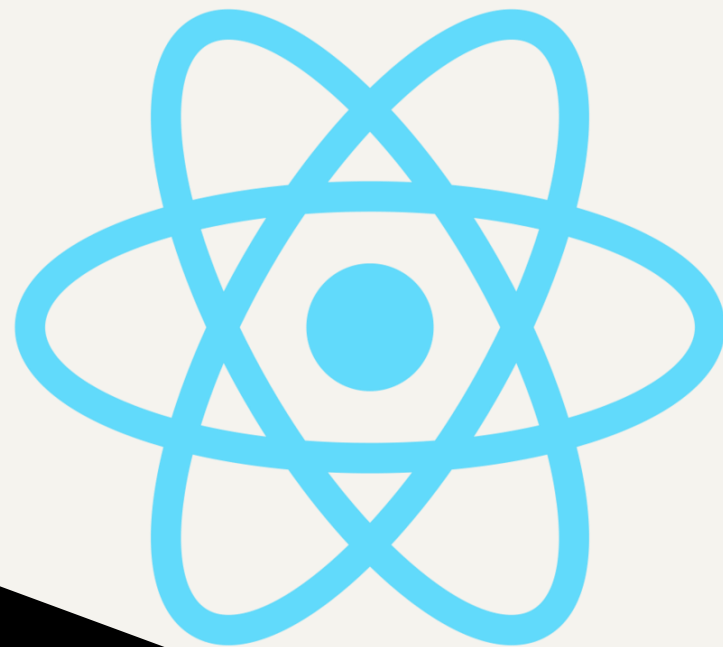




UINF/PAZ1c
epizóda 11

React:
Inštalácia, TypeScript,
komponenty a háky

Tradičné webové aplikácie

- Web = HTML + CSS + JS
- Na serveri čaká aplikácia, ktorá ich generuje
- Archaický prístup
 - Skriptovací jazyk zlepúje HTML súbor a posiela ho prehliadaču
 - Zdroják = kúsky HTML na striedačku s kusmi kódu, ktorý HTML dotvára
 - JS na webe slúži iba spríjemnenie práce s webom – animácie, kontrola vstupov vo formulári

Moderné server-side webové aplikácie

- Stále platí: server generuje HTML + CSS + JS a prehliadač ich len interpretuje
- MVC na serveri
 - HTML šablóny so špeciálnymi tagmi/atribútmi, ktoré sa odkazujú na komponenty aplikácie
 - Komponent aplikácie na základe svojho modelu doplní/nahradí príslušnú časť šablóny
 - Výsledok = šablóna + reprezentácia komponentov sa posiela zo servera ako výsledné HTML + CSS + JS
- Keď sa zmení model, dotiahne sa zo servera iba tá časť HTML, ktorá predstavuje daný komponent – AJAX volania
- V mnohých jazykoch: Java, C#, PHP, JavaScript,...
 - Java: JavaServer Faces, Apache Wicket, Vaadin,...

Moderné client-side webové aplikácie

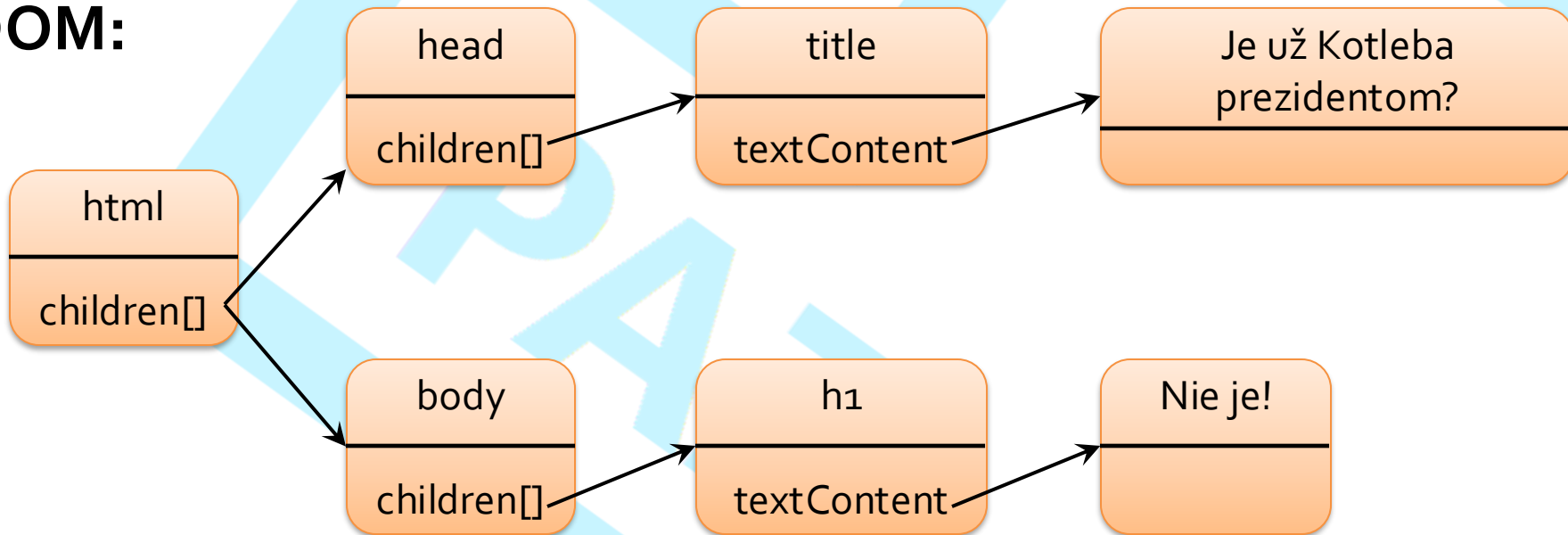
- Čistý JavaScript/TypeScript
- a.k.a. single-page applications, fat (thick) client
- Server poskytuje/prijíma entity – REST volania
 - perzistentná a business vrstva + REST service
 - ľubovoľný jazyk (Java, PHP, JavaScript,...)
- MVC v prehliadači
 - aplikácia modifikuje priamo DOM model zobrazovanej stránky
 - komponenty predstavujú podstromy DOM modelu
 - modely komponentov typicky predstavujú entity poskytované REST serverom
- Angular, Vue, React, Ember, Meteor, ExtJS, Aurelia, ..

HTML:

```
<html>
  <head>
    <title>Je už Kotleba prezidentom? </title>
  </head>
  <body>
    <h1>Nie je!</h1>
  </body>
</html>
```

DOM model tvoria JS objekty typu
Node
a okrem textových uzlov aj typu
Element, HTMLElement, ...

DOM:



React

- Final release verzie: 29.5.2013,
 - aktuálne verzia 18
- Využíva jazyk TypeScript
 - Typovaná nadstavba JavaScriptu (od ECMAScript 6)
 - podporuje OOP
 - moduly
 - lambdy
 - ...
 - dodané anotácie, generické typové parametre
- JSX/TSX – nadstavba JavaScriptu/TypeScriptu
 - Umožňuje vkladať "HTML" priamo do JS/TS

React Components

- JavaFX používa OOP a MVC
- React 16+ používa funkcionálne **komponenty**
 - Staršie verzie Reactu boli OOP a používali triedy
 - Dnes sú triedy podporované, no už sa nepoužívajú
- Každý prvok UI je iba funkcia
- Funkcia má svoje premenné (model), telo (controller) a vracia "HTML" (view)
 - **JSX/TSX** - umožňuje mixovať "HTML" s JS/TS
 - Funk. komponenty == "HTML" tagy
- Architektúra React appky
 - veľa malých autonómnych komponentov

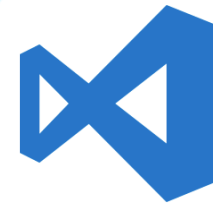
Vývojové prostredie

- Mnoho mágie na pozadí
 - TSX sa kompiluje do Typescript a VDOM (virtual DOM)
 - Typescript sa kompiluje do ECMAScript-u, ktorým rozumie webový prehliadač (ako ktorý)
 - Po každej zmene súboru
 - VDOM sa "rekonciluje" na DOM
 - NodeJS server poskytuje skompilované súbory prehliadaču
 - Prehliadačom sa napojíme na NodeJS server na <http://localhost:5173/>
 - Prehliadač si stiahne súbory a spustí našu aplikáciu
 - Chová sa tak, ako sa bude chovať, keď sa nasadí naostro

Rozbehávame vývojové prostredie

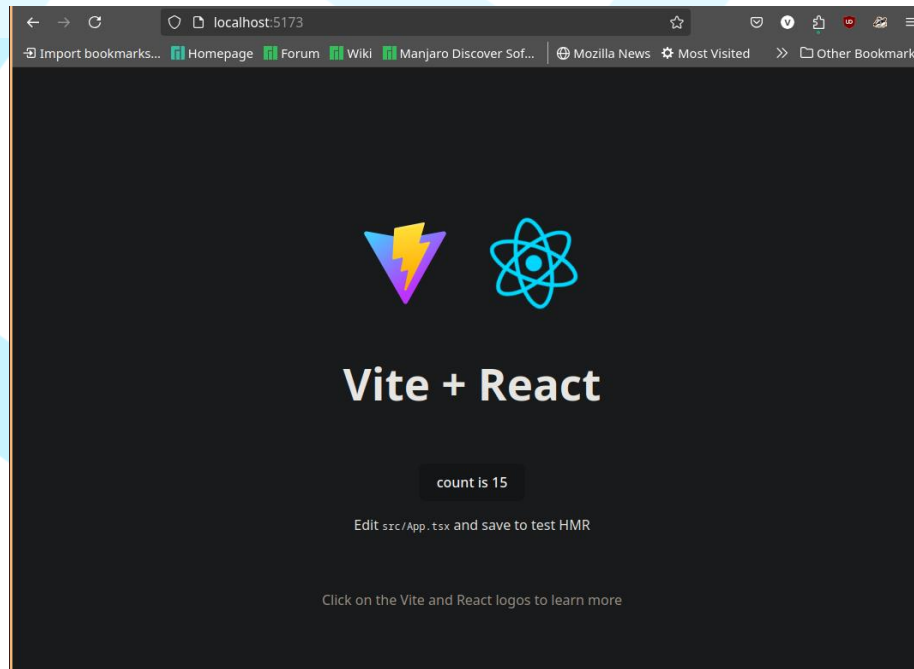


- Nainštalujeme nodejs
 - <https://nodejs.org/>
 - s ním dostaneme aj balíčkováč **npm**
- IDE:
 - Visual Studio Code:
<http://code.visualstudio.com/>
 - IntelliJ IDEA / WebStorm
- Cez Vite vieme vytvoriť kostru projektu
 - `npm create vite@latest`
 - ...vyklikáme názov projektu, React, Typescript



Spúšťame projekt

- Vojdeme do koreňa projektu a spustíme NodeJS server:
 - `npm run dev`
- Otvoríme prehliadač na URL `http://localhost:5173/`



Čo nám to vzniklo?

- node_modules

kopa modulov/knižníc, ktoré môžeme využiť

- ...

tu sa budú nachádzať naše veci

- src

- App.css

funkcia koreňového komponentu

- App.tsx

- index.css

- main.tsx

- ...

Hlavná stránka, ktorá sa v tele odkazuje na koreňový komponent



Základné typy

```
let bool: boolean = false;
```

```
let integer: number = 6;
```

```
let decimal: number = 3.14;
```

```
let hex: number = 0xf00d;
```

```
let binary: number = 0b1010;
```

```
let octal: number = 0o744;
```

NaN

Existuje číslo, ktoré nie je číslo

```
let s: number = NaN;
```

NaN nie je ekvivalentné s ničím, ani samé so sebou

```
let s = NaN;  
s === s; // false  
s !== s; // true
```

- NaN dostaneme pri
 - $1/0$
 - $\text{Infinity} * 0$
 - $\text{Infinity} - \text{Infinity}$
 - $\text{Infinity}/\text{Infinity}$
 - Neplatných konverziách

Let vs var

- **var** deklaruje premennú v rámci celej funkcie
 - Obrovský rozdiel od Javy

```
for(var i = 0; i < 5; i++){  
  console.log(i)  
}  
console.log(i) // i=5
```

- **let** deklaruje premennú v rámci bloku
 - Správa sa podobne ako deklarovanie premennej v Jave
 - **Silno preferujte let**

```
for(let i = 0; i < 5; i++){  
  console.log(i)  
}  
console.log(i) // i is undefined
```

Const

- `const` deklaruje konštantu
 - Nedá sa do nej priradiť 2x
 - Preferujte oproti `let`

Polia

```
const list: number[] = [1, 2, 3];  
list.push(4);
```

Alebo ekvivalentne

```
const list: Array<number> = [1, 2, 3];  
list.push(4);
```

Stringy

```
const fullName = "Jara Cimrman";
```

Alebo ekvivalentne

```
const fullName = 'Jara Cimrman';
```

Interpolácia stringov + viac riadkové stringy cez **backtick**

```
const age = 57;
```

```
const sentence: string = `Nazdar, volam sa ${fullName}.
```

```
Buduci mesiac budem mat ${age + 1} rokov.`;
```

Prázdné hodnoty

- **null** je prázdná hodnota
- **undefined** je prázdný typ
 - "neexistuje", "nie je vytvorené"
 - Neinicializovaná premenná je automaticky typu **undefined**

```
let a = null;  
console.log(a);    // null  
console.log(typeof a); // object
```

```
let b: undefined;  
console.log(b);    // undefined  
console.log(typeof b); // undefined
```

Ekvivalencia

== robí "chytrú" ekvivalenciu

`1 == '1' // true`

=== robí striktnú ekvivalenciu

`1 === '1' // false`

Používajte ===

```
> typeof NaN
```

```
< "number"
```

```
> 9999999999999999
```

```
< 10000000000000000
```

```
> 0.5+0.1==0.6
```

```
< true
```

```
> 0.1+0.2==0.3
```

```
< false
```

```
> Math.max()
```

```
< -Infinity
```

```
> Math.min()
```

```
< Infinity
```

```
> []+[]
```

```
< ""
```

```
> []+{}
```

```
< "[object Object]"
```

```
> {}+[]
```

```
< 0
```

```
> true+true+true===3
```

```
< true
```

```
> true-true
```

```
< 0
```

```
> true==1
```

```
< true
```

```
> true===1
```

```
< false
```

```
> (!+[ ]+[ ]+![]).length
```

```
< 9
```

```
> 9+"1"
```

```
< "91"
```

```
> 91-"1"
```

```
< 90
```

```
> []==0
```

```
< true
```



Aliasy

- Môžeme nejakému typu dať iný názov - alias

```
type Color = number;
```

Aliases a funkciové typy

- TS podporuje funkciové typy priamo
- Na lambdy nie sú potrebné jednometódové rozhrania ako v Java

```
type BinaryOperation  
  = (a: number, b: number) => number;
```

Obecné typy

- **unknown** je skoro ako Object v Jave
 - Hocičo môžeme priradiť do **unknown**

```
const aa: unknown = 4;
```

- **any** je ako premenná v čistom JavaScripte
 - Hocičo môžeme priradiť do **any**
 - Môžeme volať ľubovoľnú metódu, aj keď ju objekt (možno) nemá
 - Používa sa častejšie ako **unknown**

```
const bb: any = 4;  
bb.aha()
```

Uniony

```
type Color = 'red' | 'green' | 'blue' | number;  
const x: Color = 'red';  
const y: Color = 0xFFFFFFFF;  
const z: Color = 'ahoj';
```

- Union sa často používa namiesto enumu
- Union môže špecifikovať konkrétne hodnoty ľubovoľného typu (`stringy 'red', 'green', 'blue'`), ... alebo ľubovoľnú hodnotu konkrétneho typu (`number`)

Truthy & Falsy

```
if ("haha") {  
  ...  
}
```

- Do **if**-u vieme dávať **truthy**, alebo **falsy** hodnoty

Typ	Truthy	Falsy
boolean	true	false
string	hocičo, okrem ""	""
number	hocičo, okrem 0, NaN	0, NaN
null	nikdy	vždy
undefined	nikdy	vždy
objekty, polia	vždy	nikdy

- Pretypovanie na boolean

```
const bb = "haha"  
const bbExists: boolean = !!bb  
const bbExistsNegated: boolean = !bb
```

Objektové typy

```
type Student = {  
  name: string;  
  age: number;
```

Pomenovaný objektový typ

```
  address: {  
    city: string;  
    country: string;  
  }
```

Anonymný objektový typ

Typy môžu mať hlavičky
metód

```
  sayNameAndAge(youMust: boolean): string;
```

<alebo ekvivalentne ako inštančnú premennú typu funkcia>

```
  sayNameAndAge: (youMust: boolean) => string;
```

```
}
```

Inštancia objektového typu

```
const julka: Student = {  
  name: 'Julka',  
  age: 21,  
  address: {  
    city: 'Presov',  
    country: 'Slovakia'  
  },  
  
  sayNameAndAge(youMust: boolean): string {  
    if (youMust) {  
      return `Som ${this.name} a mam ${this.age} rokov`;  
    }  
    return 'nepoviem'  
  }  
}
```

Prípadnú metódu musíme hneď implementovať

```
julka.sayNameAndAge(true);
```

Triedy

Trieda môže implementovať
objektový typ.

```
class StudentImpl implements Student {  
  constructor(  
    public name: string,  
    public age: number,  
    public address: { city: string, country: string }  
  ) {}
```

Inštančné premenné sú zvyčajne verejné. Ak IP
nepochádza z rozhrania/typu, dá sa použiť
`private`

```
  sayNameAndAge(youMust: boolean): string {  
    if (youMust) {  
      return `Som ${this.name} a mam ${this.age} rokov`;  
    }  
    return 'nepoviem';  
  }  
}
```

```
const jano = new StudentImpl('Jano', 20, { city: 'Kosice', country: 'Slovakia' });  
const janoPovedal = jano.sayNameAndAge(true);
```

Triedy

Trieda môže implementovať objektový typ.

```
class StudentImpl implements Student {  
  constructor(  
    public name: string
```

Inštančné premenné sú zvyčajne verejné. Ak

Triedy sa v **moderných**
Typescript frameworkoch nepoužívajú často
(aspoň nie v Reacte a Vue)

```
    return 'nepoviem';  
  }  
}
```

```
const jano = new StudentImpl('Jano', 20, { city: 'Kosice', country: 'Slovakia' });  
const janoPovedal = jano.sayNameAndAge(true);
```

Rozhrania

- TS podporuje aj **interface** a abstraktné triedy
- Trieda tiež môže implementovať rozhranie
- Podobné OOP ako v Java
- Môžu mať hlavičky metód **aj inštančné premenné**
- Dve rozhrania s rovnakým menom v balíčku sa automaticky spoja

```
interface Person { id: number; }
```

```
interface Person { name: string; }
```

```
const lenka: Person = {  
  id: 1,  
  name: 'Lenka Mala'  
}
```

Objektový typ vs rozhranie

- Rozhrania sú **takmer identické** s objektovými typmi, ale
 - Vieme ich rozširovať cez **extends**
 - Dve rozhrania s rovnakým menom v balíčku sa automaticky spoja
 - Pri typoch však viem robiť uniony (User | Car | **string**)
- Použite **typ**, ak chcete "entitu"
 - Chcem najmä inštančné premenné
 - Žiadne, alebo zopár jednoduchých metód
- Použite **rozhranie**, ak chcete OOP
 - Chcem to implementovať triedou
 - Alebo implementovať anonymne

Objektový typ vs rozhranie

- Rozhrania sú **takmer identické** s objektovými typmi, ale
 - Vieme ich rozširovať cez **extends**
 - Dve rozhrania s rovnakým menom v balíčku sa automaticky spoja
 - Pri typoch však viem robiť uniony (User | Car | **string**)
- Použite **typ**, ak chcete "entitu"

Je to ale takmer jedno.

Mnohí preferujú vždy to, alebo ono.

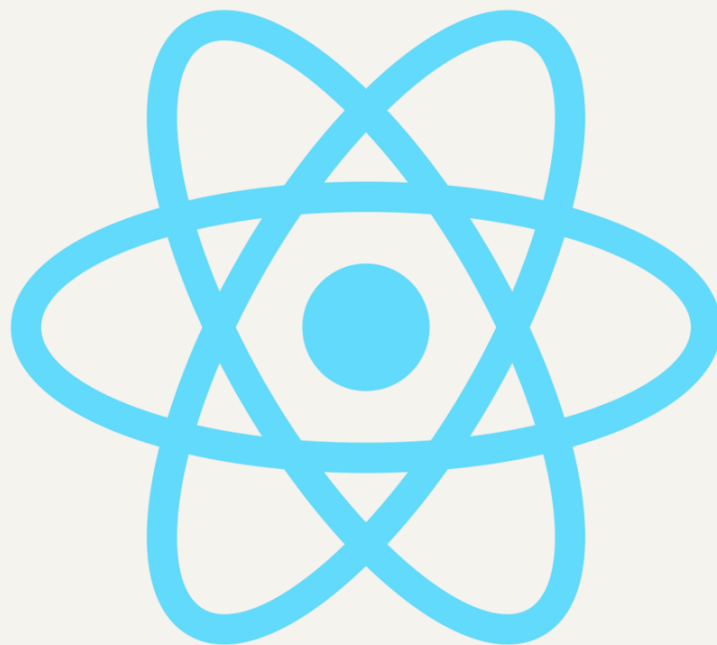
- Alebo implementovať anonymne

Kód vo viacerých súboroch

export umožní použitie v
iných súboroch

```
// student.ts  
export type Student {  
  ...  
}
```

```
// tabulka.ts  
import {Student} from student;  
  
const jano = {...};
```



Vráťme sa k Reactu

Vložme vlastný komponent do hlavného

- Vytvoríme nový súbor `UserList.tsx`
- V ňom vytvoríme exportovanú funkciu, ktorá vracia objekt typu `JSX.Element`
- Tú zavoláme vo forme JSX tagu v `App.tsx`

Vloženie komponentu do stránky

src/main.tsx

```
createRoot(document.getElementById('root')!).render(  
  <StrictMode>  
    <App />  
  </StrictMode>,  
)
```

src/App.tsx

```
function App() {  
  return (  
    <div className="App">  
      ...  
    </div>  
  );  
}
```

Zobrazenie zoznamu pomocou f-cie map

src/UserList.tsx

Návratový typ
je React.JSX.Element, ale
nemusíme ho uvádzať

```
function UserList() {  
  const users = ['Evzen', 'Alica', 'Boris']  
  return (  
    <div>  
      <h1>Zoznam pouzivatelov</h1>  
      <ul>  
        {users.map((user) => (  
          <li>{user}</li>  
        ))}  
      </ul>  
    </div>  
  );  
}
```

<div>...</div> je objekt typu React.JSX.Element

```
export default UserList;
```

Radšej pracujme s entitami

src/Entities.ts

```
export type User = {  
  id?: number;  
  name: string;  
  chipId: string;  
  active: boolean;  
  admin: boolean;  
  access: ChipReader[];  
}  
  
export type ChipReader = {  
  id?: number;  
  door: string;  
}
```

id?: number;
je skratka pre
id: number | undefined;

Premenná v TS **nesmie** byť null
ani undefined, pokiaľ to
neuvediete cez **?**, alebo **union**

Vytvorme si použivatel'ov

src/UserList.tsx

```
const users: User[] = [  
  {  
    id: 1, name: "Evzen", chipId: "123", active: true, admin: true,  
    access: []  
  },  
  {  
    id: 2, name: "Alica", chipId: "456", active: true, admin: false,  
    access: [{id: 1, door: "P10"}]  
  },  
  {  
    id: 3, name: "Boris", chipId: "789", active: true, admin: false,  
    access: [  
      {id: 1, door: "P10"},  
      {id: 2, door: "SAC103"}  
    ]  
  },  
];
```

Iterujeme entity

src/UserList.tsx

```
const users: User[] = [...]  
  
function UserList() {  
  return (  
    <div>  
      <h1>Zoznam pouzivatelov</h1>  
      <ul>  
        {users.map((user) => (  
          <li>{user.name}</li>  
        )))}  
      </ul>  
    </div>  
  );  
}
```

Odchytenie udalosti click

src/UserList.tsx

Stav manažujeme pomocou
use* funkcií => hooks (háčiky)

```
function UserList() {  
  const [selectedUser, setSelectedUser] = useState<User | undefined>(undefined);  
  
  const handleClick: MouseEventHandler = (event) => {  
    const clickedUserName = event.currentTarget.textContent;  
    const clickedUser = users.find((user) => user.name === clickedUserName);  
    setSelectedUser(clickedUser);  
  }  
  
  return (  
    <div>  
      <ol>  
        {users.map((user) => (  
          <li key={user.id} onClick={handleClick}>{user.name}</li>  
        ))}  
      </ol>  
    </div>  
  );  
}
```

Podmienené časti šablóny

src/UserList.tsx

```
function UserList() {  
  ...  
  return (  
    <div>  
      ...  
      {selectedUser && (  
        <div>  
          <h1>{selectedUser.name}</h1>  
          ...  
        </div>  
      )}  
    </div>  
  );  
}
```

Pravá strana **&&** je vždy **truthy** a tá sa vloží do stránky, ...

... akk ľavá strana **&&** je tiež **truthy**

Otázky?

