POKROČILÝ JAVASCRIPT

Pokročilé programování v JavaScriptu — objektově orientované programování a další programovací paradigmata, balíčkovací nástroj npm podrobně, asynchronní programování a sliby.

MOTIVACE

Na webových stránkách se nám často stává, že na něco v programu čekáme. JS běží na jednom vlákně a proto obyčejné čekání často způsobuje to, že celý program čeká na to, než se v kódu něco získá. To na reálných webových stránkách není možné. Představte si, že nahráváte soubor a mezitím nemůžete na stránce na cokoliv klikat - a třeba to ani pozastavit.

Proto v JS vzniklo asynchronní programování, které je založeno na **slibech**.

Poznámky:

Žádné poznámky k této lekci jste ještě nezapsali.

SLIBY

Sliby, tzv. promises, je způsob, jak v JavaScriptu můžeme říct, že na něco má kód čekat a co dělat, když výsledek dostane a případně co dělat, když se něco pokazí. Volání jsou na sobě nezávislé, a proto deklarace slibu neblokuje následující příkazy za voláním.

Na slib se můžeme zachytit pomocí funkce `then`, která se zavolá právě tehdy, když předchozí promise byla úspěšná. `then` se mohou řetízkovat a tím se hodnota předává mezi jednotlivé hodnoty. Například tedy `then` může vracet hodnotu, který dostane další zavolání `then`. Může to tedy vpadat takto:

```
// Funkce generateCatName nám vrací Promise.
//  promise bude splněna v neznámém čase
//  a vygeneruje náhodné jméno pro kočku.
const cat = generateCatName();

// cat není k dispozici - ještě jsme kočku nezískali
cat
    .then(name => {
        console.log(`Mám kočku ${name}`);
    })
    .then(() => {
        console.log(`Kočka nahrána!`);
    });
```

Když nám generateCatName() úspěšně vrátí promise, která se někdy splní, tak se vypíše její jméno a i hodnota `Kočka nahrána`. Můžeme například tedy i vracet hodnoty:

```
// Funkce generateCatName nám vrací Promise.
//  promise bude splněna v neznámém čase
//  a vygeneruje náhodné jméno pro kočku.
const cat = generateCatName();

// cat není k dispozici - ještě jsme kočku nezískali
cat
    .then(name => {
        console.log(`Mám kočku ${name}`);
        return [name, 'Yoko'];
    })
    .then((data) => {
        const name1 = data[0];
        const name2 = data[1];

        console.log(`Mám kočku ${name1} a ${name2}`);
    });
```

Metoda, která vrací promise, nebo obecně sliby jsou tvořeny různými způsoby. Různé knihovny k nim přistupují různě a i některé věci uvnitř JS s nimi rovnou pracují. My si ukážeme, jak můžeme vytvořit jednoduchou promise. Promise se vytváří z nového objektu Promise, která přijímá jednu anonymní funkci s dvěma parametry:

- `resolve` - jedná se o volatelnou funkci s neomezeným počtem parametrů, která indikuje, že se slib povedl a mají se zavolat další volání `then`.
- `reject` - jedná se o volatelnou funkci, která voláním indikuje, že se slib nepovedl.

```
const generateCatName = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Kitty');
        }, 1000);
```

```
    });
};
const cat = generateCatName();

// cat není k dispozici - ještě jsme kočku nezískali
cat
    .then(name => {
        console.log(`Mám kočku ${name}`);
        return [name, 'Yoko'];
    })
    .then((data) => {
        const name1 = data[0];
        const name2 = data[1];

        console.log(`Mám kočku ${name1} a ${name2}`);
    });
```

Poznámky:

Žádné poznámky k této lekci jste ještě nezapsali.

NESPLNĚNÍ SLIBU

Někdy se stane, že požadavek na slib se nepovede. Například nastane chyba či nemůže být splněn. V tomto případě promise místo volání `then` najde nejbližší zachycení `.catch`, které se zavolá s kódem. Může taktéž něco vracet a po zavolání se volají následující bloky `.then`. Často proto dáváme `catch` až na konec všech volání.

```
const generateCatName = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if(Math.random() > 0.5) {
                resolve('Kitty');
            } else {
                reject();
            }
        }, 1000);
    });
};

generateCatName().then(cat => {
    console.log(`The cat is ${cat}`);
}).catch(() => {
    console.log('Cat is not here.');
});
```

Volání po `catch` vypadá následovně:

```
new Promise((resolve, reject) => {
    console.log("Spouštím!");

    resolve();
})
.then(() => {
    throw new Error("Kočička neni kočkoholka");

    console.log("Oops!");
})
.catch(() => {
    console.error("Stala se (ne)očekávána chyba, prosíme, zkuste to později");
})
.then(() => {
    console.log("Přidáno do logu");
});
```

Poznámky:

Žádné poznámky k této lekci jste ještě nezapsali.

GARANCE

Sliby jsou komplikované a proto existují garance. Garance nám říká, jak se budou sliby chovat vždy, tedy něco jako standard. Garance jsou následující:

- funkce přidávány pomocí `.then` budou volány v přesném pořadí a až potom (a jen když) bude předchozí funkce úspěšně splněna.
- funkce přidávány pomocí `.then` budou zavolány i po tom, co byly přidány již na vyřešenou promise.
- pokud `.then` a nebo promise selže, najde se v řetízkovém volání nejbližší `.catch` .

Poznámky:

Žádné poznámky k této lekci jste ještě nezapsali.

POMOCNÉ VOLÁNÍ

Jako pomocné volání pro sliby máme několik metod na objektu `Promise`. Jedná se o:

- `.all` — zavolání všech předaných promise najednou a najednou získá jejich data
- `.race` — zavolání všech předaných promise najednou, vrátí první dokončenou (klidně s chybou)
- `.any` — zavolání všech předaných promise najednou, vrátí první splněnou (bez chyby)

```
const generateCatName = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(['Yoko', 'Koko', 'Moko'][Math.floor(Math.random() * 3)]);
        }, 1000);
    });
};


const cats = Promise.all([generateCatName(), generateCatName(), generateCatName()]);

cats.then(catNames => {
    console.log(catNames);
});
```

Dalšími pomocnými metodami je:

- `.reject` — vrátí vytvořenou promise, která je již dokončena a skončil s určenou chybou. Můžeme ji zachytit.
- `.resolve` — vrátí vytvořenou promise, která je již dokončená a skončila úspěšná. Danou hodnotou najdeme v dalším voláním.

```
let promise = Promise.resolve();

for(let i = 0; i < 10; i++)
   promise = promise.then(() => {
      return new Promise((resolve) => {
         setTimeout(() => resolve(), i * 1000)
      })
   });

promise.then(() => {
   console.log("Hotovo"); // Vypíše se za 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55 sekund
})
```

Poznámky:

Žádné poznámky k této lekci jste ještě nezapsali.

ASYNCHRONNÍ FUNKCE

Promises je skvělý způsob tvoření asynchronních věcí. Začínají být však špatné tehdy, když musíte čekat na více hodnot z více zdrojů, protože nastane slibové-peklo. Proto vznikl tzv. syntax sugar, tedy zlepšení syntaxe zápisu promises pomocí asynchronních funkcí.

Přibyli nám dvě nové klíčové slova a to `async`, který přidáváme před funkci, abychom ji označili jako asynchronní. To můžeme dělat i u anonymních funkcí. A `await`, který přidáváme před příkaz, který volá promise a říká, že čekáme na to, až se promise vykoná (a vrátí výsledek).

Označením funkce jako asynchronní se nyní změnění její návratový typ na slib.

```
async function cat() {
    const catName = await getCatName();
    const catName2 = await getCatName();

    catName = catName.toUpperCase();
    catName2 = catName2.toUpperCase();

    return catName;
}

cat().then(catName => {
    console.log(catName);
});
```

Poznámky:

Žádné poznámky k této lekci jste ještě nezapsali.