Kódím.cz

/ Webové aplikace / Digitální akademie Web / React

11 Async/await a ošetření chyb

Ošetření chybových stavů API a použití async/await místo then()

Předchozí
Opakování: LeviExpress, druhá část

Nemá následující lekci

Asynchronní funkce

Jak si možná pamtujete z některé z předchozích lekcí, funkce `fetch`, kterou používáme ke stahování dat ze serveru, vrací takzvaný *promise*. Funkcím, které vracejí *promise*, se obecně říká *asynchronní funkce*. Asynchronnost nám umožňuje spustit nějakou operaci (například stažení dat ze serveru) a nemuset při vykonávání kódu čekat, než se tato operace dokončí.

Představme si například volání API, které bez ošetření chyb vypadá takto:

```
const fetchData = () => {
  fetch('https://random.kodim.app/api/diceroll')
    .then((resp) => resp.json())
    .then((data) => setRoll(data.result.number));
  console.log('Konec funkce fetchData()');
};

fetchData();
console.log('Zavolána funkce fetchData()');
```

Když je kód obsluhující nějaký *promise* složitější, můžeme se v něm pro samé `then` metody začít ztrácet. Takové situaci se mezi programátory říká *callback hell*.

V novějších verzích JavaScriptu existují dvě nová klíčová slova, která umožňují napsat kód přehledněji: `async` a `await`. S použitím `async` / `await` můžeme kód zjednodušit a zlepšit čitelnost:

```
const fetchData = async () => {
  const resp = await fetch('https://random.kodim.app/api/diceroll');
  const data = await resp.json();
  setRoll(data.result.number);
  console.log('Konec funkce fetchData()');
};

console.log('Zavolána funkce fetchData()');
```

Klíčovým slovem `await` vlastaně říkáme, že na místě, kdy se volá nějaká asynchronní funkce se chceme opravdu zastavit a počkat, dokud se operace skutečně nedokončí. Možná si řeknete, co jsme tím získali, když hlavním smyslem *promisů* je, abychom nemuseli čekat na dlouho trvající operace. Vtip je v tom, 6e pokud nějaká funkce obsahuje volání pomocí `await`, musí se sama také stát asynchronní. Musíme ji tedy vytvořit pomocí slovíčka `async`. Taková funkce automaticky vrací `Promise` a opět bychom ji buď museli volat s `await` a nebo se smířit s tím, že při jejím volání nebudeme mít její výsledek.

V Reactu to většinou funguje tak, že se nám touto cestou asynchronnost zpropaguje až do nějakého posluchče události, který stejně nic nevrátí, takže klidně může být asynchronní, aniž by se nám v kódu cokoliv rozbilo.

Ošetření chyb serveru

Pro ošetření chyb serveru můžeme použít `if-else` nebo `switch`, stejně jako v předchozí části lekce. Uvidíte, že kód s `await` je čitelnější (porovnejte si ho s kódem z předchozí části lekce):

```
const fetchData = async () => {
  const resp = await fetch('https://random.kodim.app/api/diceroll');
  if (resp.status === 200) {
    setErrorMessage(null);
  } else if (resp.status === 500) {
    setErrorMessage('Server vrátil chybu.');
    return;
  } else if (resp.status === 503) {
    setErrorMessage('Server je přetížen.');
    return;
  }
```

```
  const data = await resp.json();
  setRoll(data.result.number);

  console.log('Konec funkce fetchData()');
};
```

Kód se v tomto případě provádí jednoduše shora dolů, lépe se v něm proto orientuje. Všimněte si, že tentokrát nemusíme ošetřovat případ, že `data` neexistují – část kódu pro zpracování dat je pouze ve větvi ošetřující případ, kdy server vrátil `200 OK` – můžeme očekávat, že v takovém případě nám server data poslal.

Připomeňme, že funkce `then()` (a také `catch()`) nejsou specifické pro `fetch`. Funkce `fetch` vrací objekt typu `Promise`, funkce `then()` a `catch()` jsou metodami právě objektu `Promise`. Objekt `Promise` mohou vracet i jiné funkce – např. funkce pro načtení souboru z disku nebo funkce pro export obrázku z `canvasu`. Používá se v moderních API všude tam, kde nějaká operace může trvat delší dobu a nechceme, aby její volání zablokovalo prohlížeč. Použít `async/await` můžeme ve všech případech, kdy máme k dipozici objekt `Promise`, ne jen spolu s fukcí `fetch`.

Předchozí
Cvičení: Ošetření chyb

Následující
Použití try-catch pro ošetření fatálních chyb