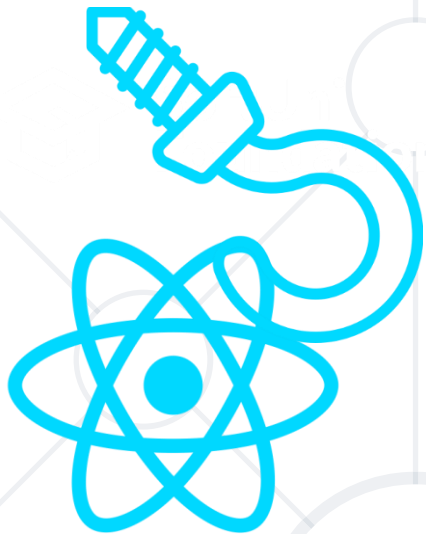


React Hooks

Introduction, State & Effect Hooks



SoftUni Teams
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

Table of Contents

1. Introduction
2. State Hook
3. Effect Hook
4. Rules of Hooks
5. Custom Hooks
6. Context API



sli.do

#react

React Timeline

March 2013

April 2015

February 2019

Timeline

March 2013: React v0.3.0 introduces the concept of components.

April 2015: React v0.13.0 introduces the `React.Component` base class, leading to the use of class components.

February 2019: React v16.8 introduces Hooks, allowing functional components to manage state and side effects.

Introduction

- React Hooks
 - JS functions which can be only used inside Functional Component or other Hooks
 - New feature in **React 16.8**
 - Let you **use state** and **other** React features **without** writing a class



- React Hooks have specified naming
 - Starting with lowercase: **"use "**
 - Followed by function name like: **"State"**
 - **useState, useEffect, useContext...**
- The basic idea is to **expose** stateful functionalities to functional component
 - **managing state**
 - **adding lifecycle methods**

- You can make **everything** work that you could make working class based components
- They are **highly re-usable** and **independent** for each component
 - Using hooks to **share functionality, NOT** data between components
- React hooks **have nothing** to do with Lifecycle Methods
 - Can't replace lifecycle methods with React hooks



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



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

```
import { 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

```
import { 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>
  );
}
```

- 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

- 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 Hook Demo



Effect Hook

Effect Hook

- You most likely perform: **data fetching, subscriptions** or **manually changing the DOM**
 - Operations like these are called **side effects**
 - They can **affect** other components and can't be done during the rendering
- **useEffect** hook adds the ability to perform side effects from a function component



- **useEffect** hook serves the same purpose as
 - **componentDidMount**
 - **componentDidUpdate**
 - **componentWillUnmount**
- But they are bundled into a single API

```
import { useEffect } from 'react';
```

- **useEffect** hook accepts a function that contains imperative, possibly effectful code
 - That function will run **after the render** is committed to the screen
- By default effects run after **every completed render**
 - But you can choose to fire them only when certain value have changed

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

const counter = () {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate
  useEffect(() => {
    document.title = `The counter reached: ${count} times`;
  });

  // ...
}
```

- When you call **useEffect** you're telling React to run your "effect" function after flushing changes to the DOM
- Effects are declared inside the component so they have access to its **props** and **state**
- Effects may also optionally specify how to "clean up" after them by **returning a function**

- Often, effects create resources that need to be **cleaned up** before the component leaves the screen
 - To do this, the function passed to **useEffect** may return a **clean-up function**

```
useEffect(() => {  
  const subscription = props.source.subscribe();  
  return () => {  
    // Clean up the subscription  
    subscription.unsubscribe();  
  };  
});
```



Effect Hook Demo



Custom Hooks

Custom Hooks

- Sometimes, is necessary to reuse some stateful logic between components
- Traditionally, there were two popular solutions to this problem
 - **Higher-order components**
 - **Render props**
- Custom Hooks let you do this, but without adding more components to your tree



- A custom hook is simple JS function whose **name starts with "use"** and that may call other Hooks
- Unlike a React component, a custom Hook **doesn't need** to have a specific signature
- We can decide
 - **What it takes as arguments**
 - **What should return**



Custom Hook Demo



Rules of Hooks

Rules of Hooks

- Hooks are JavaScript functions, but you need to follow two rules when using them
 - **Only Call Hooks at the Top Level**
 - **Only Call Hooks from Functional Components**



Only Call Hooks at the Top Level

- **Don't call Hooks inside loops, conditions, or nested functions**
 - 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 Functional Components

- Don't call Hooks from regular JavaScript functions. Instead, you can
 - Call Hooks from React function components
 - Call Hooks from custom Hooks
- By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code





Context

Context

- Context provides way to pass data through the component tree without passing the props manually
- Context API
 - **React.createContext**
 - **Context.Provider**
 - **useContext**



- Context is designed to **share data** that can be considered **global**
 - Current authenticated user
 - Theme
 - Preferred language
- Using context, we can **avoid passing props** through intermediate elements

- Context is **primarily** used when
 - Some data needs to be accessible by **many** components at **different** nesting levels
- Apply it **sparingly** because it makes component **reuse** more **difficult**
- Using Context only the top-most Page Component know about your data

- **React.createContext**

```
const someContext = React.createContext(defaultValue);
```

- Creates a Context object
- While rendering will read the current context value from the closest matching Provider above it in the tree
- The default value is used only when a component does not have a matching Provider above it in the tree

■ Context.Provider

```
<MyContext.Provider value={/* some value */}>
```

- Every Context object comes with a Provider React component
 - Allowing consuming components to subscribe to context changes
- Accepts a **value prop** to be passed to consuming components
- **One Provider** can be connected to **many consumers**

■ useContext

- Accepts a context object
- Return the current context value for that context
- The current context value is determined by the value prop of the nearest **Provider**
- Argument to useContext must be the context object itself

```
const ThemeContext = React.createContext(themes.light);  
...  
const theme = useContext(ThemeContext);
```



Context Demo

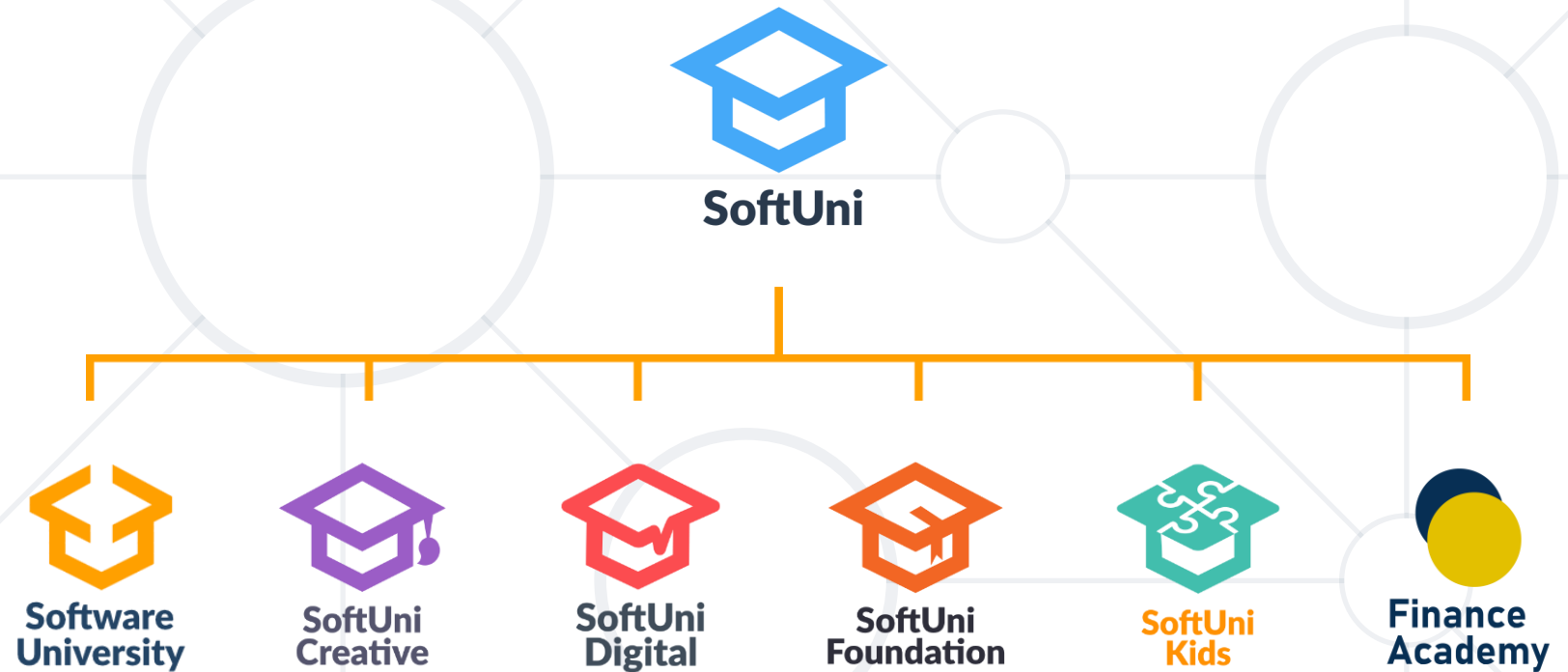
- **Hooks** is a special functions that lets you "hook into" React features
- **useState** lets you add **React state** to function components
- **useEffect** adds the ability to perform side effects from a function component
- Custom Hooks are normal JS functions, whose **names starts with "use"**
- There is **two rules** of using Hooks



- Context provides way to pass data through the component without passing the props manually
 - **Context API**
- More Hooks
 - **useContext**



Questions?



SoftUni Diamond Partners

**SUPER
HOSTING
.BG**



**Coca-Cola HBC
Bulgaria**

 **Flutter**TM
International

INDEAVR
Serving the high achievers



AMBITIONED

 **DRAFT
KINGS**



BOSCH

 **Postbank**
Решения за твоето утре

 **PHAR
VISION**

 **SmartIT**

DXC
TECHNOLOGY

createX

- 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



- 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>

