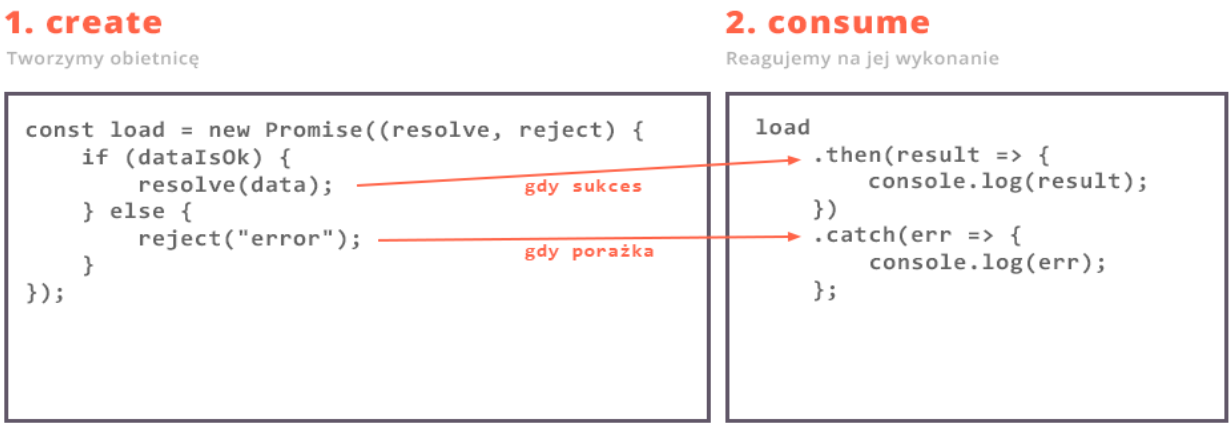
Promise

Obietnice (Promise) pozwalają nam nieco inaczej podejść do pracy z asynchronicznymi operacjami.

Dzięki nim możemy wykonać jakiś kod, a następnie odpowiednio zareagować na jego wykonanie. Można powiedzieć, że to taka inna odmiana funkcji callback.

*Kiedy kupisz mi burgera, **wtedy** będę zadowolony*

Praca z obietnicami w zasadzie zawsze składa się z 2 kroków. Po pierwsze za pomocą konstruktora Promise tworzymy obietnicę. Po drugie za pomocą odpowiedniej funkcji które nam udostępnia reagujemy na jej zakończenie. Zupełnie jak w powyższym zdaniu.



#

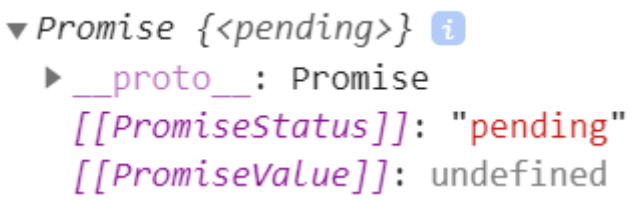
Tworzenie Promise

Do stworzenia Promise korzystamy z konstruktora `Promise()`, który w parametrze przyjmuje funkcję, do której przekazujemy referencję do dwóch funkcji (tak zwanych egzekutorów), które będą wywoływane w przypadku zwrócenia poprawnego lub błędnego kodu.

```
const promise = new Promise((resolve, reject) => {
    if (zakończono_operacje_pozytywnie) {
        resolve("Wszystko ok 😀"); //kończę i zwracam dane
    } else {
        reject("Nie jest ok 😬"); //kończę i zwracam błąd
    }
});
```

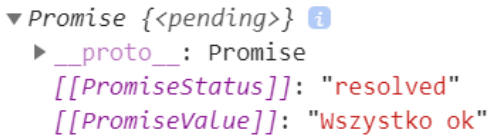
Każda obietnica może zakończyć się na dwa sposoby - powodzeniem (resolve) i niepowodzeniem (reject).
Gdy obietnica zakończy się powodzeniem (np. dane się wczytają), powinniśmy wywołać funkcję **resolve()**, przekazując do niej rezultat działania. W przypadku błędów powinniśmy wywołać funkcję **reject()**, do której przekażemy błędne dane lub komunikat błędu.

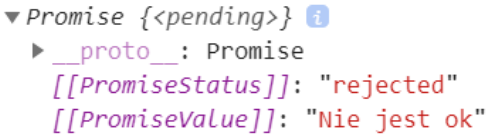
Po stworzeniu nowego obiektu Promise, w pierwszym momencie jest on w stanie oczekiwania (w debuggerze widoczna jest właściwość [[PromiseStatus]] ustawiona na "pending" oraz właściwość [[PromiseValue]] ustawiona na undefined.



Gdzieś w tle dzieją się asynchroniczne operacje, które zajmują jakiś czas (np. trwa ściąganie danych).

W momencie zakończenia wykonywania takich operacji Promise przechodzi w stan "settled" (ustalony/załatwiony) i zostaje zwrócony jakiś wynik. Status takiego promise przełączany jest odpowiednio w "fulfilled" lub "rejected", a my jako programiści wywołujemy przekazane w parametrach funkcje (resolve lub reject) z przekazanymi do nich wynikami operacji.





#

Konsumpcja Promise

Po zakończeniu Promise możemy zareagować na jej wynik. Służą do tego dodatkowe metody, które klasa Promise nam udostępnia. Pierwszą z nich jest metoda **then()**. Pozwala ona reagować zarówno na pozytywne rozwiązanie obietnicy, negatywne jak i oba na raz:

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => { //symulujemy ściąganie danych
        resolve("Przykładowe dane");
    }, 1000);
});

promise.then(result => {
    //obietnica zakończyła się pozytywnie
    console.log(result)
});
function doSomething() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Przykładowe dane");
        }, 1000);
    });
}

doSomething()
    .then(res => {
        console.log(res)
```

```
    });
```

#

catch()

Obietnica może zakończyć się pozytywnie (resolve) lub negatywnie (reject). Do reakcji na negatywną odpowiedź możemy użyć albo drugiego parametru funkcji `then()`, albo funkcji **catch()** (stosowane częściej).

```
function doSomething() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            //resolve("Gotowe dane");
            reject("Przykładowy błąd"); //niestety nie udało się ściągnąć
danych, zwracamy więc błąd
        }, 1000);
    });
}

doSomething()
    .then(result => {
        ...
    })
    .catch(error => {
        console.error(error);
    });
```

#

Promise.all()

Bardzo często nasze czynności chcielibyśmy zacząć wykonywać dopiero po zakończeniu wszystkich asynchronicznych operacji. Przykładem takiej sytuacji może być widok użytkownika, na którym wyświetlamy jego dane, jego galerię, książki, posiadane zwierzęta (koniecznie!...). Aby poczekać na zakończenie wszystkich takich operacji użyjemy funkcji **Promise.all()** do której przekażemy tablicę zawierającą nasze obietnice:

```
function loadUser() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("user data"); }, 2000)
    });
}

function loadBooks() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("books data"); }, 1000)
    });
}

function loadPets() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("pets data"); }, 500)
    });
}


Promise.all([
```

```
    loadUser(),
    loadBooks(),
    loadPets()
])
.then(res => {
    console.log(res); //["user data", "books data", "pets data"]
});
const promise1 =  new Promise((resolve, reject) => {
    setTimeout(() => { resolve("data1"); }, 1000)
});

const promise2 =  new Promise((resolve, reject) => {
    setTimeout(() => { resolve("data2"); }, 2000)
});

function loadData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("loaded data"); }, 500)
    });
}


Promise.all([
    promise1,
    promise2,
    loadData(),
])
.then(res => {
    console.log(res); //["data1", "data2", "loaded data"]
});
```

#

Promise.allSettled()

Podobną w działaniu jest funkcja **allSettled()**.

Różnica w porównaniu z `all()` jest taka, że funkcja `all()` zwraca wynik, gdy wszystkie przekazane do niej obietnice zostaną zakończone pozytywnie, natomiast `allSettled()` zwraca wynik gdy się zakończą - nie ważne czy pozytywnie czy negatywnie.

```
function loadUser() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("user data"); }, 2000)
    });
}

function loadBooks() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { reject("błąd danych"); }, 1000) //błąd danych!
    });
}

function loadPets() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("pets data"); }, 500)
    });
}
```

```
Promise.all([
    loadUser(),
    loadBooks(),
    loadPets(),
])
.then(res => {
    console.log(res);
})
.catch(err => {
    console.log(err);
})

/*
rezultat:
"błąd danych", ponieważ jedna z funkcji zwróciła reject
*/
function loadUser() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("user data"); }, 2000)
    });
}

function loadBooks() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { reject("błąd danych"); }, 1000) //błąd danych!
    });
}

function loadPets() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("pets data"); }, 500)
    });
}

Promise.allSettled([
    loadUser(),
    loadBooks(),
    loadPets(),
])
.then(res => {
    console.log(res);
})
.catch(err => {
    console.log(err);
})

/*
rezultat:
[
    {status: "fulfilled", value: "user data"},
    {status: "rejected", reason: "błąd danych"},
    {status: "fulfilled", value: "pets data"},
]
*/
```

#

Promise.race()

Jeżeli powyższe obie metody czekały na zakończenie wszystkich obietnic, tak metoda **Promise.race()** zwróci pierwszą zakończoną obietnicę:

```
function loadUser() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("user data"); }, 2000)
    });
}

function loadBooks() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { reject("błąd danych"); }, 1000) //błąd danych!
    });
}

function loadPets() {
    return new Promise((resolve, reject) => {
        setTimeout(() => { resolve("pets data"); }, 500)
    });
}

Promise.race([
    loadUser(),
    loadBooks(),
    loadPets(),
])
.then(res => {
    console.log(res); //"pets data"
})
.catch(err => {
    console.log(err);
})
```

W powyższym przypadku zostały zwrócone dane `"pets data"`. Wynika to tylko z faktu, że jako pierwsza wykonała się obietnica z funkcji `loadPets()`. Gdyby pierwsza zakończyła się obietnica z `loadBooks()` (reject), kod przeszedł by do funkcji `catch()`.

#

Promise.finally()

Funkcja `finally()` pozwala nam zareagować na moment, kiedy wszystkie operacje się zakończą - bez względu czy zakończyły się pozytywnie czy negatywnie. Wszystkie osoby, które używały `$.ajax()` w jQuery z pewnością pamiętają zdarzenie `complete()` - to właśnie jego odpowiednik.

```
button.classList.add("loading"); //pokazujemy loading
button.disabled = true; //i wyłączamy button

fetch("....")
    .then(res => res.json())
    .then(res => console.log(res))
    .catch(err => console.log(err))
    .finally(() => {
        button.classList.remove("loading");
        button.disabled = false;
    });
```

#

Łańcuchowe obietnice

Jeżeli dana funkcja zwróci nam nową obietnicę, możemy na niej wykonać jedną z powyższych funkcji czyli `then()`, `catch()` itp.

Każda z takich funkcji także zwraca obietnicę, więc możemy wykonać kolejne operacje za pomocą kolejnych `then()`:

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("ok"), 1000);
});

promise
    .then(res => {
        console.log(res); //"ok"
        return res + "2";
    })
    .then(res => {
        console.log(res); //"ok2"
        return res + "3";
    })
    .then(res => {
        console.log(res); //"ok23"
    })
function checkDataA() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("OK1"), 2000);
    });
}

function checkDataB() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("OK2"), 2000);
    });
}

function checkDataC() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("OK3"), 2000);
    });
}

function checkDataD() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("OK4"), 2000);
    });
}

checkDataA()
    .then(res => checkDataB())
    .then(res => checkDataC())
    .then(res => {
        console.log(res);
        return checkDataD()
    })
    .then(res => {
```

```
        console.log(res);
    });

//lub
checkDataA()
    .then(checkDataB)
    .then(checkDataC)
    .then(checkDataD)
    .then(res => {
        console.log(res);
    });
```

Takie łańcuchowe wywoływanie kolejnych obietnic jest o tyle istotne, ponieważ bardzo często przy tworzeniu funkcji używających obietnic, niektóre operacje będziemy wykonywać wewnątrz jakiś funkcji, natomiast resztę poza jej ciałem reagując na zwróconą obietnicę:

```
function makeThings() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("obietnica, ");
        }, 2000);
    }).then(res => {
        return res + " pierwsza zmiana, ";
    }).then(res => {
        return res + " druga zmiana";
    })
}

makeThings()
    .then(res => {
        console.log("na zewnątrz: ", res);
    })
```

Przyda nam się to szczególnie, gdy będziemy pisać kod pobierające dane z serwera. Część obróbki danych wykonamy wewnątrz funkcji, natomiast cała reszta trafi poza funkcję.

```
function loadData(countryName) {
    return fetch(`https://restcountries.com/v3.1/name/${countryName}`)
//fetch zwraca nam obietnicę
        .then(res => { //then też zwraca nam obietnicę
            if (res.ok) {
                return res.json();
            } else {
                return Promise.reject(`Http error: ${res.status}`);
            }
        })
}

loadData("Poland")
    .then(res => {
        console.log(res);
    })
    .catch(err => {
        console.error(err);
    })
```

O funkcji fetch będziemy uczyć się w jednym z kolejnych rozdziałów. Dla nas najważniejsze teraz jest to, że sama w sobie zwraca obietnicę, dzięki czemu mogę zareagować na jej zakończenie - dokładnie tak jak w poprzednich przykładach.

Zauważ przy okazji, że do zwrócenia błędu wykorzystałem funkcję statyczną `Promise.reject()`, którą udostępnia nam klasa Promise (drugą jest `Promise.resolve()`).

W powyższej sytuacji mogę też rzucić po prostu błędem:

```
function loadData(countryName) {
    return fetch(`https://restcountries.com/v3.1/name/${countryName}`)
        .then(res => {
            if (res.ok) {
                return res.json();
            } else {
                throw new Error(`Http error: ${res.status}`);
            }
        })
}
```