React Components – Basic Idea

How to Compose in React ?

SoftUni Team
Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

Have a Question?

Software University

sli.do

#react

Components Overview

Syntax, Functional and Class Components

Components Overview

- Components let you
  - Split the UI into **independent** and **reusable pieces**
  - Think about **isolation**
- React let you define components as
  - **Functions**
  - **Classes**

Functional Component

- **Functional component** is a JS function which
  - Accepts single argument called **props** (object with data)
  - Returns a **React Element**

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

Class Component

- To define a **React component class**, you need to extend **React.Component**

```
class Person extends React.Component {
  render() {
    return <h1>My name is {this.props.name}</h1>
  }
}
```

  - The only method you must define is called **render()**

Component Syntax

- Names always start with **UpperCase**
- Tags always must be **closed**
- **Information** is passed via **props**

```
<Dropdown> A dropdown list
  <UserHead name="homeHeader" />
  <Menu>
    <MenuItem>Do Something</MenuItem>
    <MenuItem>Do Something Fun!</MenuItem>
  </Menu>
</Dropdown>
```

Component Props and State

Overview

Props and State Overview

- In React **props** and **state** represent the rendered values
- Both are plain JavaScript **objects**
- Both hold information that influences the output of render

Props and State Overview

- They are different in one important way
  - **Props** get passed to the component (like function params)
  - **State** is managed within the component (like local variables)

Component Props

Passing Data, Access and Usage

Component Props

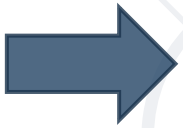
- **Props** is short for **properties**
  - Are received from above (parent)
  - **Immutable** as far as the component receiving them is concerned
- A component **cannot change** its own props, but it is responsible for putting together the props of its child components

Passing Props to Nested Components

- We use props to **pass data** from parent to child

```
const BookList = () => {
  return (
    <ul>
      <Book
        title="IT"
        author="Stephen King"
        price="20"
      />
      <Book
        title="The Hunger Games"
        author="Suzanne Collins"
        price="10"
      />
    </ul>
  );};
```

Prop name should start with lowercase letter



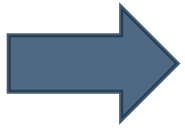
Use className to set css classes

```
const Book = (props) => {
  return (
    <li className="book">
      <div>{props.title}</div>
      <div>{props.author}</div>
      <div>{props.price}</div>
    </li>
  );
};
```

Children Property

- Use **children** property to access information between **opening** and **closing** tags

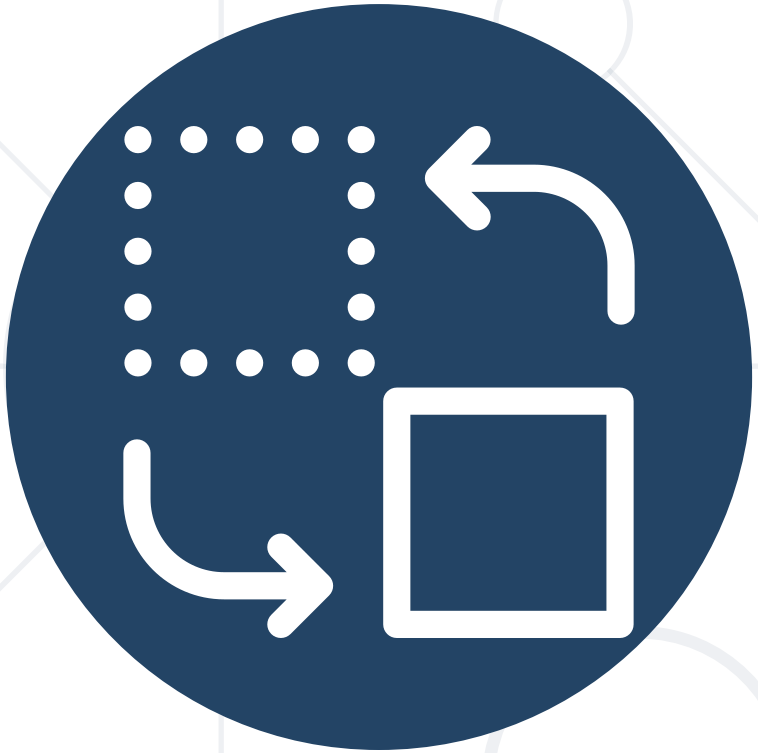
```
const BookList = () => {
  return (
    <ul>
      <Book
        title="IT"
        author="Stephen King"
        price="20">
        <span>
          Some value here
        </span>
      </Book>
    </ul>
  );};
```



```
const Book = (props) => {
  return (
    <li className="book">
      …
        <div>{props.children}</div>
    </li>
  );
};
```

Could be plain text or nested HTML

Props Demo



Storing and Modifying Data

Component State

Component State Overview

- The heart of every React component is its "**state**"
  - It determines how the component **renders** and **behaves**
  - State allows you to create components that are **dynamic** and **interactive**

State

- **State** starts with default value when a component mounts
  - After mounts, suffers from **mutations** in time
  - Its **serializable**
- Component manages its own state internally0

Component State Example

- State holds information that **can change** over time
  - Usually as a result of **user input** or **system events**

```
function Timer(props) {
  const [time, setTime] = React.useState(0);

  setTimeout(() => {
    setTime(time + 1);
  }, 1000);

  return <h1>{time} sec.</h1>;
}
```

State Hook

State Hook

- Hook is a special function that lets you "hook into" React features
  - `useState` is a Hook that lets you add **React state** to function components
  - You don't have to convert functional component into class to use state

State Hook

- Calling **useState** inside functional component to add some local state to it

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

- React will preserve this state between re-renders
- **useState** returns a pair `const [count, setCount] = useState(0);`
  - Current state **value**
  - **Function** that lets you update it

State Hook

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

const counter = () {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Counter: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

State Hook

- You can call the update function from anywhere
- It's similar to **this.setState** in class, except it **doesn't merge** the **old** and **new** state together
- The only argument to **`useState`** hooks is the **initial state**
  - Unlike **this.state**, here doesn't have to be an object
    - Although it can be if you want

State Hook

- You can use the **State Hook** more than once in a single component

```
const registerComponent = () {
  const [email, setEmail] = useState("");
  const [age, setAge] = useState("0");
  const [password, setPassword] = useState("");
  // ...
}
```

- The initial state argument is only used during the **first render**

State Demo

Handling Events

Handling Events

- Handling events with React elements is very similar to handling event on **DOM elements**
- The syntactic differences are:
  - React events are named using **camelCase**
  - With JSX you pass a function as the **event handler**

Handling Events

- When using React you should generally
  - **Not** need to call **addEventListener** to add listeners to a DOM element after it is created
  - Just provide a listener when the element is initially rendered

```
<button onClick={this.clickHandler}
  Click me! I'm a counter
</button>
```

Handling Events

- There are two ways to passing arguments to event handlers
  - Using **arrow functions**

```
<button onClick={(e) => this.deleteRow(id, e)}>
  Delete Row
</button>
```

  - Using **bind**

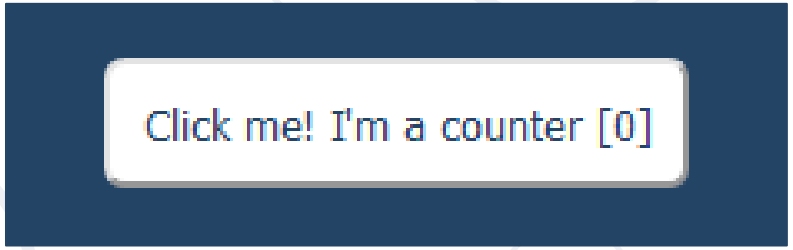
```
<button onClick={this.deleteRow.bind(this, id)}>
  Delete Row
</button>
```

Handling Events

```
const [clicks, setClicks] = useState(0);
clickHandler = () => {
   setClicks(c => c + 1)
}
```

```
<Button
  clickHandler={clickHandler}
  clicks={clicks}
/>
```

```
<button className="counter"
  onClick={props.clickHandler}>
    Click me! I'm a counter [{props.clicks}]
</button>
```

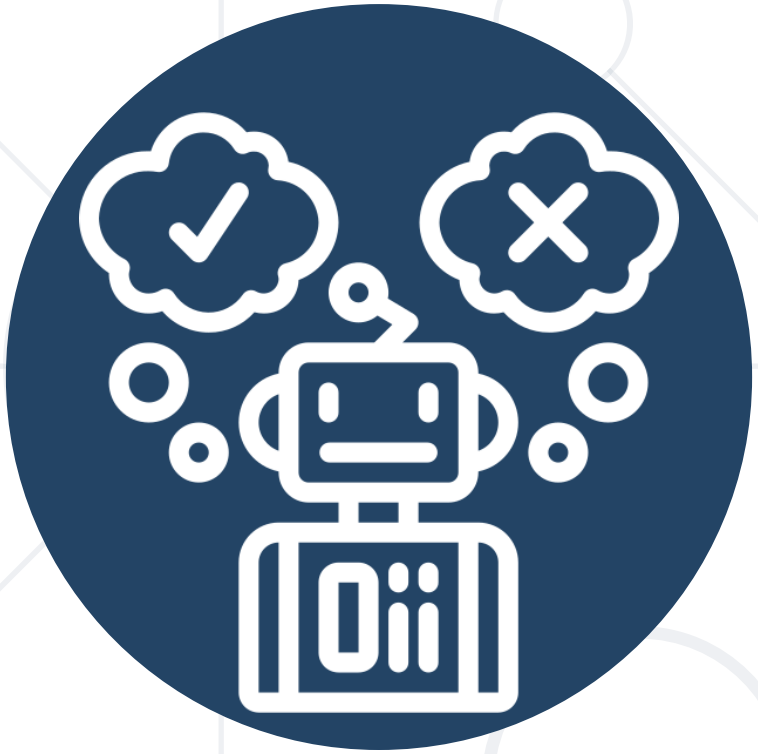


SyntheticEvent

- Event handlers will be passed instances of `SyntheticEvent`
  - It has the same interface as the browser's native event
    - Including `stopPropagation()` and `preventeDefault()`
    - Except the events work identically across all browsers

```
function onClick(event) {
  console.log(event);
  console.log(event.type);
  const eventType = event.type;
}
```

Handling Events Demo



Conditional Rendering

Conditional Rendering

- Conditional rendering in React works the same way conditions work in JavaScript using:
  - Operators like **if**
  - Conditional (**ternary**) operators

Conditional Rendering

- Using **if** operator

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
    if (isLoggedIn) {
      return <UserGreeting />;
    }
  return <GuestGreeting />
}
```

Conditional Rendering

- Using **ternary** operator

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}

function Greeting(props) {
  return (
    <div>
      { props.isLoggedIn ? < UserGreeting /> : <GuestGreeting /> }
    </div>
  )}
```

Conditional Rendering Demo

Debugging React Components with the Components Tab

Debugging React Components

- Debugging is a **critical** skill in development. It helps identify and fix issues within your code
- React applications can have **complex** component hierarchies. Debugging ensures components work as expected

Debugging React Apps Using Visual Studio Code

Software University

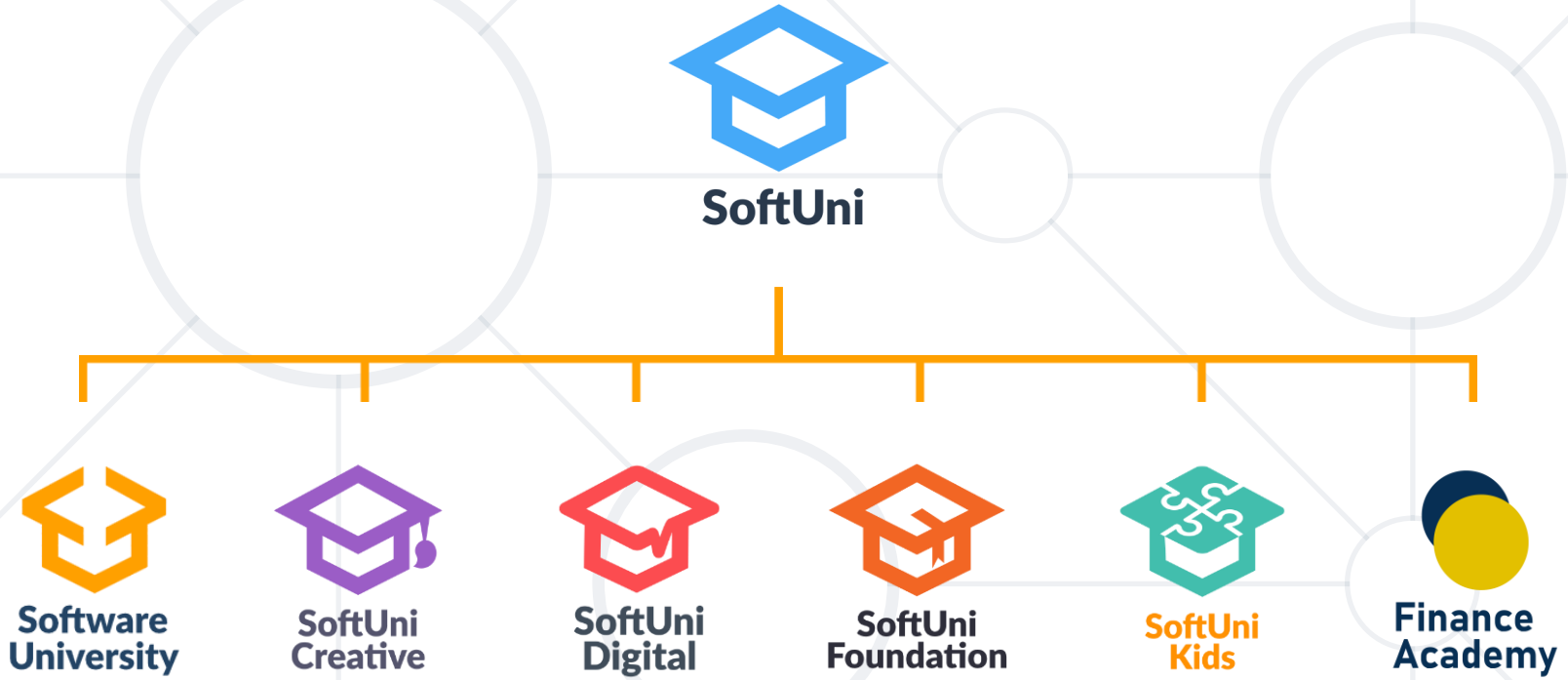
- Open your React app project folder in VS Code
- Click on the "**Run and Debug**"
- Select "**Web App (Chrome)**" for debugger
- Modify the generated "**launch.json**" by changing **URL** to match your app address
- Press **F5** button

```
"version": "0.2.0",
"configurations": [
    {
        "type": "chrome",
        "request": "launch",
        "name": "Launch Chrome against localhost",
        "url": "http://localhost:5173",
        "webRoot": "${workspaceFolder}"
    }
]
```

Summary

- **Components** reusable elements
  - **Functional** and **Class**
- **Props** are used to pass down data
- **State** is used to hold component data
- Handling Events in React
- Conditional Rendering
  - **If** and **ternary** operators

Questions?



SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter International

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank
Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg
- © Software University – https://softuni.bg