Lekcja 6 i 7 i 8 i 9 i 10

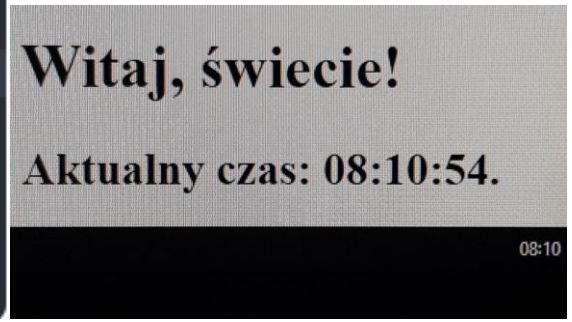
Stan i cykl życia

- Zacznijmy od wyizolowania kodu, który odpowiada za wygląd zegara:

```
function Clock(props) {
  return (
    <div>
      <h1>Witaj, świecie!</h1>
      <h2>Aktualny czas: {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```



Idealnie byłoby móc napisać tylko tyle i oczekiwać, że Clock zajmie się resztą:

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Przekształcanie funkcji w klasę

Proces przekształcania komponentu funkcyjnego (takiego jak nasz Clock) w klasę można opisać w pięciu krokach:

1. Stwórz klasę zgodną ze standardem ES6 o tej samej nazwie i odziedzicz po klasie React.Component przy pomocy słowa kluczowego extend.

2.Dodaj pustą metodę o nazwie render().

3.Przenieś ciało funkcji do ciała metody render().

4.W render() zamień wszystkie props na this.props.

5.Usuń starą deklarację funkcji.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Dodawanie lokalnego stanu do klasy

- Przenieśmy teraz date z atrybutów do stanu w trzech krokach:

1.Zamieńmy wystąpienia this.props.date na this.state.date w ciele metody render():

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

- 2.Dodaj konstruktor klasy i zainicjalizuj w nim pole this.state:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

- 3. Usuń atrybut date z elementu <Clock />:

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Timer dodamy do komponentu nieco później.
W rezultacie powinniśmy otrzymać następujący kod:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Dodawanie metod cyklu życia do klasy

- W klasie możemy zadeklarować specjalne metody, które będą uruchamiały kod w momencie montowania i odmontowywania komponentu:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
  }

  componentWillUnmount() {
  }

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Takie metody nazywamy “metodami cyklu życia”.

Metoda componentDidMount() uruchamiana jest po wyrenderowaniu komponentu do drzewa DOM. To dobre miejsce na inicjalizację timera:

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

Zatrzymaniem timera zajmie się metoda cyklu życia zwana componentWillUnmount():

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

Na koniec zaimplementujemy metodę o nazwie tick(), którą komponent Clock będzie wywoływał co sekundę.

Użyjemy w niej this.setState(), aby zaplanować aktualizację lokalnego stanu komponentu:

```jsx
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Witaj, świecie!</h1>
        <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

PropTypes (Typy propów)

- Przyjrzyjmy się, jak dziś tworzyć komponenty React wielokrotnego użytku, abyśmy mogli udostępniać nasze komponenty w aplikacjach i zespołach.

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

class Greeting extends React.Component {
  render() {
    return (
      <h1>Witaj, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

- React ma wiele typów do wyboru, eksportowane na obiekt, a nawet pozwala nam zdefiniować niestandardowy typ obiektu. Przyjrzyjmy się ogólnej liście dostępnych typów: PropTypes

```
import PropTypes from 'prop-types'

class Header extends React.Component {
  // ...
}

Header.propTypes = {
  title: PropTypes.string
}
```

Podstawowe typy

Typu	Przykład	Klasa
Ciąg	"cześć"	`React.PropTypes.string`
Numer	10, 0.1	`React.PropTypes.number`
Boolean	prawda / fałsz	`React.PropTypes.bool`
Funkcja	`const say => (msg) => console.log("Hello world")`	`React.PropTypes.func`
Symbol	Symbol("msg")	`React.PropTypes.symbol`
Obiektu	`{name: 'Ari'}`	`React.PropTypes.object`
Nic	"cokolwiek", 10, `{}`	

```
Clock.propTypes = {
  title: PropTypes.string,
  count: PropTypes.number,
  isOn: PropTypes.bool,
  onDisplay: PropTypes.func,
  symbol: PropTypes.symbol,
  user: PropTypes.object,

  name: PropTypes.node
}
```

Wymaganie dokładnie jednego potomka

Wykorzystując PropTypes.element możesz sprawdzić, czy do komponentu przekazano dokładnie jednego potomka.

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

class MyComponent extends React.Component {
  render() {
    // Musi zawierać dokładnie jeden element. W przeciwnym wypadku zostanie wyświetlone ostrzeżenie.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};
```

Domyślne wartości właściwości

Możemy zdefiniować domyślne wartości dla właściwości przez przypisanie specjalnej właściwości defaultProps:

```
class Greeting extends React.Component {
  render() {
    return (
      <h1>Witaj, {this.props.name}</h1>
    );
  }
}

// Definiuje domyślne wartości dla właściwości:
Greeting.defaultProps = {
  name: 'obcy'
};

// Renderuje "Witaj, obcy":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

Style i CSS

Jak można dodać klasy CSS do komponentów? Przekazujemy ciąg znaków używając atrybutu className:

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

Klasy CSS mogą być zależne od właściwości i stanu komponentów:

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

Czym jest CSS-w-JS?

Czy można robić animacje w w Reactie?

```
const Button = styled.button`
  font-size: 1em;
  margin: 1em;
  padding: 0.25em 1em;
  border: 2px solid palevioletred;
  border-radius: 3px;

  /* Adapt the colors based on primary prop */
  background: ${props => props.primary && 'palevioletred'};
  color: ${props => props.primary ? 'white' : 'palevioletred'};
`;

<Button primary>Github</Button>
<Button>Try it out!</Button>
```

Interakcja z użytkownikiem

Przeglądarka to aplikacja sterowana zdarzeniami. Wszystko, co użytkownik robi w przeglądarce, uruchamia zdarzenie, od klikania przycisków po nawet poruszanie myszą. W prostym języku JavaScript możemy nasłuchiwać tych zdarzeń i dołączać funkcję JavaScript do interakcji z nimi.

W React nie musimy jednak wchodzić w interakcje z pętlą zdarzeń przeglądarki w surowym JavaScript, ponieważ React umożliwia nam obsługę zdarzeń za pomocą props.

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // Poniższe wiązanie jest niezbędne do prawidłowego przekazania
`this` przy wywołaniu funkcji
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'WŁĄCZONY' : 'WYŁĄCZONY'}
      </button>
    );
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

Czym jest funkcja constructor ?

- W JavaScript constructor jest funkcją, która jest uruchamiana podczas tworzenia obiektu. Zwraca referencję do funkcji Object, która utworzyła instancję prototyp
- Używając ES6 składni klasy do utworzenia obiektu, musimy wywołać tę metodę super() przed każdą inną metodą. Wywołanie funkcji super() wywołuje funkcję klasy nadrzędnej constructor(). Nazwiemy go tymi samymi argumentami, co constructor() naszej klasy.

```
constructor(props) {
  super(props);
  this.state = {isToggleOn: true};

  // Poniższe wiązanie jest niezbędne do prawidłowego przekazania
`this` przy wywołaniu funkcji
  this.handleClick = this.handleClick.bind(this);
}
```

Przekazywanie argumentów do procedur obsługi zdarzeń

Dość często, na przykład w pętli, potrzebujemy przekazać do procedury obsługi zdarzenia dodatkowy parametr. Na przykład, jeśli zmienna id zawierałaby identyfikator wiersza w tabeli, można by rozwiązać to na dwa sposoby:

```
<button onClick={(e) => this.deleteRow(id, e)}>Usuń wiersz</button>
<button onClick={this.deleteRow.bind(this, id)}>Usuń wiersz</button>
```

Obydwie linie robią to samo, przy użyciu, odpowiednio, funkcji strzałkowej oraz Function.prototype.bind.