daftcode

ASYNCHRONOUS JS

Konrad Świercz

JAVASCRIPT IS SINGLE THREADED

EVENT LOOP

```
// `eventQueue` is an array that acts as a queue (first-in, first-out)
const eventQueue = [ ];

// keep going "forever"
while (true) {
  // perform a "tick"
  if (eventQueue.length > 0) {
    // get the next event in the queue
    let event = eventQueue.shift();

    // now, execute the next event
    try {
      event();
    }
    catch (err) {
      reportError(err);
    }
  }
}
```

Źródło: You Don't Know JS: Async & Performance, Kyle Simpson

RUN-TO-COMPLETION

Każda funkcja w JS będzie działała od początku do końca bez przerwy.

ASYNCHRONOUS JAVASCRIPT

CALLBACKS

- Naming convention
- Funkcja na później

```
setTimeout(function callback() {}, 1000);

window.addEventListener('load', function callback() {});

ajax('https://www.somepage.com', function callback() {});
```

PROBLEMY Z CALLBACKAMI

CALLBACK HELL

PYRAMID OF DOOM

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });

      }
    })
  }
});
```

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continue after all scripts are loaded
  }
```

INVERSION OF CONTROL

```
ajax('http://www.somepage.com', function(data) {
  // some very important business logic here
});
```

```
function success(data) {
  console.log( data );
}

function failure(err) {
  console.error( err );
}

ajax( "http://some.url.1", success, failure );
```

```
function response(err,data) {
  if (err) {
    throw new Error(err);
  }

  console.log( data );
}

ajax( "http://some.url.1", response );
```

PROMISE

Obiekt reprezentujący przyszłą wartość asynchronicznej operacji

```
const promise = new Promise(function(resolve, reject) {
  // EXECUTOR
});
```

Cechy Promise'a

Result: `undefined`

State:

- PENDING
- FULFILLED
- REJECTED

resolve(value)

- State: FULFILLED
- Resylt: value

reject(error)

- State: REJECTED
- Result: error

Executor wykonuje się automatycznie i od razu, w momencie stworzenia promise'a (`new Promise`)

Argumenty `resolve` / `reject` są zaimplementowane w JS.

Executor powinien wykonać jakąś akcję (asynchroniczną?) i wywołać jedną z nich zmieniając stan Promise'a.

```
const delaySuccess = function(time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('done!'), time);
  });
};
```

- State: FULFILLED
- Result: 'done!'

```
const delayFailure = function(time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject('ooops'), time);
  });
};
```

- State: REJECTED
- Result: 'ooops'

Promise może mieć tylko jeden wynik

Po zmianie z **PENDING** na **RESOLVED/REJECTED** jej stan nie zmienia się

```
const promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("…")); // ignored
  setTimeout(() => resolve("…")); // ignored
});
```

KONSUMERZY

- .then()
- .catch()
- .finally()

.then()

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

```
delaySuccess(1000).then(
  (result) => console.log(result), // > done!
  (error) => console.error(result), // ignored
);

delayFailure(1000).then(
  (result) => console.log(result), // ignored
  (error) => console.error(result), // > oops
);
```

```
.catch()
```

===

```
.then(null, () => {})
```

Dokładnie to samo, shorthand

`.finally()`

Uruchamiana niezależnie czy fullfilled czy rejected.

Nie dostaje żadnych argumentów, nie wie w jakim stanie jest Promise'a

Jest 'przezroczysta' => podaje argumenty dalej

W CZYM PROMISE JEST LEPSZY?

Rzeczy dzieją się w naturalym dla ludzkiego mózgu porządku

.then można dodać wiele razy, **RESOLVED** od razu je wykona, **PENDING** grzecznie zaczeka

Kontrola znowu w rękach programisty

PROMISE CHAINING

FIGHTING CALLBACK HELL

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then((result) => {

  alert(result); // 1
  return result * 2;

}).then((result) => {

  alert(result); // 2
  return result * 2;

}).then((result) => {

  alert(result); // 4
  return result * 2;

});
```

To nie jest Promise chaining:

```
const promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then((result) => {
  alert(result); // 1
  return result * 2;
});

promise.then((result) => {
  alert(result); // 1
  return result * 2;
});

promise.then((result) => {
  alert(result); // 1
  return result * 2;
});
```

ZWRACANIE PROMISE Z `.then`

Normalnie, wartość zwrócona z .then jest natychmiast podawana dalej

Jeżeli zwracamy Promise, cały łańcuch czeka na jego resolve/reject

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then((result) => {

  alert(result); // 1

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then((result) => {

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then((result) => {

  alert(result); // 4

});
```

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continue after all scripts are loaded
  }
```

```
function step1(script) {
  // do stuff with script 1

  return loadScript('2.js');
}

function step2(script) {
  // do stuff with script 2

  return loadScript('3.js');
}

function step3(script) {
  // do stuff with script 3
  // ...continue after all scripts are loaded
};

loadScript('1.js')
  .then(step1)
  .then(step2)
  .then(step3)
  .catch(handleError);
```

ŹLE!

```
loadScript('1.js').then((script1) => {
    // do stuff with script 1

    loadScript('2.js').then((script2) => {
      // do stuff with script 2

      loadScript('3.js').then((script3) => {
        // do stuff with script 2

        // ...continue after all scripts are loaded
      });
    });
  })

  .catch(handleError); // ??????????????????????
```

ERROR HANDLING - PROMISES

Jeżeli promise reject'uje, kontrola przeskakuje do najbliższego rejection handlera w dół łańcucha.

```
.then(() => {}, errorHandler)
```

```
.catch(errorHandler)
```

```
fetch('https://www.nie-ma-takiego-adresu.pl') // REJECT
  .then(response => response.json())
  .then(json => stuff(json))
  .catch(err => alert(err)) // TypeError: Failed to fetch
```

```
fetch('https://www.jest-taki-adres-ale-nie-umie-w-jsona.pl')
  .then(response => response.json()) // REJECT
  .then(json => stuff(json))
  .catch(err => alert(err)) // SyntaxError: Unexpected token < in JSON at position 0
```

BŁĄD W EXECUTOR'ZE

Jest traktowany tak samo jak reject

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

```
new Promise((resolve, reject) => {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

RETHROWING

`.catch()` może rzucić błędem...

```
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch((error) => {

  alert(error); // Whoops!
  alert("Ogarnięte");

}).then(() => alert("Wszystko w porządku"));
```

```
new Promise((resolve, reject) => {

  throw new Error("Whoops!");

}).catch((error) => {

  if (error instanceof URIError) {
    // ogarnę
  } else {
    alert("Nie umiem takiego błędu");

    throw error;
  }

}).then(function() {
  /* some stuff */
}).catch(error => {

  alert(`Wystąpił nieogarnięty błąd: ${error}`);

});
```

UNHANDLED REJECTION

Global error

Skrypt umarł...

R.I.P.

PROMISE API

`Promise.resolve()`

Zwraca resolve'owany Promise z daną wartością

Przydatne jeżeli znamy wartość od razu, ale chcemy ją wrappować w promise

```
function loadData(url) {
  const cache = getCache();

  if (cache.has(url)) {
    return Promise.resolve(cache.get(url));
  }

  return fetch(url)
    .then(response => response.json())
    .then(text => {
      cache.set(url, text);
      return text;
    });
}
```

Promise.reject()

Zwraca reject'owany Promise z daną wartością

```
Nie udało mi się wymyślić przykładu :)
```

`Promise.all([promises])`

Jako argument podajemy tablicę promises

Jeżeli wszystkie resolve'ują, zwraca tablicę z ich wynikami

Jeżeli którakolwiek reject'uje, zwraca jej błąd

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // [1, 2, 3]
```

Resolve po 3s

[1, 2, 3] mimo, że wyniki były 3, 2, 1

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => reject('Ups'), 1000))  // REJECT
]).catch(alert); // Ups
```

Reject już po 1s

Nie ma mechanizmu **`cancel`** więc pozostałe się wykonają tylko zostaną zignorowane

`Promise.race([promises])`

Jako argument podajemy tablicę promises

Podobna do **`Promise.all()`** ale zamiast czekać na wszystkie czeka na pierwszą i resolve'uje z jej wynikiem

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  2, // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(alert); // [1, 2, 3]
```

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  Promise.resolve(2), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // [1, 2, 3]
```

MACROTASK QUEUE
VS.
MICROTASK QUEUE

Macrotask - jeden na TICK, w konkretnym miejscu

Microtask - wszystkie na raz, w konkretnym miejscu ale też po każdym opróżnieniu stacku

`async / await`

Specjalna składnia do pracy z Promises imitująca synchroniczność

async

Funkcja zawsze zwraca Promise

Można użyć **await**

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

await

Może być użyty tylko w funkcji **async**

Każe JS poczekać na wynik Promise'a

```
const delaySuccess = function(time) {
  return new Promise((resolve) => {
    setTimeout(() => resolve('done!'), time);
  });
};

async function f() {
  const result = await delaySuccess(1000); // CZEKA 1s

  return result; // 'done!'
}

f();
```

```
async function f() {
  const result = await delaySuccess(1000); // CZEKA 1s

  return result; // 'done!'
}

const f = async () => {
  const result = await delaySuccess(1000); // CZEKA 1s

  return result; // 'done!'
};

class Obj {
  async f() {
    const result = await delaySuccess(1000); // CZEKA 1s

    return result; // 'done!'
  }
}
```

ERROR HANDLING

```
async function f() {

  try {
    const response = await fetch('https://www.nie-ma-takiego-adresu.pl');
  } catch(err) {
    alert(err); // TypeError: Failed to fetch
  }
}

f();
```

```
async function f() {
  let response = await fetch('https://www.nie-ma-takiego-adresu.pl');
}

f().catch(alert); // TypeError: Failed to fetch
```

await Promise.all([promises])

```
const results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

```
async function loadScripts() {
  const script1 = await loadScript('1.js');

  // do stuff with script 1

  const script2 = await loadScript('2.js');

  // do stuff with script 2

  const script3 = await loadScript('3.js');

  // do stuff with script 3
  // ...continue after all scripts are loaded
}

loadScripts();
```

BROWSER API'S

- setTimeout()
- setInterval()
- requestAnimationFrame()
- requestIdleCallback()

`requestAnimationFrame()`

Wywołuje się przed następnym repaint'em

Stworzona głównie do animacji

Dopasowuje się do częstotliwości odświeżania ekranu, ale generalnie stara się wykonywać 60 razy na sekundę

`requestIdleCallback()`

Kolejkuje funkcję do wywołania, kiedy przeglądarka nie ma co robić.

Przeznaczona do wykonywania mało istotnych rzeczy

Można podać drugi parametr (opcje) w którym podajemy timeout (za ile maksymalnie może się wykonać)

debounce

Funkcja limitująca częstotliwość wywoływania innej funkcji

keypress, scroll, resize

https://davidwalsh.name/javascript-debounce-function

https://www.npmjs.com/package/debounce

daftcode

DZIĘKI!