React 2

Jiří Zacpal

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

KMIWEBA Webové aplikace

React

- Doposud jsme knihovnu React používali poměrně omezeným způsobem.
- V principu totiž vytváříme modulární aplikaci (v našem případě jsou těmito moduly React komponenty), ale JavaScript je ze své podstaty monolitický (načítá se vždy celý skript).
- Navíc toto omezené použití neumožňuje využít nejnovější možnosti knihovny React.

JS moduly

- Do nedávna měl JavaScript poměrně omezené možnosti pro vytváření modulárního kódu.
- S příchodem Javascript ES6 byly do JS přidány JS moduly, které vše znatelně usnadňují. Dnes jsou JS moduly standardně podporovány všemi prohlížeči.
- Při práci s moduly se používají klíčová slova export a import. export označuje část kódu, kterou bude možné importovat do jiné části kódu (pomocí import). Jednoduchý příklad následuje.
- Uvažme následující modul (soubor App.js)

```
export const PI = Math.PI;
export function degreesToRadians(d) {return d * PI / 180;}
```

- Modul načteme a použijeme následovně.

```
<script type="module">
  import {PI} from './App.js';
  console.log(PI);
</script>
```

JS moduly

- Výhodou uvedeného kódu je, že se nenačítá celý soubor App.js, ale pouze požadovaná potřebná část.
- Ruční práce s moduly je poměrně obtížná a typicky se pro správu modulů používají různé nástroje jako například Webpack, které umožňují sestavení aplikace programátorem (ve výše uvedeném je sestavení ponecháno na klientech).
- Knihovna React obsahuje skript create-react-app, který umožňuje připravit vývojové workflow pro vývoj pomocí této knihovny.
- V kostce je provedena
  - automatická konfigurace sestavování modulů,
  - zpřístupňují se externí moduly z npm,
  - provádí se automatická kontrola kódu (syntaxe i best practices, obsahuje ESlint), stejně tak i jeho automatické sestavení a spuštění.

```
npm install -g create-react-app
create-react-app hello-world
cd hello-world
npm start
```

- Ve výchozím stavu jsou všechny zdrojové kódy uloženy ve složce src. Nyní máme vše připravené na pokročilejší práci s knihovnou React.

React hooks

- Doposud měli možnost mít stav pouze objektové komponenty.
- React hooks byly představeny v zatím poslední verzi knihovny React a umožňují vytvářet funkční komponenty s vnitřním stavem.
- Tímto objektové komponenty ztrácejí smysl (ve skutečnosti jsou v knihovně postupně utlačovány, jejich odstranění se ale zatím neplánuje).

useState

```
// načtení hooku
import React, { useState } from 'react';

// funkční komponenta
function Example() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default Example;
```

React hooks

useEffect

- React knihovna obsahuje několik dalších hooku. V následující ukázce je použit useEffect, který umožňuje provádět vedlejší efekt (například načtení dat, výpis do konzole, změna DOM a další). useEffect je spuštěn při každém renderování (tedy i při prvním).

```
// načtení hooks
import React, { useState, useEffect } from 'react';

// funkční komponenta
function Example() {
  const [count, setCount] = useState(0);

useEffect(() => {
  console.log (`You clicked ${count} times`);
});

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default Example;
```

React hooks

useContext

- Dalším užitečným hookem je useContext, který umožňuje vytvářet globální stav. To je poněkud nepřesné označení, jelikož React komponenty jsou uspořádány ve stromu a jednotlivé vlastnosti mohou být přejímány od rodičů v tomto stromu (pokud je uvedeno).
- useContext umožňuje vytvářet props, které jsou sdíleny v daném podstromu.

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

const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

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

React hooks

```
function Example() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      ...
     </button>
  );
}

export default Example;
```

React hooks

Další hooky

- Za zmínku ještě stojí useReducer hook, který z aktuálního stavu vytváří stav nový. Například přepínání true a false, které je ukázáno níže.

```
const [checked, toggle] = useReducer(checked => !checked, false);
```

- Užitečný hook je také useMemo, který umožňuje zapamatovat při renderování vypočítané hodnoty. Ty následně nemusí být opět přepočítávány (pouze pokud dojde ke změně hodnot, ze který byly vypočítány). Tento hook slouží pro optimalizaci výkonu.
- Hooky je možné volat pouze na top-level úrovni. Nesmí být v cyklech a podmínkách (můžou je ale obsahovat). Hook by měl být volán pouze uvnitř komponenty. ESlint v create-react-app tato pravidla hlídá.
- Kromě vestavěných hooks je možné vytvářet vlastní.

Získávání dat

- Z pohledu aplikace jako celku, je každá komponenta samostatný celek, který zabezpečuje danou funkcionalitu. Například seznam uživatelů by mohl být komponenta. Tato komponenta by měla data nejen vizualizovat, ale také si je získat od serveru. Získání dat jsme již ukázali. Následuje příklad integrace fetch() do React komponenty.

```
export function GitUser({login}) { // destukturalizace props
  const [data, setData] = useState();

  useEffect(() => {
    if(!login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(r => r.json())
      .then(setData)
      .catch(console.error);
  }, [login]); // [] je dependeci array (kdy má být hook spušten), pokud je [] tak jen na začátku

  if(!data) return loading...;

  if(data) {
    return {JSON.stringify(data, null, 2)}
  }
}
// poznámka: vyzkoušet lze například <GitUser login="martin-trnecka" />
```

Získávání dat

- Pro řízení spuštění useEffect() se používá dependenci pole. Dodejme, že hooků stejného typu může být i více a mohou se lišit v dependenci poli. Vždy je dobré složitější funkcionalitu rozložit do více hooků.
- Při ladění výše uvedené komponenty si můžeme všimnout, že fetch() je volán dvakrát. To se děje pouze ve vývojovém módu. V produkční verzi je volání pouze jedno. Spuštění produkční verze lze provést (například) následovně.

```
npm install -g serve
npm run build # sestavení aplikace
serve -s build
```

Posílání dat

- Pro úplnost dodejme, že pomocí fetch() je možné data i poslat.

```
const formData = new FormData();

formData.append("name", "Samantha");
formData.append("surname", "Carter");

fetch("/USER/ADD", {method: "POST", body: formData}).catch(console.error);
```

Uložení dat

- Doposud jsme uvažovali, že uložení dat je v režii webového backendu. Tím, že jsme značnou část funkcionality přesunuli na webový frontend, může být výhodné udržovat data i lokálně u klienta. Zde je nutné upozornit, že to má svá úskalí, jelikož u klienta nejsme schopni zaručit integritu a perzistenci dat.
- JS umožňuje ukládat data do uložiště prohlížeče nebo do webové databáze. Obě jsou součástí prohlížeče.

Uložení dat

- Na straně klienta je možné data ukládat do dvou uložišť: local strage a session storage.
- Obě umožňují ukládat data ve tvaru klíč-hodnota.
  - Local storage je omezné na 2–5 MB (v závislosti na prohlížeči) a uchovává data i po ukončení prohlížeče. Tato data mohou být uživatelem vymazána (jedná se o data prohlížeče, může je tedy vymazat i operační systém, typicky se tak ale děje na přání uživatele).
  - Session storage uchovává data pouze po dobu existence session (ukončení prohlížeče data vymaže), velikost je omezena maximální pamětí, kterou lze přidělit procesu (tedy prohlížeči). Dodejme, že session storage nefunguje v anonymním režimu prohlížeče. Obsah storage je možné prohlížet ve webovém prohlížeči v nástrojích pro vývojáře. Ukázka práce s local storage následuje.

```
const d = new Date();
const user = {name: "Samantha", surname: "Carter"};

localStorage.setItem("user", JSON.stringify(user));
localStorage.setItem("time", d.getTime());

const savedUser = JSON.parse(localStorage.getItem("user"));
console.log(`${savedUser.name} ${savedUser.surname}`);
```

- Později ukážeme, že možnosti uložení dat na straně klienta jsou v JS mnohem rozsáhlejší.

Kam dál?

- Ukázali jsme pouze základní koncepty a práci s knihovnou React. Velkým tématem je v poslední době server side rendering. Doposud jsme nechali generování webové aplikace na klientovi. V principu je možné nechat vygenerovat aplikaci webový server, který klientovi pošle pouze to co má být zobrazeno. Výhodou je, že kód na serveru může být totožný s kódem, který jsme doposud psali na straně klienta.
- V tomto kontextu se můžeme setkat s pojmem izomorfní (nebo také univerzální) kód, který může běžet na straně klienta i serveru.
- Vygenerováním na straně serveru je dosaženo vysoké rychlosti. Velmi poplulární nástroje jsou například Gatsby a Next.js, které podporují server side rendering.
- Dodejme, že možné jsou oba směry, tedy kód z klienta přesunout na server a stejně tak kód ze serveru je možné přesunout na klienta. Příkladem je knihovna axions, primárně určená pro node.js o kterém budeme mluvit později. Tato knihovna slouží pro vytváření AJAX requestů je možné ji použít i na straně klienta.
- Na knihovně React je vystavěna technologie React native, která umožňuje generovat nativní (přesněji řečeno hybridní) aplikace pro různé platformy. Například iOS nebo Android. Obvzvláště ve spojení s Expo Go nabízí velmi zajímavé možnosti vývoje.

Úkoly

ÚKOL 1

- Výpis logů v sekci dashboard realizujte jako React komponentu. Přidejte možnost smazat, konkrétní log a možnost vypsat pouze zadaný počet posledních logů (tuto informaci uchovávejte pouze na straně klienta).