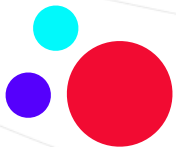


TypeScript

Wykorzystanie TypeScript w React

infoShare Academy



HELLO

Jakub Wojtach

Senior **full stack** developer





Zacznijmy ten dzień z przytupem!

Chopin

Beethoven

Gitara

Bębny

Xbox

Playstation

Chmura

Dysk twardy



Agenda

- **Podstawy** teorii oraz konfiguracja TypeScript w projekcie z React
- **Wykorzystanie** React wraz z TypeScript
- **Podsumowanie** zdobytej wiedzy o TypeScript i sposoby na migracje z JavaScriptu



Współpraca

- Zadajemy pytania w dowolnym momencie – kanał **merytoryka**
- Krótkie przerwy (**5 min**) co godzinę
- Długa przerwa (**20 min**) po ostatnim bloku





Typowanie props

- Typowanie propsów w React polega na określeniu typów danych i nazw dla właściwości, które komponent otrzymuje jako argumenty.
- Składnia z destrukuryzacją propsów umożliwia uzyskanie dostępu do wybranych właściwości bez potrzeby wielokrotnego odwoływania się do props.
- Takie typowanie pozwala na lepszą kontrolę nad aplikacją, pomaga uniknąć błędów i ułatwia późniejszą modyfikację kodu.
- W efekcie zwiększa to jakość i niezawodność aplikacji React.



Typowanie props

```
enum UserRole {  
  CEO = "ceo",  
  CTO = "cto",  
  SUBORDINATE = "inferior-person",  
}
```

```
type UserProfileProps = {  
  firstName: string;  
  role: UserRole;  
}
```

```
function UserProfile({ firstName, role }: UserProfileProps) {  
  if (role === UserRole.CTO) {  
    return <div>Hey Pat, you're AWESOME!!</div>  
  }  
  return <div>Hi {firstName}, you suck!</div>  
}
```

```
const UserProfile = ({ firstName, role }: UserProfileProps) => {  
  if (role === UserRole.CTO) {  
    return <div>Hey Pat, you're AWESOME!!</div>;  
  }  
  return <div>Hi {firstName}, you suck!</div>;  
};
```



Typowanie props – zadanie

```
// Challenge: Otypuj komponent by mógł być wyświetlony jak poniżej
//
// <Product
//   name="Shampoo"
//   price={2.99}
//   images={["image-1.png", "image-2.png"]}
// />
```

```
export function Product({ name, price, images }) {
  return (
    <div>
      <div>
        {name} ${price}
      </div>
      {images.map((src) => (
        <img src={src} />
      ))}
    </div>
  );
}
```



Typowanie stanu

- Typowanie stanu w React polega na określeniu typów danych i nazw dla zmiennych, które reprezentują dynamiczne dane wewnętrzne komponentu.
- Typowanie stanu umożliwia kompilatorowi JavaScript wykrywanie potencjalnych błędów w czasie kompilacji, zanim zostaną one zauważone podczas działania aplikacji.
- W React, stan jest zarządzany przez funkcję **useState**, która jest częścią **react hooks**.
- Typowanie stanu jest kluczowe dla poprawnego funkcjonowania aplikacji React.


```
// TypeScript doesn't know what type the array elements should have  
const [names, setNames] = useState([]);
```

```
// The initial value is undefined so TS doesn't know its actual type  
const [user, setUser] = useState();
```

```
// Same story when we use null as initial value  
const user = useState(null);
```



```
const [names, setNames] = useState<never>();  
setNames(["Pat", "Lisa"]);
```

Type 'string' is not assignable to type 'never'. ts(2322)

[View Problem \(⌘F8\)](#) No quick fixes available

```
// the type of names is string[]  
const [names, setNames] = useState<string[]>([]);  
setNames(["Pat", "Lisa"]);
```

```
// the type of user is User | undefined (we can either set a user or undefined)  
const [user, setUser] = useState<User>();  
setUser({ firstName: "Pat", age: 23, role: UserRole.CTO });  
setUser(undefined);
```

```
// the type of user is User | null (we can either set a user or null)  
const [user, setUser] = useState<User | null>(null);  
setUser({ firstName: "Pat", age: 23, role: UserRole.CTO });  
setUser(null);
```

```
// Challenge: Fix the addImage handler
```

```
import { useState } from "react";
```

```
type ProductProps = {  
  name: string;  
  price: number;  
};
```

```
export function Product({ name, price }: ProductProps) {  
  const [images, setImages] = useState([]);
```

```
    const addImage = () => {  
      setImages(images.concat(`image-${images.length + 1}.png`));  
    };
```

```
    return (  
      <div>  
        <div>  
          {name} ${price}  
        </div>  
        <button onClick={addImage}>Add image</button>  
        {images.map((src) => (  
          <img src={src} />  
        ))}  
      </div>  
    );  
  }  
}
```



Typowanie children i wykorzystanie wbudowanego typu React.FC

- Typowanie children w React polega na określeniu typu danych dla elementów, które są przekazywane do komponentu jako dzieci (content).
- Typowanie children umożliwia kontrolę nad typem danych, które są przekazywane i pomaga uniknąć błędów.
- Wbudowany typ React.FC jest funkcyjnym skrótem dla typu React.FunctionComponent i pozwala na łatwe tworzenie komponentów funkcyjnych.
- W połączeniu z typowaniem children, React.FC jest przydatnym narzędziem do budowy elastycznych i łatwych w użyciu komponentów w aplikacji React.




Typowanie children i wykorzystanie wbudowanego typu React.FC

```
type LayoutProps = {  
  children: React.ReactNode;  
};
```

```
function Layout({ children }: LayoutProps) {  
  return <div>{children}</div>;  
}
```

Typowanie children i wykorzystanie wbudowanego typu React.FC

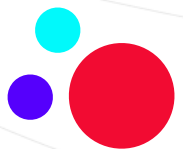
```
function App() {  
  return (  
    <>  
      <Layout>  
        <div>Content</div>  
      </Layout>  
      <Layout>Content</Layout>  
      <Layout>{undefined}</Layout>  
      <Layout>{null}</Layout>  
      <Layout>{0}</Layout>  
      <Layout>{false}</Layout>  
      <Layout>{[]}</Layout>  
      <Layout>{{}}</Layout>  
    </>  
  );  
}
```





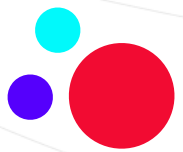
Typowanie children i wykorzystanie wbudowanego typu React.FC

```
type LayoutProps = {  
  children: React.ReactElement; // same as JSX.Element  
};
```

Typowanie children i wykorzystanie wbudowanego typu React.FC

```
function App() {  
  return (  
    <>  
      <Layout>  
        <div>Content</div>  
      </Layout>  
      <Layout>  
        <>Content</>  
      </Layout>  
      <Layout>Content</Layout>  
      <Layout>{undefined}</Layout>  
      <Layout>{null}</Layout>  
      <Layout>{0}</Layout>  
      <Layout>{false}</Layout>  
      <Layout>{[]}</Layout>  
      <Layout>{{}}</Layout>  
    </>  
  );  
}
```

`React.ReactNode`, `React.ReactText`, `React.ReactElement` i `React.ReactChild`

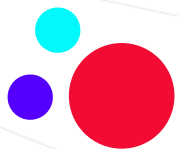
- Wykorzystanie typów `React.ReactNode`, `React.ReactText`, `React.ReactElement` i `React.ReactChild` w React polega na określeniu typów danych dla elementów, które są renderowane w aplikacji.
- Typy te są używane do opisanego różnego rodzaju elementów, takich jak tekst, elementy HTML, komponenty lub ich mieszanki.
- Dzięki temu, programiści mogą łatwo kontrolować typy danych, które są przekazywane do aplikacji i unikać potencjalnych błędów.
- Typy te są często używane w komponentach, które muszą obsługiwać różne rodzaje danych jako swoje dzieci.



Komponenty generyczne oraz generyczne wyrażenia funkcyjne

- Komponenty generyczne i generyczne wyrażenia funkcyjne w React pozwalają na tworzenie uniwersalnych komponentów i funkcji, które są w stanie pracować z różnymi typami danych.
- Dzięki temu, programiści mogą napisać jedno rozwiązanie, które może być użyte w wielu miejscach w aplikacji, zamiast pisać osobne komponenty dla każdego typu danych.
- Generyczne wyrażenia funkcyjne pozwalają na określenie typu danych jako argumentu, co umożliwia pracę z różnymi typami danych w jednej funkcji.
- Komponenty generyczne i generyczne wyrażenia funkcyjne są często używane w React do tworzenia bardziej elastycznych i uniwersalnych rozwiązań.

<https://profy.dev/article/react-typescript#using-generics>



Typowanie hooków: useState, useRef oraz custom hooków

```
function useFireUser(firstName: string) {  
  const [isFired, setIsFired] = useState(false);  
  const hireAndFire = () => setIsFired(!isFired);  
  
  return {  
    text: isFired ? `Oops, hire ${firstName} back!` : "Fire this loser!",  
    hireAndFire,  
  };  
}
```

```
function UserProfile({ firstName, role }: UserProfileProps) {  
  const { text, hireAndFire } = useFireUser(firstName);  
  return (  
    <>  
      <div>Hi {firstName}, you suck!</div>  
      <button onClick={hireAndFire}>{text}</button>  
    </>  
  );  
}
```

```
// Challenge: the two errors in useImages and Product
```

```
import { useState } from "react";
```

```
type ProductProps = {  
  name: string,  
  price: number,  
  images: string[],  
};
```

```
function useImages(initialImages) {  
  const [images, setImages] = useState(initialImages);
```

```
  const addImage = () => {  
    setImages(images.concat(`image-${images.length + 1}.png`));  
  };  
  return { images, setImages };  
}
```

```
export function Product({ name, price, images: initialImages }: ProductProps) {  
  const { images, addImage } = useImages(initialImages);
```

```
  return (  
    <div>  
      <div>  
        {name} ${price}  
      </div>  
      <button onClick={addImage}>Add image</button>  
      {images.map((src) => (  
        <img src={src} />  
      ))}  
    </div>  
  );  
}
```



Typowanie obiektów reprezentujących zdarzenia

- React używa swojego własnego systemu zdarzeń.
- Nie możesz używać typowych MouseEvents lub podobnych na swoich elementach.
- Na szczęście typowanie React daje ci właściwy odpowiednik każdego zdarzenia, które możesz znać ze standardowego DOM.
- Mają nawet tę samą nazwę, co może być czasami kłopotliwe.

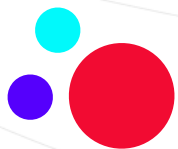


Typowanie obiektów reprezentujących zdarzenia (ogólne)

```
import React, { Component, MouseEvent } from "react";
```

```
export class Button extends Component {  
  handleClick(event: MouseEvent) {  
    event.preventDefault();  
    alert(event.currentTarget.tagName); // alerts BUTTON  
  }  
}
```

```
render() {  
  return <button onClick={this.handleClick}>{this.props.children}</  
button>;  
}
```



Typowanie obiektów reprezentujących zdarzenia (szczegółowo)

```
import React, { Component, MouseEvent } from "react";

export class Button extends Component {
  /*
  Here we restrict all handleClicks to be exclusively on
  HTMLButton Elements
  */
  handleClick(event: MouseEvent<HTMLButtonElement>) {
    event.preventDefault();
    alert(event.currentTarget.tagName); // alerts BUTTON
  }

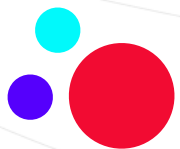
  /*
  Generics support union types. This event handler works on
  HTMLButtonElement and HTMLAnchorElement (links).
  */
  handleAnotherClick(event: MouseEvent<HTMLButtonElement | HTMLAnchorElement>) {
    event.preventDefault();
    alert("Yeah!");
  }

  render() {
    return <button onClick={this.handleClick}>{this.props.children}</button>;
  }
}
```



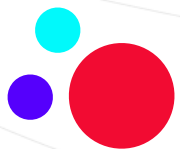

Typowanie obiektów reprezentujących zdarzenia

- Typowanie obiektów reprezentujących zdarzenia jest ważnym elementem w React. Dzięki temu można określić, jakie dane będą przetwarzane w odpowiedzi na zdarzenia, takie jak formularze, zmiany wartości i kliknięcia myszką.
- Wbudowane typy takie jak `React.FormEvent`, `React.ChangeEvent` i `React.MouseEvent` pozwalają na łatwiejsze typowanie obiektów zdarzeń.
- Typ funkcji handlera, taki jak `React.FormEventHandler`, `React.ChangeEventHandler` i `React.MouseEventHandler`, pozwala na łatwe typowanie funkcji, które są używane jako obsługa zdarzeń.



Typowanie obiektów reprezentujących zdarzenia

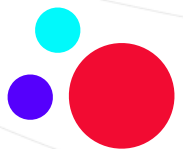
```
function FireButton() {  
  return (  
    <button  
      onClick={(event) => event.preventDefault()}  
    >  
      Fire this loser!  
    </button>  
  );  
}
```



Typowanie obiektów reprezentujących zdarzenia

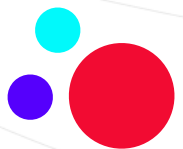
```
function FireButton() {  
  const onClick = (event: ???) => {  
    event.preventDefault();  
  };  
};
```

```
  return (  
    <button onClick={onClick}>  
      Fire this loser!  
    </button>  
  );  
}
```



Typowanie obiektów reprezentujących zdarzenia

```
function FireButton() {  
  const onClick = (event: React.MouseEvent<HTMLButtonElement, MouseEvent>) => {  
    event.preventDefault();  
  };  
  
  return <button onClick={onClick}>Fire this loser!</button>;  
}
```



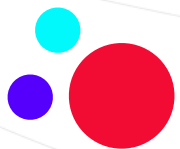
Typowanie obiektów reprezentujących zdarzenia

```
function FireButton() {  
  const onClick = (event: React.MouseEvent<HTMLButtonElement, MouseEvent>) => {  
    event.preventDefault();  
  };  
  
  return <button onClick={onClick}>Fire this loser!</button>;  
}
```



Typowanie obiektów reprezentujących zdarzenia

```
function Input() {  
  const onChange = (event: React.ChangeEvent<HTMLInputElement>) => {  
    console.log(event.target.value);  
  };  
  
  return <input onChange={onChange} />;  
}
```



Typowanie obiektów reprezentujących zdarzenia

```
function Select() {  
  const onChange = (event: React.ChangeEvent<HTMLSelectElement>) => {  
    console.log(event.target.value);  
  };  
  
  return <select onChange={onChange}>...</select>;  
}
```

// Challenge: Type the event param in the onClick handler

```
export function ProductCard() {  
  const onClick = (event) => {  
    event.preventDefault();  
  };  
}
```

```
return <div onClick={onClick}>Fire this loser!</div>;  
}
```




Zadanie 2

```
// Challenge: Type the event param in the onChangeName  
// handler and set the name correctly
```

```
import { useState } from "react";
```

```
export function CreateProductForm() {  
  const [name, setName] = useState("");
```

```
  const onChangeName = (event) => {  
    setName();  
  };
```

```
  return (  
    <form>  
      <input onChange={onChangeName} placeholder='Name' value={name} />  
    </form>  
  );  
}
```



Typowanie Context API

- Typowanie Context API jest kluczowe dla zapewnienia jakości i niezawodności w aplikacjach React.
- Pozwala na łatwiejsze zarządzanie i dostęp do danych współdzielonych przez wiele komponentów.
- Dzięki typowaniu można określić, jakie typy danych będą przechowywane i udostępniane przez kontekst, co ułatwia programistom łatwiejszą implementację i debugowanie.

- <https://jsdzem.pl/jak-typowac-react-context-api-typescript/>



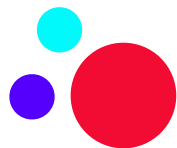
Typowanie asynchronicznego kodu

- Typowanie asynchronicznego kodu, takiego jak komunikacja z API, jest ważne dla zapewnienia niezawodności i jakości aplikacji React.
- Pozwala na określenie typu danych, które będą przesyłane jako payload oraz typu danych, które będą otrzymywane jako odpowiedź.
- Typowanie asynchronicznego kodu ułatwia debugowanie i unika błędów, które mogą wystąpić podczas wymiany danych pomiędzy aplikacją a serwerem.
- Dzięki temu programiści są w stanie łatwiej zrozumieć i kontrolować przepływ danych w aplikacji.

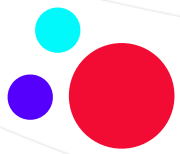


Typowanie asynchronicznego kodu

- Przykład auth context w Cyclops

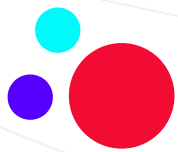


Podsumowanie zdobytej wiedzy o TypeScript i sposoby na migracje z JavaScriptu



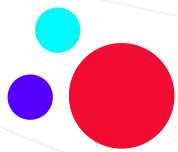
Zalety i wady typowania statycznego w TypeScript

- Zaletą typowania statycznego jest zwiększenie bezpieczeństwa kodu i uniknięcie błędów w czasie kompilacji.
- Typowanie statyczne umożliwia lepsze zrozumienie kodu przez programistów i narzędzia.
- Wady to mniejsza elastyczność i trudność w napisaniu kodu dla skomplikowanych zastosowań.
- Typowanie statyczne może wydłużyć czas pisania kodu, ponieważ wymaga dokładnego określenia typów.
- Typowanie statyczne może utrudnić pracę z bibliotekami i frameworkami, które nie są natywnie obsługiwane przez TypeScript.



Zalety i wady typowania statycznego w TypeScript

- Typowanie statyczne może powodować błędy w czasie kompilacji, jeśli nie są dokładnie określone typy.
- Typowanie statyczne umożliwia łatwiejsze debugowanie kodu i łatwiejsze wyszukiwanie błędów.
- Typowanie statyczne zwiększa wydajność aplikacji poprzez optymalizację kodu podczas kompilacji.
- Wady to także trudność w migracji kodu między różnymi wersjami TypeScript.
- Typowanie statyczne jest szczególnie przydatne dla dużych projektów z wieloma zespołami i współpracującymi bibliotekami.



Zalety i wady typowania statycznego w TypeScript

- Typowanie statyczne może powodować błędy w czasie kompilacji, jeśli nie są dokładnie określone typy.
- Typowanie statyczne umożliwia łatwiejsze debugowanie kodu i łatwiejsze wyszukiwanie błędów.
- Typowanie statyczne zwiększa wydajność aplikacji poprzez optymalizację kodu podczas kompilacji.
- Wady to także trudność w migracji kodu między różnymi wersjami TypeScript.
- Typowanie statyczne jest szczególnie przydatne dla dużych projektów z wieloma zespołami i współpracującymi bibliotekami.



Migracja z JS do TS



Migracja z JavaScript do TypeScript

- Migracja z JavaScript do TypeScript może być wykonana zarówno od dolnej jak i od górnej warstwy aplikacji.
- W przypadku migracji od dołu, jednostki kodu są stopniowo przekształcane z JavaScript do TypeScript.
- W ten sposób można zwiększyć jakość i bezpieczeństwo kodu stopniowo, bez potrzeby jednoczesnej zmiany całego projektu.
- Migracja od góry polega na przekształceniu całego projektu naraz, co może być bardziej ryzykowne i wymagać więcej pracy.
- **Bottom-up** jest preferowanym podejściem, ponieważ pozwala na szybsze i bardziej efektywne uzyskanie korzyści z TypeScript.



Migracja z JavaScript do TypeScript

- Bottom-up jest także łatwiejsze do zarządzania, ponieważ pozwala na bieżące testowanie i wprowadzanie zmian.
- W przypadku migracji od góry, trudniej jest identyfikować i rozwiązywać błędy, ponieważ wprowadzone zmiany dotyczą całego projektu.
- Bottom-up umożliwia także łatwiejsze wprowadzanie zmian i aktualizacji w przyszłości, ponieważ kod jest już w części oparty na TypeScript.
- Migracja od góry może być konieczna w przypadku projektów o dużym zasięgu i skomplikowanej architekturze.
- Ważne jest, aby wybrać odpowiednie podejście do migracji, biorąc pod uwagę specyfikę projektu i potrzeby zespołu.



Migracja z JavaScript do TypeScript

- Bottom-up jest także łatwiejsze do zarządzania, ponieważ pozwala na bieżące testowanie i wprowadzanie zmian.
- W przypadku migracji od góry, trudniej jest identyfikować i rozwiązywać błędy, ponieważ wprowadzone zmiany dotyczą całego projektu.
- Bottom-up umożliwia także łatwiejsze wprowadzanie zmian i aktualizacji w przyszłości, ponieważ kod jest już w części oparty na TypeScript.
- Migracja od góry może być konieczna w przypadku projektów o dużym zasięgu i skomplikowanej architekturze.
- Ważne jest, aby wybrać odpowiednie podejście do migracji, biorąc pod uwagę specyfikę projektu i potrzeby zespołu.

- <https://typescript-exercises.github.io/>
- <https://dev.to/typescripttv/typing-react-props-in-typescript-5hal>
- <https://github.com/typescript-cheatsheets/react#types-or-interfaces>
- <https://profy.dev/article/react-typescript#typescript-basics-required-for-react>

Dziękuję za uwagę

Jakub Wojtach