

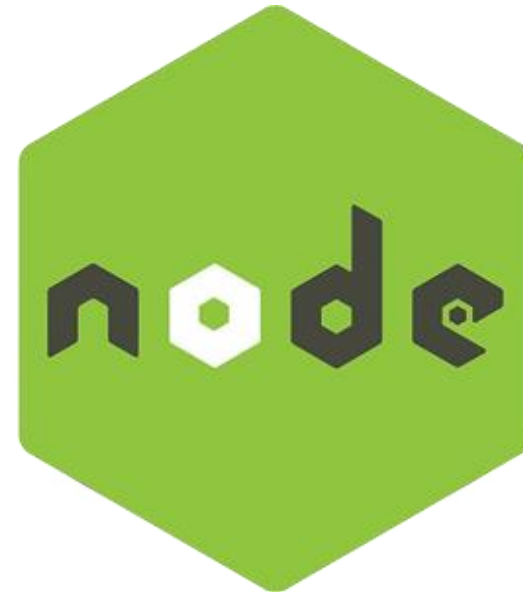
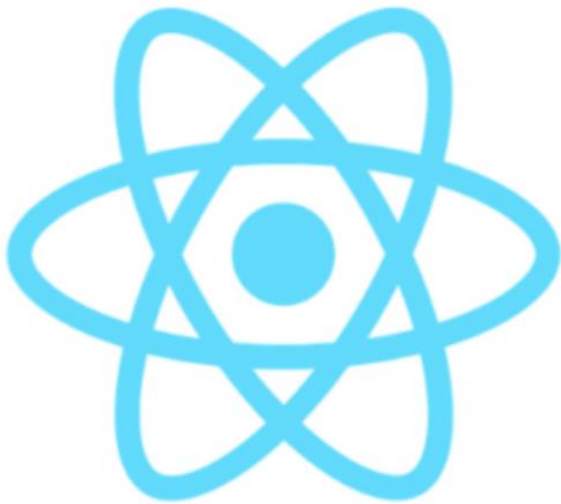


# Full-stack Application Development

## Introduction to React

# Where to Find The Code and Materials?

<https://github.com/iproduct/react-typescript-academy-2022>



# Agenda

1. MVC flavours
2. Single Page Applications (SPA)
3. SIMPLE Webpack Project Bootstrapping
4. Why React - simple and superfast, component oriented development using pure JavaScript (ES 6), virtual DOM, one-way reactive data flow, MVC framework agnostic
5. React by example – JSX syntax
6. React by example – JavaScript syntax
7. Lets do some code :)
8. Top level API
9. ES6 class syntax

# Agenda

10. JSX in depth – differences with HTML, transformation to JavaScript, namespaced components,
11. Expressions, child expressions and comments, props mutation anti-pattern, spread attributes, using HTML entities, custom attributes, if-else, immediately-invoked function expressions.
12. React Components Lifecycle Callbacks and ES6 class syntax
13. Events in React, managing DOM events
14. Components composition in depth – ownership, [\*this.props.children\*](#), [\*React.Children\*](#) utilities, child reconciliation, stateful children and dynamic children using keys
15. Transferring props

# MVC Comes in Different Flavors



What is the difference between following patterns:

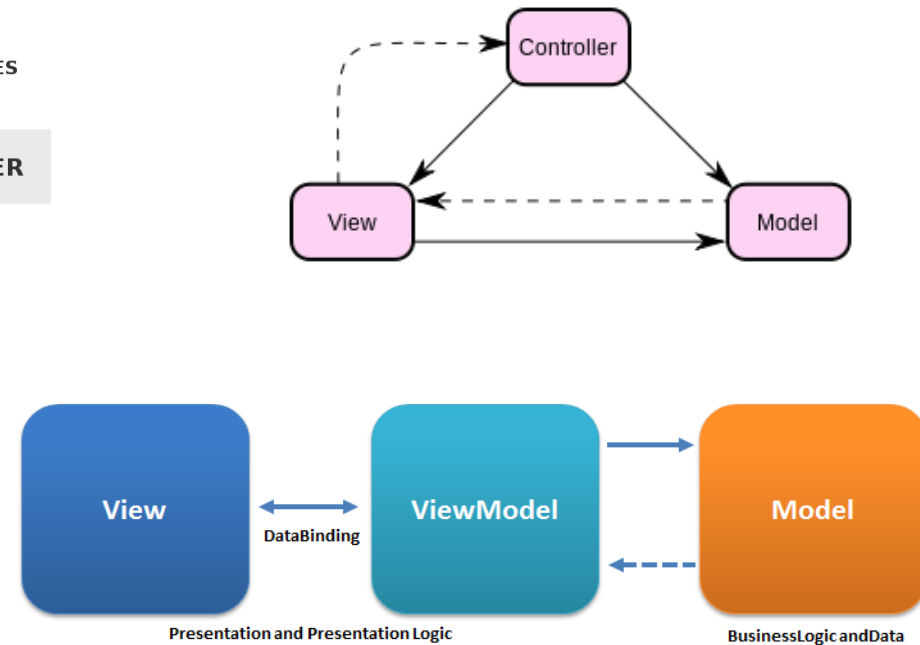
- Model-View-Controller (MVC)
- Model-View-ViewModel (MVVM)
- Model-View-Presenter (MVP)

# MVC Comes in Different Flavors - 2

- MVC



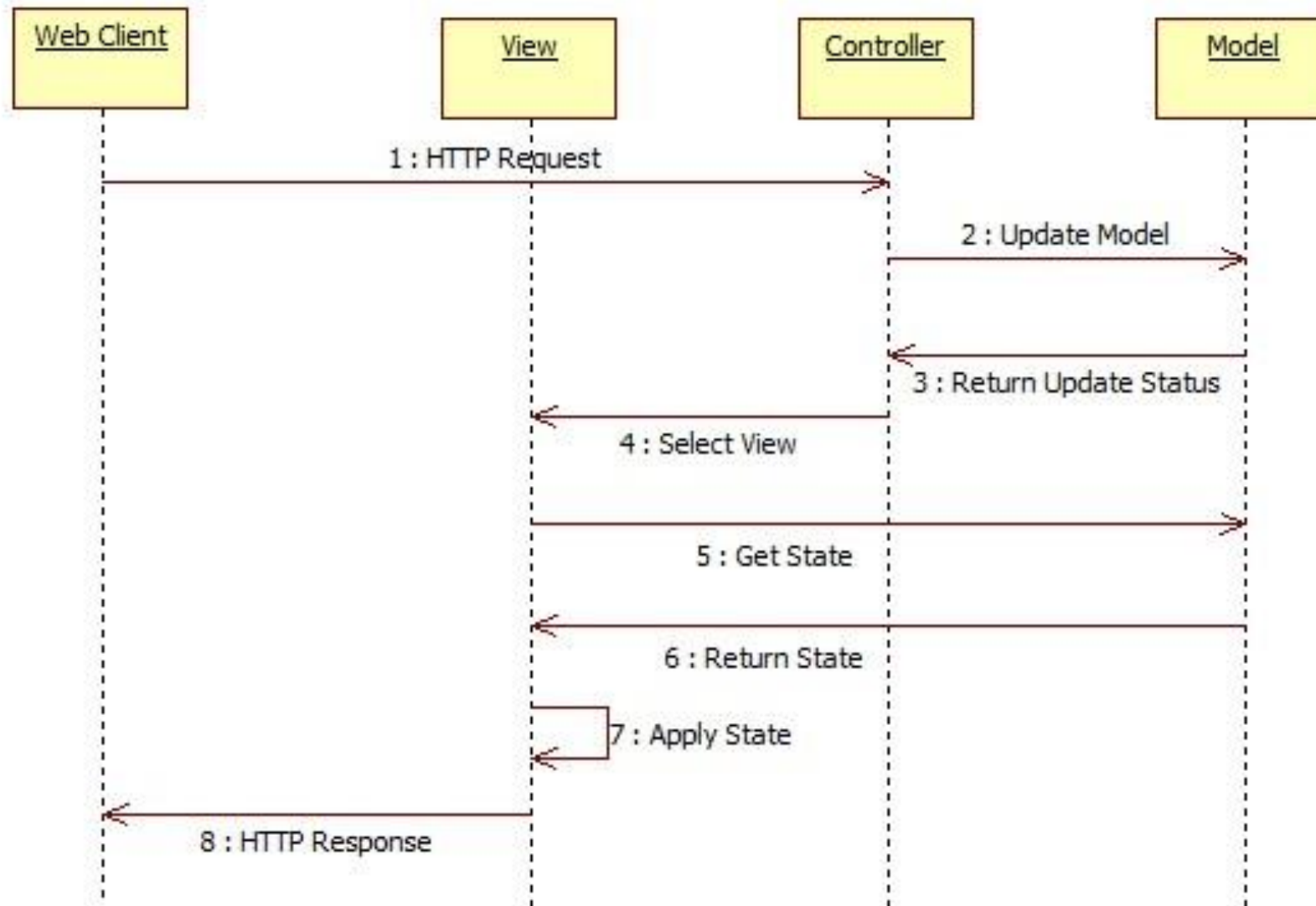
- MVVM



- MVP

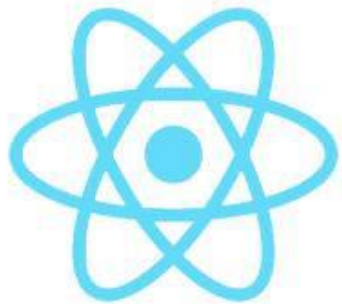


# Web MVC Interactions Sequence Diagram



# Why React?

- **React.js** is a JavaScript library for creating user interfaces by Facebook and Instagram – the V in MVC.
- **Solves well one problem**: building large applications with data that changes over time
- Simple and **superfast** – one-way reactive data flow

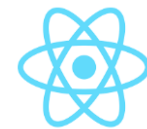


# React



# Why React?

- **Declarative** and **one-way reactive data flow** – simply express how your app should look, and React will automatically manage all UI updates when your underlying data changes
- **Component oriented SPA** development using pure JavaScript (**ES 6**) – React is all about building composable and reusable components - code reuse, testing, and separation of concerns
- **Virtual DOM** – allows decoupling of components from DOM, rendering done as last step
- Allows **isomorphic (client + server side) rendering**
- **MVC framework agnostic** – Flux, Redux, Reflux, ...
- Available at: <https://facebook.github.io/react>



# React.js by Example – JSX Syntax

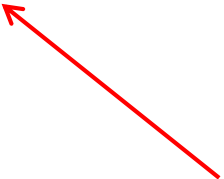
```
import React from "react";

import ReactDOM from "react-dom";

import Hello from "./hello";

ReactDOM.render(
  <Hello name="World" />,
  document.getElementById('app')
);
```

```
import React from "react";
export default
class Hello extends React.Component
{
  render() {
    return (
      <div className="hello">
        <h2>
          Hello, {this.props.name}!
        </h2>
      </div>
    );
  }
}
```



JavaScript syntax extension (JSX)  
that looks similar to XML

# Hello React TypeScript Example - index.tsx

- `npx create-react-app 07-ts-react --template typescript`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import reportWebVitals from './reportWebVitals';
import TodoAppFunction from './TodoAppFunction';

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <TodoAppFunction />
  </React.StrictMode>
);
```

# Hello React TypeScript Example – App.tsx (Class Component)

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';
interface AppProps {
  name: string;
}
class AppClass extends Component<AppProps, {}> {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <p>
            Edit <code>src/App.tsx</code> and save to reload.
          </p>
          <h2>Hello {this.props.name}!</h2>
        </header>
      </div>
    );
  }
}
```

# Hello React TypeScript Example – App.tsx (Function Component)

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
interface AppProps {
  name: string;
}
function AppFunction({ name }: AppProps) {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</code> and save to reload.
        </p>
        <h2>Hello {name}!</h2>
      </header>
    </div>
  );
}
export default AppFunction;
```

# Hello React TypeScript Example – App.tsx (Fat Arrow Function)

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
interface AppProps {
  name: string;
}
const AppLambda: React.FC<AppProps> = ({ name }) => {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</code> and save to reload.
        </p>
        <h2>Hello {name}!</h2>
      </header>
    </div>
  );
}
export default AppLambda;
```

# Hello React TypeScript Example – Hello.tsx (Class Component)

```
import React from 'react';
```

```
export interface HelloProps {  
  compiler: string;  
  framework: string;  
}
```

*// 'HelloProps' describes the shape of props. State is never set so we use the '{}' type.*

```
export class Hello extends React.Component<HelloProps, {}> {  
  render() {  
    return (  
      <div>  
        <h1>Hello from {this.props.compiler} and {this.props.framework}!!! </h1>  
      </div>  
    );  
  }  
}
```

# React TypeScript Component with PropTypes Runtime Validation

```
import React from 'react';
import PropTypes from 'prop-types';

export interface HelloProps {
  compiler: string;
  framework: string;
}

export class Hello extends React.Component<HelloProps, {}> {
  static propTypes = {
    compiler: PropTypes.string.isRequired,
    framework: PropTypes.string.isRequired,
  };
  render() {
    return (
      <div>
        <h1>Hello from {this.props.compiler} and {this.props.framework}!</h1>
      </div>
    );
  }
}
```



# Comments Demo Example – Pure JavaScript

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

let CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am new CommentBox."
      )
    );
  }
});

ReactDOM.render(
  React.createElement(CommentBox, null),
  document.getElementById('app')
);
```

# Lets Do Some React Code :)

React comment box example available @GitHub:

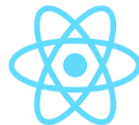
<https://github.com/reactjs/react-tutorial>

React.js documentation and API:

<https://facebook.github.io/react/docs>

# Top Level API

- **React** – the entry point to the React library. If you're using one of the prebuilt packages it's available as a global; if you're using CommonJS modules you can `require()` it.
- **ReactDOM** – provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside of the React model if you need to. Most of your components should not need to use this module.
- **ReactDOMServer** – the `react-dom/server` package allows you to render your components on the server: **`ReactDOMServer.renderToString(ReactElement element)`**
- @: <https://facebook.github.io/react/docs/top-level-api.html>



# React ES6 Demo Example

```
export class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
    this.tick = this.tick.bind(this);
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div onClick={this.tick}>
        Clicks: {this.state.count}
      </div>
    );
  }
}
Counter.propTypes = { initialCount: PropTypes.number };
Counter.defaultProps = { initialCount: 0 };
```

# React Components

- **Virtual DOM** – everything is a component (e.g. `<div>` in JSX), rendering done as last step
- **Components are like functions** – of three arguments:
- **this.props** – these is the external interface of the component, passed as attributes – allow the parent component (“owner”) to pass state and behavior to embedded (“owned”) components. Should never be mutated within component – immutable.
- **this.props.children** – part of the component interface but passed in the body of the component (component tag)
- **this.state** – internal state of the component should be mutated only using **React.Component.setState(nextState, [callback])**

# React Components as Pure Functions

```
function HelloMessage(props) {  
    return <div>Hello {props.name}</div>;  
}
```

```
ReactDOM.render(<HelloMessage name="React User" />, mountNode);
```

- OR using ES6 => syntax:

```
const HelloMessage = (props) => <div>Hello {props.name}</div>;
```

```
ReactDOM.render(<HelloMessage name="React User" />, mountNode);
```

# What Components Should Have State?

- Most components should just render data from props. However, sometimes you need to **respond to user input**, a **server request** or the **passage of time** => then use state.
- Try to keep as many of your components as possible **stateless** – makes easier to reason about your application
- Common pattern: create several **stateless components** that just **render data**, and have a **stateful component above them** in the hierarchy that **passes its state to its children via props**.
- **Stateful component** encapsulates all of the **interaction logic**
- **Stateless components** take care of **rendering data in a declarative way**

# JSX Syntax

- With JSX: `<a href="https://facebook.github.io/react/">Hello!</a>`
- In pure JS: `React.createElement('a',  
 {href: 'https://facebook.github.io/react/'}, 'Hello!')`
- JSX is optional – we could write everything without it, but it is not very convenient:

```
var child1 = React.createElement('li', null, 'First Text Content');
```

```
var child2 = React.createElement('li', null, 'Second Text Content');
```

```
var root = React.createElement('ul', { className: 'my-list' }, child1, child2);
```



# JS Syntax Using Factories

- We can use factories to simplify the component use from JS:

```
var Factory = React.createFactory(ComponentClass);
```

```
...
```

```
var root = Factory({ custom: 'prop' });
```

```
ReactDOM.render(root, document.getElementById('example'));
```

- For standard components like <div> there are factories built-in:

```
var root = React.DOM.ul(  
    { className: 'my-list' },  
    React.DOM.li(null, 'Text Content')  
);
```

# JS Syntax in Depth

- Since **JSX is JavaScript**, identifiers such as **class** and **for** are discouraged as XML attribute names. Instead, React DOM components expect DOM property names like **className** and **htmlFor**, respectively.

```
var myDivElement = <div className="foo" />;
```

```
ReactDOM.render(myDivElement, document.getElementById('example'));
```

- To render it use **Uppercase variable** → **comp.displayName**:

```
var MyComponent = React.createClass({/*...*/});
```

```
var myElm = <MyComponent someProperty={true} />;
```

```
ReactDOM.render(myElm, document.getElementById('example'));
```

# JavaScript Expressions

- Attribute Expressions:

```
var person = <Person name= {app.isLoggedIn ? app.currentUser : ""} />;
```

- Boolean Attributes:

```
<input type="button" disabled />;
```

```
<input type="button" disabled={true} />;
```

- Child Expressions:

```
var content = <Container>
```

```
    {app.isLoggedIn ? <Nav /> : <Login />}
```

```
</Container>;
```

# JSX Spread Attributes

- Mutating props is bad – should be treated as immutable
- Spread Attributes:

```
var props = {};
```

```
props.foo = x;
```

```
props.bar = y;
```

```
var component = <Component {...props} />;
```

- Order is important – property value overriding:

```
var props = { foo: 'default' };
```

```
var component = <Component {...props} foo={'override'} />;
```

```
console.log(component.props.foo); // 'override'
```

# HTML Entities in JSX

- Double escaping (all content is escaped by default – XSS):

`<div>First &middot; Next</div>` - **OK**

`<div>{'First &middot; Next'}</div>` - **Double escaped**

- Solution 1: type (and save) it in UTF-8:

`<div>{'First · Next'}</div>`

- Solution 2: use Unicode

`<div>{'First \u00b7 Next'}</div>`

`<div>{'First ' + String.fromCharCode(183) + ' Next'}</div>`

- Solution 3: use mixed arrays with strings and JSX elements:

`<div>['First ', <span key="middot">&middot;</span>, 'Next']</div>`

- Solution 4 (last resort): type (and save) it in UTF-8:

`<div dangerouslySetInnerHTML={{__html: 'First &middot; Next'}} />`

# Custom Attributes in JSX

- If you pass properties to native HTML elements that do not exist in the HTML specification, React will not render them.

- Custom attributes - should be prefixed with **data-** :

```
<div data-custom-attribute="foo" />
```

- Custom elements (with a hyphen in the tag name) support arbitrary attributes:

```
<x-my-component custom-attribute="foo" />
```

- Web Accessibility attributes starting with **aria-** are rendered:

```
<div aria-hidden={true} />
```

# Immediately-Invoked Function Expressions

```
return (  
  <section>  
    <h1>Color</h1>  
    <h3>Name</h3> <p>{this.state.color || "white"}</p>  
    <h3>Hex</h3><p>  
      {(() => {  
        switch (this.state.color) {  
          case "red": return "#FF0000";  
          case "green": return "#00FF00";  
          default: return "#FFFFFF";  
        }  
      })()}  
    </p>  
  </section>  
);
```

# Events in React

- **SyntheticEvent(s)** - event handlers are passed instances of SyntheticEvent – cross-browser wrapper around native events
- Same interface: **stopPropagation()**, **preventDefault()**
- **Event pooling** – all SyntheticEvent(s) were pooled in React before v17 (objects were reused and all properties were nullified after the event callback has been invoked)
- **From React v 17 no event pooling happens**
- Example: <https://facebook.github.io/react/docs/events.html>
- Event types: **Clipboard, Composition, Keyboard, Focus, Form, Mouse, Selection, Touch, UI Events, Wheel, Media, Image, Animation, Transition**
- Handlers in React are written in Camel Case – e.g.: **onClick, onSubmit**



# Transferring Props

```
function FancyCheckbox(props) {  
  let { checked, ...other } = props;  
  let fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';  
  // `other` contains { onClick: console.log } but not the checked property  
  return (<div {...other} className={fancyClass} />);  
}
```

```
ReactDOM.render(  
  <FancyCheckbox checked={true} onClick={console.log.bind(console)}>  
    Hello world!  
  </FancyCheckbox>,  
  document.getElementById('example')  
>);
```

# React Hooks – New in React 16!

[\[https://reactjs.org/docs/hooks-intro.html\]](https://reactjs.org/docs/hooks-intro.html)

- **Hooks** are a new addition in React 16.8. They let you use state and other React features without writing a class.
- **Basic Hooks**
  - **useState**: `const [state, setState] = useState(initialState);`
  - **useEffect**: `useEffect(() => {  
 const subscription = props.source.subscribe();  
  
 return () => { subscription.unsubscribe() };  
});`
  - **useContext** – allows to access resources application wide
- **Additional Hooks** – `useReducer`, `useCallback`, `useMemo`, `useRef`, `useImperativeHandle`, `useLayoutEffect`, `useDebugValue` – will be discussed later during the course

# React Hooks Example: *useState*

```
const GOOLE_BOOKS_API_BASE = "https://www.googleapis.com/books/v1/volumes?q=";

function App() {
  const [books, setBooks] = useState(mockBooks);
  return (
    <React.Fragment>
      <Nav searchBooks={onSearchBooks} />
      <div className="section no-pad-bot" id="index-banner">
        <div className="container">
          <Header />
          <BookList books={books} />
        </div>
      </div>
      <Footer />
    </React.Fragment>
  );

  async function onSearchBooks(searchText) {
    const booksResp = await fetch(GOOLE_BOOKS_API_BASE + encodeURIComponent(searchText));
    const booksFound = await booksResp.json();
    console.log(booksFound.items);
    setBooks(booksFound.items.map(gbook => ({
      'id': gbook.id,
      'title': gbook.volumeInfo.title,
      'subtitle': gbook.volumeInfo.subtitle,
      'frontPage': gbook.volumeInfo.imageLinks && gbook.volumeInfo.imageLinks.thumbnail
    })));
  }
}
```

# React Hooks Example: *useEffect*

- `import { useState, useEffect } from 'react';`

```
function useFriendStatus(friendID) {  
  const [isOnline, setIsOnline] = useState(null);  
  
  useEffect(() => {  
    function handleStatusChange(status) {  
      setIsOnline(status.isOnline);  
    }  
  
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);  
    return () => {  
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);  
    };  
  }, [friendID]);  
  
  return isOnline;  
}
```

# Rules of Hooks

- **Only Call Hooks at the Top Level** - don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function. By following this rule, you ensure that Hooks are called in the same order each time a component renders. That's what allows React to correctly preserve the state of Hooks between multiple `useState` and `useEffect` calls.
- **Only Call Hooks from React Functions** - don't call Hooks from regular JavaScript functions. Instead, you can:
  - ✓ Call Hooks from React function components.
  - ✓ Call Hooks from custom Hooks (we'll learn about them on the next slide)..

# Custom Hooks

```
import { useState, useEffect } from 'react';
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  }, [friendID]);
  return isOnline;
}
```

# React Component Lifecycle Callbacks (1)

- React components **lifecycle** has 3 phases:
  - **Mounting**: A component is being inserted into the DOM.
  - **Updating**: A component is being re-rendered to determine if the DOM should be updated.
  - **Unmounting**: A component being removed from the DOM.
- **Mounting** lifecycle callbacks:

**constructor()**

**static getDerivedStateFromProps()** - if the state depends on changes in props

**render()**

**componentDidMount()** - is invoked immediately after mounting occurs. Initialization that requires DOM nodes should go here.

# React Component Lifecycle Callbacks (2)

**static `getDerivedStateFromProps(props, state)`** - invoked right before calling the render method, both on the initial mount and on subsequent updates. It should **return an object to update the state, or null to update nothing**. It enables a component to **update its internal state as the result of changes in props**.

- Examples:
  - recording the current scroll direction based on a changing offset prop
  - loading external data specified by a source prop
  - reading DOM properties before an update



# Simpler Alternatives to `getDerivedStateFromProps`

- If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use [componentDidUpdate](#) lifecycle instead.
- If you want to **re-compute some data only when a prop changes**, use a [memoization helper](#) instead.
- If you want to **“reset” some state when a prop changes**, consider either making a component [fully controlled](#) or [fully uncontrolled](#) with a key instead.

# Updating Lifecycle Callbacks

**static getDerivedStateFromProps(props, state)**

**shouldComponentUpdate(object nextProps, object nextState): boolean** – invoked when a component decides whether to update - optimization comparing **this.props** with **nextProps** and **this.state** with **nextState** and return **false** if React should skip updating.

**render()**

**getSnapshotBeforeUpdate(prevProps, prevState)** - invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle will be passed as a parameter to `componentDidUpdate()`.

**componentDidUpdate(object prevProps, object prevState)** – invoked after updating occurs.

# Unmounting Lifecycle Callbacks:

- **Unmounting**

**componentWillUnmount()** – invoked immediately before a component is unmounted and destroyed. Cleanup should go here.

- **Error Handling**

**static getDerivedStateFromError(error)** - invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

**componentDidCatch(error, info)** - invoked after an error has been thrown by a descendant component

- **Mounted** composite components also support:

**component.forceUpdate()** – can be invoked on any mounted component when you know that some deeper aspect of the component's state has changed without using **this.setState()**.

# Forms in React – Controlled Components

- **Interactive Props** - form components support a few props that are affected via user interactions:
  - value - supported by `<input>` and `<textarea>` components
  - checked - supported by `<input>` of type checkbox or radio
  - selected - supported by `<option>` components
- Above form components allow listening for changes by setting a callback to the **onChange** prop:

```
handleAuthorChange(e) { this.setState({author: e.target.value}); }
```

```
<input type="text" value={this.state.author} placeholder="Your name"  
      onChange={this.handleAuthorChange}/>
```

- **Controlled component** does not maintain its own internal state – the component renders purely based on **props**

# Component Properties Validation (1)

```
import PropTypes from 'prop-types';
```

```
MyComponent.propTypes = {
```

```
// Optional basic JS type properties
```

```
  optionalArray: PropTypes.array,
```

```
  optionalBool: PropTypes.bool,
```

```
  optionalFunc: PropTypes.func,
```

```
  optionalNumber: PropTypes.number,
```

```
  optionalObject: PropTypes.object,
```

```
  optionalString: PropTypes.string,
```

```
  optionalSymbol: PropTypes.symbol,
```

```
}
```

# Component Properties Validation (2)

// Anything that can be rendered: numbers, strings, elements or  
// an array (or fragment) containing these types.

optionalNode: PropTypes.node,

// A React element.

optionalElement: PropTypes.element,

// You can also declare that a prop is an instance of a class.

optionalMessage: PropTypes.instanceOf(Message),

# Component Properties Validation (3)

// You can ensure that your prop is limited to specific enum.

```
optionalEnum: PropTypes.oneOf(['News', 'Photos']),
```

// An object that could be one of many types

```
optionalUnion: PropTypes.oneOfType([  
  PropTypes.string,  
  PropTypes.number,  
  PropTypes.instanceOf(Message)  
]),
```

# Component Properties Validation (4)

// An array of a certain type

optArray: PropTypes.arrayOf(React.PropTypes.number),

// An object with property values of a certain type

optObject: PropTypes.objectOf(React.PropTypes.number),

// An object taking on a particular shape

optionalObjectWithShape: PropTypes.shape({

  color: PropTypes.string,

  fontSize: PropTypes.number

}),



# Component Properties Validation (5)

// You can chain any of the above with `isRequired`

requiredFunc: PropTypes.func.isRequired,

// A required value of any data type

requiredAny: PropTypes.any.isRequired,

// You can also specify a custom validator => return an Error

customProp: function(props, propName, componentName) {

if (!/matchme/.test(props[propName])) {

return new Error('Invalid prop `' + propName + '` supplied to' +

` + componentName + `'. Validation failed.'

); }} /\* ... \*/

# Component Ownership

- **Multiple Components** – allow separation of concerns and reusability
- **Ownership** – an owner is the component that **sets the props** of owned components.
- When a component **X** is created in component **Y**'s **render()** method, it is said that **X** is owned by **Y**.
- Only defined for React components – different from parent-child DOM relationship.
- **Child Reconciliation** – the process by which React updates the DOM with each new render pass. In general, children are reconciled according to the order in which they are rendered.

# Reconciliation Example

- // Render Pass 1
  - <Card>
    - <p>Paragraph 1</p>
    - <p>Paragraph 2</p>
  - </Card>
- // Render Pass 2
  - <Card>
    - <p>Paragraph 2</p>
  - </Card>

# Stateful Children Reconciliation – Keys

```
var ListItemWrapper = React.createClass({
  render: function() {
    return <li>{this.props.data.text}</li>;
  }
});

var MyComponent = React.createClass({
  render: function() {
    return (
      <ul>
        {this.props.results.map(function(result) {
          return <ListItemWrapper key={result.id} data={result}/>;
        })}
      </ul>
    );
  }
});
```

# React.Children Utilities

- **React.Children.map**(object children, function fn [, object thisArg]): array – invoke **fn** on every immediate child contained within **children** with **this** set to **thisArg**
- **React.Children.forEach**(object children, function fn [, object thisArg]) – same as **map**, but does not return an array
- **React.Children.count**(object children): number - returns children count
- **React.Children.only**(object children): object – returns the only child in children. Throws otherwise
- **React.Children.toArray**(object children): array – returns the children as a flat array with keys assigned to each child

# Refs to Components

- **Refs (references)** – allow to find the **DOM markup** rendered by a component, and invoke methods on **component instances** returned from **render()**
- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.
- Using React components in larger non-React applications, transition existing code to React.
- Avoid using refs for anything that can be done declaratively. For example, instead of exposing **open()** and **close()** methods on a Dialog component, pass an **isOpen** prop to it.
- **Not Recommended!:**
  - `var myComponentInstanceRef = ReactDOM.render(<MyComp />, myContainer); myComponentInstanceRef.doSomething();`
  - `ReactDOM.findDOMNode(componentInstance)` – this function will return the DOM node belonging to the outermost HTML element returned by render.

# Accessing Refs

<https://reactjs.org/docs/refs-and-the-dom.html#accessing-refs>

- When a ref is passed to an element in render, a reference to the node becomes accessible at the **current** attribute of the ref:

```
const node = this.myRef.current;
```

- The **value of the ref** differs depending on the type of the node:
  - When the ref attribute is used on an **HTML element**, the ref created in the constructor with **React.createRef()** receives the **underlying DOM element** as its current property.
  - When the ref attribute is used on a **custom class component**, the ref object receives the **mounted instance of the component** as its current.
  - **You may not use the ref attribute on function components because they don't have instances.**

# Using References with Class Component

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }
  handleSubmit(event) { alert('A name was submitted: ' + this.input.current.value); event.preventDefault(); }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```



# Using References with Function Component

```
function App() {  
  const inputRef = useRef();  
  function focusTextInput() {  
    // Explicitly focus the text input using the raw DOM API  
    // Note: we're accessing "current" to get the DOM node  
    // this.textInput.current.focus();  
    inputRef.current.focus();  
  }  
  return (  
    <div>  
      <input type="text" ref={inputRef} />  
      <input  
        type="button"  
        value="Focus the text input"  
        onClick={focusTextInput}  
      />  
    </div>  
  );  
}
```

# From Classes to Hooks [<https://reactjs.org/docs/hooks-faq.html>]

- **constructor**: Function components don't need a constructor. You can initialize the state in the `useState` call. If computing the initial state is expensive, you can pass a function to `useState`.
- **getDerivedStateFromProps**: Schedule an update while rendering instead.
- **shouldComponentUpdate**: See `React.memo` below.
- **render**: This is the function component body itself.
- **componentDidMount, componentDidUpdate, componentWillUnmount**: The `useEffect` Hook can express all combinations of these (including less common cases).
- **getSnapshotBeforeUpdate, componentDidCatch and getDerivedStateFromError**: There are no Hook equivalents for these methods yet, but they will be added soon.

# Using React Component Context

- **React props** allow to track data-flow easy between components
- React Context is alternative if you want to pass data through the component tree **without having to pass the props down manually at every level.**
- **Inversion of Control (IoC)** principle and **Dependency Injection (DI)** pattern.
- React's "context" feature lets you do this.

# 1. Creating React Context

```
import React from 'react';
export const themes = {
  light: {
    name: 'Light Theme',
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    name: 'Dark Theme',
    foreground: '#ffffff',
    background: '#222222',
  },
};
```

*//1. Create context: Context lets us pass a value deep into the component tree  
// without explicitly threading it through every component.  
// Create a context for the current theme (with "light" as the default).*

```
export const ThemeContext = React.createContext(themes.light);
ThemeContext.displayName = 'ThemeContext';
```

## 2. Providing React Context - I

```
class App extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      theme: themes.dark,  
      user: { name: 'Site Admin' },  
    };  
  }  
}
```

```
toggleTheme = () => {  
  this.setState((state) => ({  
    theme: state.theme === themes.dark ? themes.light : themes.dark,  
  }));  
};
```

## 2. Providing React Context -II

```
render() {  
  // Use a Provider to pass the current theme to the tree below.  
  // Any component can read it, no matter how deep it is.  
  // In this example, we're passing "dark" as the current value.  
  return (  
    //2. Provide value  
    <React.Fragment>  
      <UserContext.Provider value={this.state.user}>  
        <ThemeContext.Provider value={this.state.theme}>  
          <Toolbar changeTheme={this.toggleTheme} />  
        </ThemeContext.Provider>  
      </UserContext.Provider>  
      <Toolbar changeTheme={this.toggleTheme} />  
    </React.Fragment>  
  );  
}
```

### 3. Consuming React Context

```
function Toolbar(props) {  
  return (  
    <UserContext.Consumer>  
      {(user) => (  
        <ThemeContext.Consumer>  
          {(theme) => (  
            <React.Fragment>  
              <h3>Theme: {theme.name}, Logged User: {user.name}</h3>  
              <ThemedButton onClick={props.changeTheme}>Change Theme</ThemedButton>  
            </React.Fragment>  
          )}  
        </ThemeContext.Consumer>  
      )}  
    </UserContext.Consumer>  
  );  
}
```

# Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>