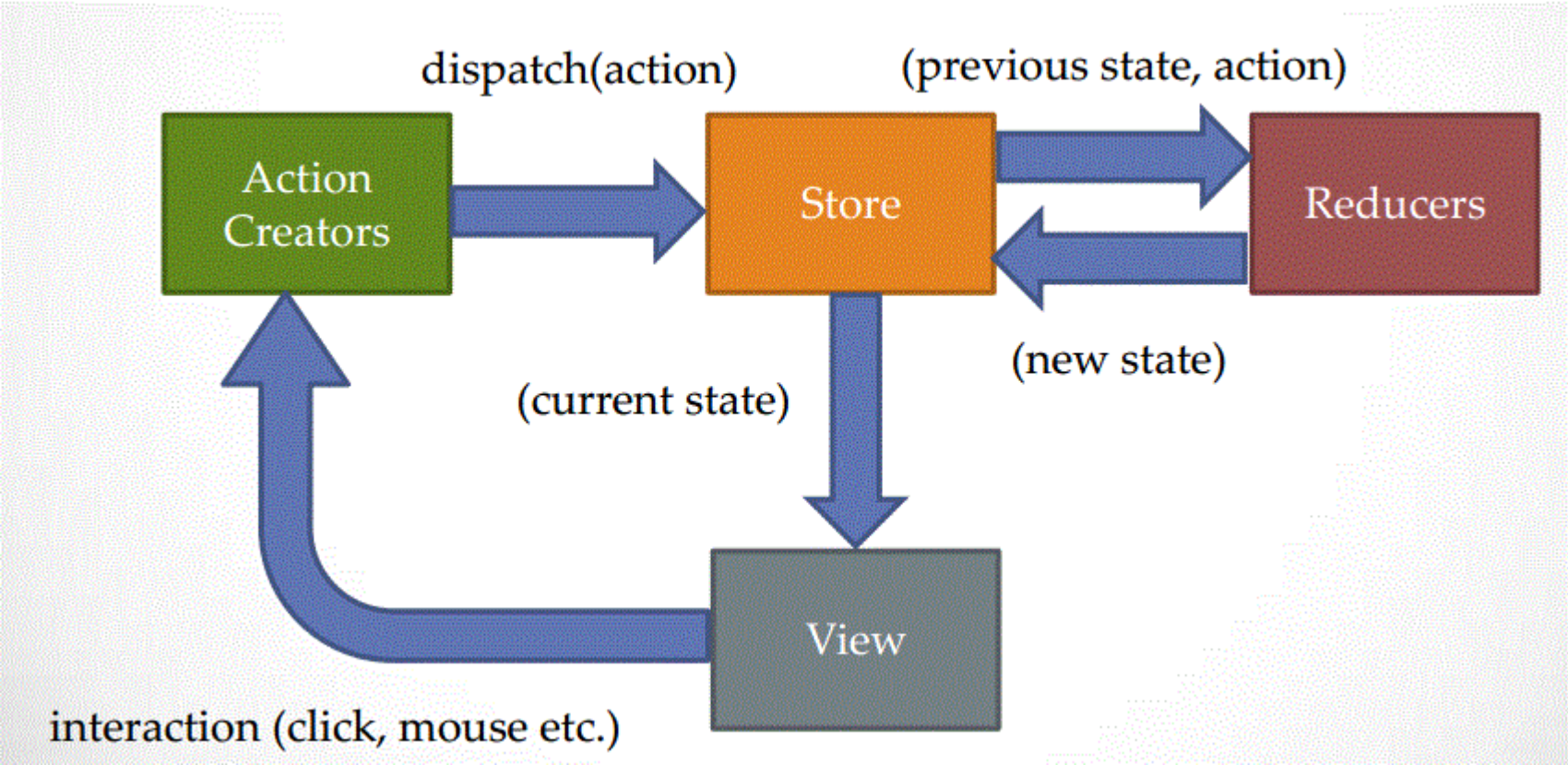
- The application state is stored in an object tree
  - Within a single store
- The only way to mutate the state is to emit an action describing what happened

Redux key principles

Changes are made with pure functions

To specify how the state is transformed by actions, you write a pure function (called Reducer)

Redux Data Flow



Implement Redux in React

Implement Redux in React

```
> npm install redux –save
> npm install react-redux --save
```

Implement Redux in React

We'll create a store with a counter data..

Implement Redux in React

The Reducer – Converts a current state to a new state according to the Type given

```
const mainReducer = (state = { counter : 0}, action) => {
  switch(action.type) {
    case 'INCREMENT':
      return { counter :  state.counter + 1 };

    case 'DECREMENT':
      return { counter :  state.counter -  1 };

    default:
      return state;
  }
}
export default mainReducer;
```

Implement Redux in React

```
import App from './App';
import registerServiceWorker from './register

import { createStore } from 'redux';
import { Provider } from 'react-redux';
import mainReducer from './mainReducer';
const store = createStore(mainReducer);


ReactDOM.render(
    <Provider store={store} >
    <App />
    </Provider>
    , document.getElementById('root'));
registerServiceWorker();
```

Implement Redux in React

```
import App from './App';
import registerServiceWorker from './register

import { createStore } from 'redux';
import { Provider } from 'react-redux';
import mainReducer from './mainReducer';
const store = createStore(mainReducer);


ReactDOM.render(
    <Provider store={store} >
    <App />
    </Provider>
    , document.getElementById('root'));
registerServiceWorker();
```

We Create 1 store for our app, and use it by a Provider

Implement Redux in React

```
import React, { Component } from 'react';
import {connect} from 'react-redux';

class Comp1 extends Component {
increment = () =>
{
  const data = { counter : 3};
  this.props.dispatch({
    type : 'INCREMENT',data});
}
  render() {
    return (
      <div className="App">
      Comp1
      <input type="button" value="+" onClick={this.increment}/>
      </div>);}
}
export default connect()(Comp1);
```

A Component dispatch an action (in the reducer) that changes the state in the store.

The “connect” connects the component to the store

Implement Redux in React

```
import React, { Component } from 'react';
import { connect } from 'react-redux';

class Comp2 extends Component {
  render() {
    return (
      <div className="App">
       Comp2
       {console.log(this.props.data.counter)}
      </div>
    );
  }}
const mapStateToProps = (state) => {
    return {
        data: state
    }
}
export default connect(mapStateToProps)(Comp2);
```

A Component “registered” to any changes in the store

The “mapStateToProps” maps current state to the component props