**React**

# Czym jest React?

01

**React** to otwarta biblioteka JavaScript stworzona przez Facebook w 2011 roku w celu usprawnienia tworzenia zaawansowanych aplikacji internetowych i mobilnych - w szczególności SPA - *Single Page Applications*.

Główny nacisk jest położony na wydajność i szybkość renderowania komponentów.

Do najważniejszych cech Reacta należą:

- Wykorzystanie **Virtual DOM** zamiast *prawdziwego* DOM w celu manipulowania widokiem,
- **Jednokierunkowy** przepływ danych w aplikacji,
- Wsparcie dla SSR (ang. *Server-Side Rendering*),
- Tworzenie re-używanych komponentów,
- Wykorzystanie **JSX** do projektowania komponentów.

**React**

# Jakie są zalety i wady Reacta?

02

Do korzyści korzystania z **Reacta** należy:

- Lepsza wydajność działania aplikacji,
- Czytelny i łatwy w utrzymaniu kod,
- Proste i spójne komponenty dzięki zastosowaniu JSX,
- Duża swoboda w doborze dodatkowych narzędzi, bibliotek oraz technik programowania,
- Szybkie i czytelne testy.

Do wad można zaliczyć:

- Trudność w zachowaniu spójnych standardów pisania kodu pomiędzy zespołami, wynikająca z dużej swobody wyboru bibliotek i technik programowania,
- Rozbudowany ekosystem Reacta utrudniający pracę mniej doświadczonym programistom - każdy problem można rozwiązać z wykorzystaniem różnych bibliotek.

**React**

# Co odróżnia React od Angular?

03

| **React** | **Angular** |
| --- | --- |
| Biblioteka zapewniająca obsługę *View* w klasycznym modelu MVC | Framework zapewniający pełne wsparcie dla MVC |
| Jednokierunkowy przepływ danych | Dwukierunkowy przepływ danych |
| Wykorzystuje Virtual DOM | Wykorzystuje *prawdziwy* DOM |
| Wsparcie dla *Server Side Rendering* | *Server Side Rendering* jedynie poprzez użycie Angular Universal |
| Tworzenie widoków za pomocą JSX | Tworzenie widoków na podstawie szablonów HTML |

**React**

# Czym jest Virtual DOM?

04

**Virtual DOM** to koncepcja programistyczna, w której *wirtualna* reprezentacja interfejsu użytkownika jest przechowywana w pamięci i synchronizowana z *prawdziwym* modelem DOM przez bibliotekę React. Proces ten nazywa się **rekoncyliacją** (ang. reconciliation).

Wykorzystanie **wirtualnego modelu DOM** tworzy przejrzysty interfejs pozwalający programistom na pominięcie lub przyspieszenie kosztownych operacji. Dodatkowo React ukrywa pod **warstwą abstrakcji** manipulację atrybutami, obsługę zdarzeń i ręczną aktualizację modelu DOM.

Zmiany dokonane na Virtual DOM są **synchronizowane** w większych *paczkach* a nie pojedynczo, co znacznie przyspiesza aktualizację widoku.

**React**

# Jaka jest różnica między DOM a Virtual DOM?

05

| DOM | Virtual DOM |
| --- | --- |
| Powolna aktualizacja modelu | Bardzo szybka aktualizacja |
| Zmiany w modelu DOM są bardzo kosztowne | Modyfikacje Virtual DOM są *tanie* a rzeczywista synchronizacja z *prawdziwym* modelem DOM następuje w fazie rekoncyliacji |
| Może aktualizować HTML bezpośrednio | Nie może aktualizować HTML bezpośrednio |
| Duże zużycie pamięci i możliwe trudne do zdiagnozowania wycieki | Ograniczone zużycie pamięci |

**React**

# Czym jest JSX?

**JSX** to przypominające XML rozszerzenie języka JavaScript, za pomocą którego można łączyć kod JavaScript z HTML w jednym pliku. Dzięki temu zarządzanie i utrzymywanie kodu jest znacznie łatwiejsze.

JSX może przypominać język oparty o szablony, jednakże daje on do dyspozycji pełnię możliwości JavaScriptu.

```
const name = 'Bob';
const element = <h1>Hello, {name}</h1>;

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

**React**

# Do czego służą *props*?

07

Koncepcyjnie, komponenty są jak funkcje w JavaScript. Przyjmują parametry (nazywane właściwościami - **props**) na wejściu i zwracają komponenty React opisujące, co powinno się pojawić na ekranie.

Mogą to być zarówno pojedyncze wartości, jak i całe obiekty.

**Props** wykorzystuje się do:

- Do przekazania danych do komponentu,
- Wywołania zmiany stanu komponentu,
- Tworzenia widoku wewnątrz metody `render()`.

```
function Welcome(props) {
    return <h1>Hello, {props.name}</h1>;
}
```

**React**

# Jaka jest różnica pomiędzy *state* i *props*?

08

Zarówno **state**, jak i **props** to obiekty JavaScript przechowujące informacje, na podstawie których następuje renderowanie.

**Props** są przekazywane do komponentu z zewnątrz i stanowią jego *konfigurację*. W podobny sposób parametry są przekazywane do wywoływanej funkcji w JavaScript.

Innymi słowy, **props** opisują w jaki sposób komponenty komunikują się między sobą. Komponent nie może modyfikować obiektu **props**.

**State** nie jest przekazywany do komponentu z zewnątrz, ale jest utrzymywany i zarządzany przez komponent w sposób podobny do tego, jak zmienne lokalne są obsługiwane wewnątrz ciała funkcji. Komponent podczas tworzenia otrzymuje *stan domyślny*, który następnie może być wielokrotnie modyfikowany w trakcie życia komponentu.

**React**

# Na czym polega *prop drilling*?

09

**Prop drilling** występuje, gdy chcemy przekazać dane z komponentu rodzica do komponentu występującego głębiej w drzewie komponentów.

Przekazując informacje przez kolejne poziomy komponentów uzależniamy te komponenty od danych, których nie powinny być świadome. Utrudnia to ponowne wykorzystanie kodu.

Aby uniknąć problemów związanych z **prop drilling** można wykorzystać **React Context**.

Komponent `Provider` jest wtedy odpowiedzialny za dostarczenie danych, które mogą być odczytane przez zagnieżdżone komponenty poprzez hook `useContext()` albo komponent `Consumer`.

Drugim rozwiązaniem jest wykorzystanie mechanizmów zarządzania stanem, np. **Redux**.

**React**

# W jaki sposób wymusić typ *props*?

10

Aby sprawdzić typ właściwości przekazanych do komponentu należy skorzystać z biblioteki **propTypes**, która zawiera walidatory do sprawdzania typu i poprawności danych wejściowych.

Kiedy wartość przekazanych **props** będzie nieprawidłowego typu, zostanie wyświetlone ostrzeżenie w konsoli przeglądarki.

Ze względów wydajnościowych, **propTypes** są sprawdzane tylko w trybie deweloperskim.

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

class Greeting extends React.Component { … }

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

 **React**

# Jak sprawdzić czy w props przekazano jeden *child*?

Aby sprawdzić czy do komponentu przekazano tylko jeden ***child* komponent** (jednego potomka) należy skorzystać z walidatora `propTypes.element`, przykładowo:

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

class Sidebar extends React.Component {
    render() {
        const children = this.props.children;
        return (
            <div>
                {children}
            </div>
        );
    }
}

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

**React**

# Do czego służy *React Context*?

12

**React Context** służy do przekazywania danych wewnątrz drzewa komponentów bez konieczności przekazywania ich przez właściwości każdego komponentu po drodze. Unikamy w ten sposób tzw. *prop drilling*.

Konteksty zaprojektowano do **współdzielenia danych**, które można uznać za *globalne* dla całego **drzewa komponentów**,

```
const ThemeContext = React.createContext('dracula');

// render parent komponentu
<ThemeContext.Provider value="solar">
    <Toolbar />
</ThemeContext.Provider>

class Toolbar extends React.Component {
    static contextType = ThemeContext;
    render() {
        return <Button theme={this.context} />;
    }
}
```

**React**

# Czym jest ReactDOM?

13

**ReactDOM** jest elementem łączącym React z modelem DOM. Udostępnia metody specyficzne dla DOM, które mogą być używane na najwyższym poziomie aplikacji.

Podstawowa biblioteka `react` udostępnia funkcjonalności, które są wspólne niezależnie tego, czy tworzymy aplikację webową czy mobilną.

Przykładowo są to: funkcja `React.createElement()`, klasy `React.Component`, `React.Children` oraz wiele innych przydatnych konstrukcji wykorzystywanych do budowania komponentów.

Biblioteka `react-dom` zawiera z kolei `ReactDOM.render()` oraz kod potrzebny do obsługi Server-side Rendering.

**React**

# Do czego służą *hooki*?

14

**Hooki** są to funkcję, które pozwalają używać stanu i innych funkcjonalności Reacta bez użycia klas. Przenoszą mocne strony komponentów opartych na klasach (np. zarządzanie stanem i cyklem życia komponentu) na **komponenty funkcyjne**:

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

export function CounterHook() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times!</p>
      <button onClick={() => setCount(count + 1)}>
        Click me!
      </button>
    </div>
  );
}
```

 **React**

# Jakie są korzyści korzystania z *hooków*?

15

**Hooki** mają wiele zalet związanych z organizacją kodu oraz wykorzystaniem komponentów funkcyjnych:

- Oddzielają logikę związaną ze stanem od komponentu,
- Ułatwiają ponowne wykorzystanie kodu bez konieczności zmiany hierarchii komponentów,
- Przyspieszają testowanie, ponieważ nie są wymagane dodatkowe zależności do komponentów,
- Pozwalają podzielić komponent na mniejsze funkcje, bazując na ich odpowiedzialności (np. tworzenie subskrypcji czy pobieranie danych), zamiast wymuszać sztuczny podział związany z metodami cyklu życia,
- Pozwalają na korzystanie z większej liczby funkcjonalności Reacta bez użycia klas,
- Są kompatybilne wstecz i działają równolegle z istniejącym kodem, co ułatwia ich wdrażanie

**React**

# Jak pominąć wywołanie *useEffect*?

16

Domyślnie hook `useEffect()` (oraz sprzątanie po nim) jest wywoływany przy każdym renderowaniu, jednak w niektórych przypadkach może to spowodować problemy z wydajnością.

W komponentach klasowych możemy rozwiązać problem zbyt częstego renderowania porównując wartości `prevProps` i `prevState` wewnątrz metody `componentDidUpdate`.

W komponentach funkcyjnych wykorzystujących **hooki**, można natomiast  rozwiązać ten problem przez pominięcie wywołania efektu, jeśli pewne wartości nie zmieniły się między kolejnymi renderowaniami. Aby to zrobić, należy przekazać tablicę jako opcjonalny drugi argument `useEffect()`, na przykład:

```
useEffect(() => {
    document.title = `Clicked ${count} times!`
}, [count]);
```

**React**

# Jak wywołać *hook* *useEffect* tylko raz?

17

Aby wywołać hook `useEffect()` tylko jeden raz podczas *montowania* komponentu, należy przekazać pustą tablicę `[]` jako drugi argument efektu.

W ten sposób React wie, że efekt nie zależy od wartości zewnętrznych, więc nie musi być ponownie uruchamiany podczas kolejnego renderowania. Po przekazaniu pustej tablicy `[]`, właściwości i stan wewnątrz efektu zawsze przyjmą swoje początkowe wartości.

```
useEffect(async () => {
  const result = await axios(
    'https://fiszkijs.pl/api/v1/questions',
  );

  setQuestions(result.data);
}, []);
```

**React**

# Jak memoizować obliczenia za pomoc *hooków*?

**Memoizację** można osiągnąć zapamiętując wynik operacji pomiędzy kolejnymi renderowaniami za pomocą hooka `useMemo()`.

Można w ten sposób przechowywać wyniki *ciężkich* obliczeń lub kosztownego renderowania *child* komponentów:

```
const counter = useMemo(
  () => <Counter count={total} />, [total]
);
```

Powyższy kod sprawia, że komponent `Counter` będzie ponownie renderowany tylko wtedy, gdy zmieni się wartość `total`.

Jeśli jednak zależność `[total]` nie zmieniła się od ostatniego razu, `useMemo()` pominie kolejne wywołanie funkcji i zamiast tego zwróci poprzedni wynik.

 **React**

# Jak za pomocą *hooków* tworzyć "ciężkie" obiekty?

"Ciężkie" obiekty można tworzyć w **leniwy sposób** (*lazy loading*), lub wykorzystać do tego **memoizację.**

**Memoizacja** realizowana za pomocą `useMemo()` pozwala na przechowywanie wyników kosztownych obliczeń, pod warunkiem, że ich zależności są takie same. Jednak `useMemo()` **nie gwarantuje**, że obliczenia te zostaną wykonane tylko raz.

Tworzenie obiektów w sposób leniwy można zrealizować za pomocą `useState()` oraz **funkcji inicjalizującej**, co gwarantuje, że React wywoła ją tylko przy pierwszym renderowaniu:

```
function DataGrid(props) {
    const [data, setData] = useState(
        () => loadGridData(props.source)
    );
    // ...
}
```

**React**

# Do czego służy funkcja *render*?

20

Funkcja `ReactDOM.render()` renderuje element do drzewa DOM i umieszcza go w kontenerze podanym jako argument. Zwraca referencję do komponentu (lub null dla komponentów bezstanowych).

W przypadku renderowania więcej niż jednego elementu, muszą być one objęte wspólnym tagiem, np. `<form>` lub `<div>`.

Jeśli element był wcześniej renderowany, zostanie automatycznie **zaktualizowany** przez Reacta, który odpowiednio zmodyfikuje DOM, aby odzwierciedlić najnowszą wersję komponentu.

```
ReactDOM.render(
    <React.StrictMode>
        <App />
    </React.StrictMode>,
    document.getElementById('root')
);
```

**React**

# Co różni komponenty *funkcyjne* i *klasowe*?

21

**Komponenty klasowe** pozwalają na korzystanie z lokalnego stanu komponentu oraz metod cyklu życia komponentu. Dziedziczą je z klasy `React.Component`, którą muszą rozszerzać.

**Komponenty funkcyjne** nie posiadają swojego wewnętrznego stanu oraz nie można w nich korzystać z metod cyklu życia komponentu.

Są to zwykłe funkcje JavaScript, które otrzymują obiekt `props` jako parametr wejściowy i zwracają nowy element. W związku z tym nie mogą przechowywać swojego wewnętrznego stanu.

Aby korzystać z obiektu stanu w komponentach funkcyjnych można a) skorzystać z hooków, b) zmienić je na komponenty klasowe, lub c) przekazać stan do obiektu rodzica i stamtąd tworzyć komponenty, przekazując stan jako `props`.

**React**

# Kontrolowane vs niekontrolowane komponenty?

**Komponentem kontrolowanym** jest komponent reprezentujący pole formularza (`<select>`, `<textarea>`, `<input>`, itp.), którego wartością zarządza React.

Gdy użytkownik wprowadzi do niego dane, wywoływana jest obsługa zdarzenia, podczas której następuje decyzja czy wartość jest poprawna i można komponent ponownie renderować.

Z kolej **komponent niekontrolowany** działa tak, jak wszystkie pola formularza istniejące poza Reactem. Gdy użytkownik wprowadzi do niego dane, zmiana wartości następuje automatycznie, bez konieczności obsługiwania tego w kodzie Reacta.

Innymi słowy komponent niekontrolowany traktuje DOM jako *source of truth* wartości pól formularza, a komponent kontrolowany korzysta ze swojego wewnętrznego stanu.

**React**

# Czym są komponenty wyższego rzędu?

23

**Komponent wyższego rzędu (HOC)** to funkcja, która przyjmuje jako argument inny komponent i zwraca nowy komponent.

Tak jak zwykły komponent przekształca **props** na element na stronie, tak komponent wyższego rzędu **przekształca komponent w inny komponent**.

Co ważne **nie modyfikuje** przekazanego mu komponentu ani nie stosuje dziedziczenia w celu skopiowania jego zachowania. Zamiast tego wkomponowuje przekazany komponent poprzez jego opakowanie w kontener.

Komponent wyższego rzędu jest zatem **czystą funkcją** (ang. *pure function*), nie mającą żadnych efektów ubocznych. Ułatwia to re-użycie kodu, logiki i abstrakcji występujących w kodzie i jest dobrym sposobem na **wydzielenie wspólnych odpowiedzialności** do jednego spójnego komponentu.

**React**

# Do czego służy React.memo()?

24

`React.memo()` jest funkcją tworzącą komponenty wyższego rzędu, które pozwalają na optymalizację wydajności aplikacji poprzez zapobiegają zbyt częstemu renderowaniu komponentów funkcyjnych.

Komponent można opakować w `React.memo()` w celu poprawy wydajności, jeśli przy takich samych `props` renderuje ten sam widok i tą samą strukturę. Jeśli komponent używa hooków `useState()` lub `useContext()`, nadal będzie aktualizował się przy zmianie stanu komponentu lub kontekstu.

Domyślnie, komponent wykona jedynie płytkie (ang. *shallow*) porównanie obiektów przekazanych we właściwościach.

```
const Tweet = React.memo(function Tweet(props) {
    // ...
});
```

**React**

# Czym jest PureComponent?

25

`React.PureComponent` jest to klasa zbliżona do `React.Component`, z tą różnicą, że metoda cyklu życia `shouldComponentUpdate()` jest w niej obsługiwana automatycznie i wykonuje porównanie obiektów `props` i `state` z użyciem płytkiego (ang. *shallow*) porównania.

Dzięki temu można zapobiec zbyt częstemu renderowaniu komponentów. W przypadku `React.Component` takie porównanie trzeba napisać samemu. Należy jednak pamiętać, że porównywanie obiektów może również okazać się kosztowne, więc korzystanie z `PureComponent` powinno być przemyślane.

Jeśli metoda `render()` komponentu wyświetla ten sam rezultat przy tych samych `props` i `state`, można przekształcić go na `React.PureComponent`, aby poprawić wydajność.

**React**

# Czym są granice błędów?

**Granice błędów** (ang. *error boundary*) to komponenty, które przechwytują błędy występujące wewnątrz drzewa komponentów, a następnie logują je i wyświetlają zastępczy interfejs UI, zamiast pokazywać ten niepoprawnie działający.

Aby komponent klasowy stał się granicą błędu, musi definiować jedną, lub obie z poniższych metod cyklu życia:

- `static getDerivedStateFromError()` do wyrenderowania zastępczego UI po rzuceniu błędu
- `componentDidCatch()` do zalogowania informacji o błędzie.

Granice błędów nie obsługują błędów w: procedurach obsługi zdarzeń (ang. *event handlers*), asynchronicznym kodzie, komponentach renderowanych po stronie serwera oraz błędów rzuconych w ramach działania samego *error boundary*.

**React**

# Jakie znasz metody cyklu życia komponentu?

- `getDerivedStateFromProps`: wywoływana tuż przed każdym wywołaniem `render()`; Powinna zwrócić obiekt, aby zaktualizować stan, lub zwrócić null, aby nie aktualizować.
- `componentDidMount`: wywoływana po pierwszym renderowaniu; jest miejscem gdzie można umieścić obsługę żądań HTTP, *event listenery* lub inicjalizację wymagającą DOM
- `shouldComponentUpdate`: określa, czy komponent zostanie zaktualizowany (domyślnie tak).
- `getSnapshotBeforeUpdate`: wywoływana tuż przed zapisaniem zmian w DOM.
- `componentDidUpdate`: obsługuje zmiany w DOM w odpowiedzi na zmiany w props lub state.
- `componentWillUnmount`: wywoływana tuż przed usunięciem komponentu; jest miejscem, gdzie można anulować żądania HTTP oraz usunąć *event listenery*.

**React**

# Jak wykonać akcję tylko raz - podczas renderowania?

Aby wykonać akcję **tylko raz** podczas pierwszego renderowania, można skorzystać:

w przypadku **komponentów klasowych** - z metody cyklu życia komponentu `componentDidMount()`:

```
componentDidMount() {
    trackPageView('Homepage');
}
```

w przypadku **komponentów funkcyjnych** - z hooka `useEffect` przekazując dodatkowo pustą tablicę `[]` jako drugi parametr:

```
useEffect(() => {
    trackPageView('Homepage');
}, []);
```

**React**

# Metoda setState() jest synchroniczna czy asynchroniczna?

29

Metoda `setState()` działa asynchronicznie a jej wywołania są grupowane ze względów wydajnościowych.

Dzięki temu, jeśli zarówno *parent component*, jak i *child component* wywołają `setState()` podczas zdarzenia, komponent dziecko nie zostanie wyrenderowany dwukrotnie. Zamiast tego React uruchomi wszystkie te aktualizacje stanu na koniec obsługi zdarzenia.

React celowo *czeka*, aż wszystkie komponenty wywołają `setState()` w swoich procedurach obsługi zdarzeń, zanim zacznie ponownie renderować drzewo komponentów.

Dzięki temu **unikamy niepotrzebnego i kosztownego** wielokrotnego renderowania, co znacząco wpływa to na wydajność.

**React**

# Dlaczego nie należy wprost zmieniać wartości *this.state*?

Gdyby aktualizować bezpośrednio obiekt `this.state` wewnątrz komponentu, **React** nie miałby możliwości rozpoznać kiedy należy taki komponent ponownie renderować.

Korzystanie z `setState()` zapewnia nam taką możliwość. Sama operacja aktualizacji stanu jest **asynchroniczna**, dzięki czemu React może ją optymalizować.

Jedynym miejscem, w którym można przypisywać `this.state` bezpośrednio, jest **konstruktor** komponentu.

Aby zaktualizować stan w oparciu o poprzednie jego wartości, można dodatkowo przekazać do `setState()` funkcję, która przyjmuje **state** i **props** jako parametry:

```
this.setState((state, props) => ({
    count: state.count + props.increment
}));
```

**React**

# Jaka jest rola funkcji przekazywanej do *setState()*?

31

Funkcja przekazywana jako parametr do `setState()` zamiast obiektu pozwala na aktualizację stanu komponentu w oparciu o **poprzedni stan** oraz wartość **props**.

Sama aktualizacja stanu jest **asynchroniczna,** co pozwala Reactowi grupować tego typu operacje i je optymalizować.

Stan może nie zostać zaktualizowany natychmiast po wywołaniu `setState()`, co ma znaczenie, jeśli wywołujemy tą metodę wielokrotnie, np:

```
this.setState({ count: this.state.count + 1 })
this.setState({ count: this.state.count + 1 })
// po wykonaniu nadal state.count === 1
// aby pozbyć się błędu, należy skorzystać z:
this.setState((state, props) => ({
    count: state.count + props.increment
}));
```

**React**

# Jaka jest różnica między zdarzeniami w React i w HTML?

32

W React zdarzenia są podobne do natywnych zdarzeń przeglądarki, takich jak *mouse hover*, *mouse click*, *key press*. Istnieje jednak kilka różnic:

- Zdarzenia Reacta zapisywane są jako `camelCase`, a nie jako `lowercase`.
- W przeciwieństwie do zdarzeń HTML nie mogą zwracać **false** w celu przerwania obsługi, tylko muszą wywoływać `preventDefault()`.
- W JSX procedura obsługi zdarzenia przekazywana jest jako funkcja, a nie łańcuch znaków, jak w przypadku zdarzeń HTML.
- Wywołanie funkcji obsługującej zdarzenie w React nie musi się kończyć nawiasami `()`.

```
<button onClick={handleClick}>Click me!</button>
```

**React**

# Czym jest *SyntheticEvent*?

33

**SyntheticEvent** (*zdarzenie syntetyczne*) jest to obiekt opakowujący zdarzenie, będący jednocześnie częścią systemu obsługi zdarzeń Reacta. Zapewnia jednolity interfejs obsługi zdarzeń niezależnie od stosowanej przeglądarki.

Zdarzenia syntetyczne posiadają taki sam interfejs jak natywne zdarzenia, wliczając w to metody `stopPropagation()` oraz `preventDefault()` i gwarantują identyczne działanie na wszystkich przeglądarkach.

Aby skorzystać z opakowanego, natywnego zdarzenia, należy odwołać się do niego poprzez właściwość `nativeEvent`.

Na przykład, w zdarzeniu `onMouseLeave` wartość `event.nativeEvent` będzie wskazywało na natywne zdarzenie `mouseout` z API przeglądarki.

**React**

# Jakie znaczenie mają klucze w React?

**Klucze** pomagają Reactowi zidentyfikować elementy kolekcji, które uległy zmianie, zostały dodane lub usunięte. Są wykorzystywane do rozróżniania elementów wirtualnego modelu DOM. Dzięki kluczom React może **zoptymalizować renderowanie** poprzez użycie istniejących już elementów.

Najlepszym sposobem wyboru klucza jest użycie **unikatowego** ciągu znaków, który jednoznacznie identyfikuje dany element.

Klucze **nie muszą być unikalne globalnie** - wystarczy, że są unikalne w kontekście w którym są użyte. Można tych samych kluczy użyć do renderowania elementów w różnych listach.

```
const todoItems = Todos.map((todo) =>
    <li key={todo.id}>
        {todo.text}
    </li>
);
```

**React**

# Jakie znaczenie mają *refs* w React?

35

Referencje `ref` to specjalny atrybut wspierany przez React, który zapewnia dostęp do API elementów modelu DOM lub elementów stworzonych przez wywołanie `render()`.

Może on być funkcją, lub obiektem utworzonym przy użyciu `React.createRef()`. Zazwyczaj jest tworzony w konstruktorze i od razu przypisywany do instancji komponentu. Podczas odczytu węzeł DOM jest dostępny przez atrybut `current`:

```
// utworzenie ref i przypisanie w konstruktorze
this.todosRef = React.createRef();

// użycie ref w komponencie
<div ref={this.todosRef} />

// dostęp do elementu przez atrybut current
const todos = this.todosRef.current;
```

**React**

# Do czego służy React Router?

36

**React Router** to biblioteka Reacta, która pozwala na dodawanie nowych ekranów oraz nawigowanie między nimi. Posiada wiele zalet ułatwiających pisanie kodu, przykładowo:

- Posiada proste API, za pomocą którego użytkownik definiuje i konfiguruje router podając ścieżki do poszczególnych ekranów.
- Opakowuje dostęp do obiektu `window.history`, będącego częścią *History API* z HTML5.
- Umożliwia konfigurację pod postacią zwykłego komponentu Reacta `<BrowserRouter>`, w którym definiujemy ścieżki - `<Route>`.
- Jest podzielona na trzy części: *Web*, *Native* oraz część wspólną *Core*, co ułatwia pisanie kodu na różne środowiska.

**React**

# Jakie znasz typy komponentów <Router>?

37

**React Router** posiada kilka implementacji Routera, z których można korzystać w zależności od potrzeb i środowiska:

`<Router>` - Podstawowa, niskopoziomowa implementacja bazowa dla wszystkich *konkretnych* implementacji routerów.

`<BrowserRouter>` - Używa *History API* z HTML5 do nawigacji i utrzymywania adresu URL spójnego ze stanem aplikacji.

`<HashRouter>` - Do nawigacji używa tylko części *hash* z adresu URL - dostępnego przez `window.location.hash` - przykładowo `#/users/guest`.

`<MemoryRouter>` - Przechowuje informacje o URL i jego zmianach w pamięci; nie modyfikuje przy tym adresu strony w pasku adresu przeglądarki, co jest szczególnie przydatne przy testowaniu w React Native.

**React**

# Co różni <BrowserRouter> i <HashRouter>?

38

Zarówno `<BrowserRouter>` oraz `<HashRouter>` to dwie najważniejsze implementacje komponentu routera w aplikacjach webowych.

Różnią się głównie sposobem w jaki **przechowują informacje** o URL i **komunikują się z serwerem**.

`<BrowserRouter>` korzysta z pełnej ścieżki URL, co jest najbardziej czytelnym rozwiązaniem, jednak wymaga konfiguracji po stronie serwera, aby ten zwracał tą samą stronę HTML niezależnie od ścieżki przekazanej w żądaniu HTTP.

`<HashRouter>` przechowuje informacje o lokalizacji strony w części *hash* adresu URL, przykładowo `#/users/admin`. Zgodnie ze specyfikacją *Location API*, zmiana hash nie powoduje przekierowania, więc komunikacja z serwerem nie jest wymagana.

**React**

# Jak wywołać routing programistycznie?

39

Aby programistycznie wywołać routing i przekierować użytkownika na inny ekran można skorzystać z dwóch rozwiązań:

- Wykorzystać hook `useHistory()` w komponentach funkcyjnych, który daje dostęp do obiektu *history* i jego metod `pushState()` oraz `replaceState()`:

```
let history = useHistory();
```

- Skorzystać z funkcji tworzącej komponenty wyższego rzędu `withRouter()`, która udostępnia informacje o obiekcie *history* komponentowi, który opakowuje:

```
const Goto = withRouter(({ history }) => (
  <button type='button'
    onClick={() => {history.push('/abc')}}>
      Click Me!
  </button>
))
```

**React**

# Jak obsłużyć brak strony (status 404) w React Router?

40

Należy w tym celu skorzystać z komponentu `<Switch>`. Jego działanie polega na renderowaniu pierwszego elementu `<Route>`, którego `path` pokrywa się z wyszukiwanym adresem.

Ostatni `<Route>` może być wykorzystany do wyłapywania wszystkich niezdefiniowanych przypadków. W tym celu można pominąć `path`, lub zdefiniować `path` pasujący do wszystkich przypadków, na przykład `path="*"`.

```
<Switch>
    <Route exact path="/">
        <Home />
    </Route>
    <Route path="*">
        <NotFound />
    </Route>
</Switch>
```

**React**

# Jak obsłużyć przekierowanie w React Router?

41

**Przekierowanie** w React Router można zrealizować za pomocą komponentu `<Redirect>`. Przekierowanie na nowy adres nastąpi w momencie renderowania.

Podobnie, jak w przypadku przekierowania realizowanego po stronie serwera - czyli **HTTP 3xx**, nastąpi zmiana adresu w pasku adresu przeglądarki i zostanie dodany nowy wpis do historii przeglądania.

```
<Route exact path="/">
    {
        loggedIn
            ? <Redirect to="/dashboard" />
            : <PublicHomePage />
    }
</Route>
```

**React**

# Jak podzielić kod na podstawie ścieżki URL?

42

Podział kodu na podstawie URL można zrealizować za pomocą funkcji `React.lazy`, która pozwala renderować dynamicznie importowane komponenty tak samo jak wszystkie inne.

"*Leniwy*" komponent powinien zostać wyrenderowany wewnątrz elementu `<Suspense>`, który pozwala na wyświetlenie komunikatu na czas ładowania, np. informacji o postępie:

```
const UserList = React.lazy(
  () => import('./UserList')
);
function DashboardComponent() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <UserList />
    </Suspense>
  );
}
```

**React**

# Czym jest Redux?

43

**Redux** to otwarta biblioteka do zarządzania stanem aplikacji. Opiera się na następujących założeniach:

- ***Single source of truth*** - stan całej aplikacji jest przechowywany jako drzewo obiektów w jednym, centralnym miejscu, którym jest **store**.
- Obiekt stanu jest **tylko do odczytu** - jedynym sposobem na jego modyfikowanie jest wykorzystanie akcji. Akcje to obiekty opisujące zmianę stanu. Dzięki temu mamy pewność, że stan nie zostanie zmieniony w niekontrolowany sposób.
- Zmiany stanu następują przez reduktory (ang. *reducers*), które określają jak zmienia się stan pod wpływem akcji. Są to funkcje nie posiadające efektów ubocznych, które przyjmują jako parametr bieżący stan oraz akcję i zwracają nowy stan aplikacji.

**React**

# Jakie znasz komponenty Redux?

44

**Redux** składa się z następujących komponentów:

- Akcje (ang. *actions*) - obiekty reprezentujące **CO** zmieniło się w stanie aplikacji. Posiadają informację o tym co się zmienia oraz dane reprezentujące zmianę. Wysyłane są przez wywołanie metody `store.dispatch()`.
- Reduktory (ang. *reducers*) - funkcje opisujące **JAK** dane z akcji modyfikują stan aplikacji przechowywany wewnątrz store.
- Store - obiekt, który przechowuje stan aplikacji w postaci drzewa i pilnuje zmian stanu. Umożliwia również odczyt stanu przez metodę `getState()` a także wysłanie akcji za pomocą metody `dispatch()`. Dodatkowo rejestruje listenery za pomocą metody `subscribe()` oraz umożliwia ich wyrejestrowanie.

**React**

# Jakie są korzyści z zastosowania Redux?

- Czytelny, zrozumiały i **łatwy w utrzymaniu kod**; aktualizacja stanu odbywa się zawsze w ten sam sposób, za pomocą akcji i reduktorów.
- Jasna **struktura** aplikacji i podział kodu ułatwiające pracę w dużych zespołach.
- Łatwa integracja z **server-side rendering**.
- Dostęp do narzędzi programistycznych ułatwiających **śledzenie zmian stanu** aplikacji w jednym, centralnie zarządzanym miejscu.
- Możliwość **odtworzenia stanu aplikacji** z dowolnego momentu w przeszłości dzięki zastosowaniu funkcji bez efektów ubocznych.
- Łatwe **testowanie** działania akcji, *store* i reduktorów, które są zwykłymi funkcjami, dobrze izolowanymi od reszty aplikacji i nie posiadającymi efektów ubocznych.
- Duża i aktywna **społeczność**.

**React**

# Do czego służy Redux Thunk?

46

**Redux Thunk** dostarcza *middleware* pozwalający na tworzenie funkcji, których zadaniem jest wykonanie **asynchronicznej operacji** a następnie odczytanie stanu i wysłanie akcji.

Realizuje się to poprzez tzw. *action creators*. Dzięki nim Thunk opóźnia wysłanie akcji do momentu kiedy zostanie zrealizowana operacja asynchroniczna, np. żądanie do REST API.

Wewnętrzna funkcja - *thunk* - przyjmuje jako parametry metody `dispatch()` oraz `getState()`, przykładowo:

```
function updateUserRole(role) {
  return dispatch => fetchUsers().then(
    users => dispatch(changeRoleTo(role, users)),
    error => dispatch(apologize('Failed', error))
  );
}
store.dispatch(updateUserRole('support'));
```

React

# Do czego służy Redux Saga?

47

**Redux Saga** to (podobnie do `redux-thunk`) *middleware* do obsługi efektów ubocznych. Różnica polega jednak na tym, że jest zbudowana wokół generatorów z ES6.

Dzięki temu kod nie musi wykorzystywać callbacków a wykonanie operacji asynchronicznej jest równie proste jak realizacja akcji synchronicznej. Poprawia przy tym czytelność kodu, obsługę wyjątków oraz ułatwia testowanie.

```
// Saga fetchUser czeka na wywołanie akcji USER_REQUESTED
// np. dispatch({ type: 'USER_REQUESTED', payload: { userId } })
function* fetchUser(action) {
  const user = yield call(Api.fetchUser, action.payload.userId);
  yield put({ type: "USER_SUCCEEDED", user: user });
}

// Wywołuje fetchUser dla każdej akcji tego typu
function* mySaga() {
    yield takeEvery("USER_REQUESTED", fetchUser);
}
```

React

# Presentational vs Container Components?

48

| **Presentational Component** (a.k.a *Dumb Component*) | **Container Component** (a.k.a *Smart Component*) |
|---|---|
| Odpowiada za to jak dane wyglądają na ekranie | Odpowiada za to jak działa logika wyświetlania |
| Korzysta z danych i callbacków pochodzących z props | Odczytuje dane z Redux store i tworzy akcje |
| Nie ma zależności do reszty aplikacji | Posiada wiedzę i korzysta z Redux |
| Nie posiada własnego stanu (lub posiada tylko stan UI) | Posiada stan wewnętrzny |
| Często implementowane jako komponenty funkcyjne | Często generowane np. przez funkcję `connect()` |
| Przykład: *UserInfo, Sidebar* | Przykład: *FilteredStoryList* |

**React**

# Do czego służy komponent StrictMode?

Komponent `<StrictMode />` jest *narzędziem* pozwalającym na znalezienie w trybie deweloperskim potencjalnych problemów w aplikacji.

Nie renderuje żadnego widocznego UI, ale aktywuje dodatkowe ostrzeżenia wyświetlające informacje o problemach w konsoli, dotyczące np:

- Użycia niebezpiecznych metod cyklu życia komponentu.
- Użycia przestarzałego API, np. tekstowych *refs*.
- Wykrycia nieoczekiwanych efektów ubocznych
- Wykorzystania przestarzałego API kontekstów

```
<React.StrictMode>
  <div>
    <Sidebar />
  </div>
</React.StrictMode>
```

**React**

# Co zrobisz jeśli aplikacja renderuje się zbyt wolno?

Należy rozpocząć od sprawdzenia **Profilera** dostępnego w ramach React Dev Tools. Na podstawie danych można znaleźć komponenty, które renderują się **zbyt długo**, lub **zbyt często**.

Jednym z najczęstszych problemów jest ponowne renderowanie komponentu, gdy nie jest to wymagane. React dostarcza *narzędzia* pozwalające rozwiązać ten problem:

`React.memo()`, które zapobiega renderowaniu komponentów funkcyjnych, oraz

`React.PureComponent`, który zapobiega renderowaniu komponentów klasowych.

Oba te rozwiązania bazują na płytkim (ang. *shallow*) porównywaniu obiektów state i prop, co również może okazać się kosztowne. Należy z nich korzystać w przemyślany sposób.