Technologie Internetu. Programowanie asynchroniczne

Aleksander Denisiuk (denisjuk@pja.edu.pl)
Polsko-Japońska Akademia Technik Komputerowych
Wydział Informatyki w Gdańsku
ul. Brzegi 55, 80-045 Gdańsk

27 maja 2020

Programowanie asynchroniczne

Najnowsza wersja tego dokumentu dostępna jest pod adresem
http://users.pja.edu.pl/~denisjuk/

Promise'y

Asynchroniczność

- ✔ Większość funkcji jest wykonana asynchronicznie
- ✔ Na przykład:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

- ✔ Asynchroniczność może stworzyć problem:

```
loadScript('/my/script.js'); // definicja funkcji
newFunction(); // nie ma takiej funkcji!
```

- ✔ Jak rozwiązać?

Rozwiązanie: callback


```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

✔ przykładowo

```
loadScript('/my/script.js', function() {
  newFunction(); // teraz wszystko działa
  ...
});
```

Dwa skrypty


```
loadScript('/my/script.js', function(script) {
  alert(`${script.src} załadowany, kolejny...`);

  loadScript('/my/script2.js', function(script) {
    alert(`drugi też`);
  });

});
```

Trzy skrypty


```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ... i tak dalej
    });

  })

});
```

✔ Więcej?

Opracowanie błędów


```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror
   = () => callback(new Error(`unable to load ${src}`))

  document.head.append(script);
}
```

✔ Wzorzec *error-first callback*

Przykładowy callback


```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // opracowanie błędu
  } else {
    // normalne wykonanie
  }
});
```

Więcej callback'ów


```
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 {
            // ...i tak dalej
          }
        });

      }
    })
  }
});
```

Piramida zagłady


```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

Rozwiązanie


```
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 {
    // ...i tak dalej
  }
};
```

✔ Problemy:

- ✘ kod nie jest spójny
- ✘ funkcje są wykorzystane tylko jeden raz

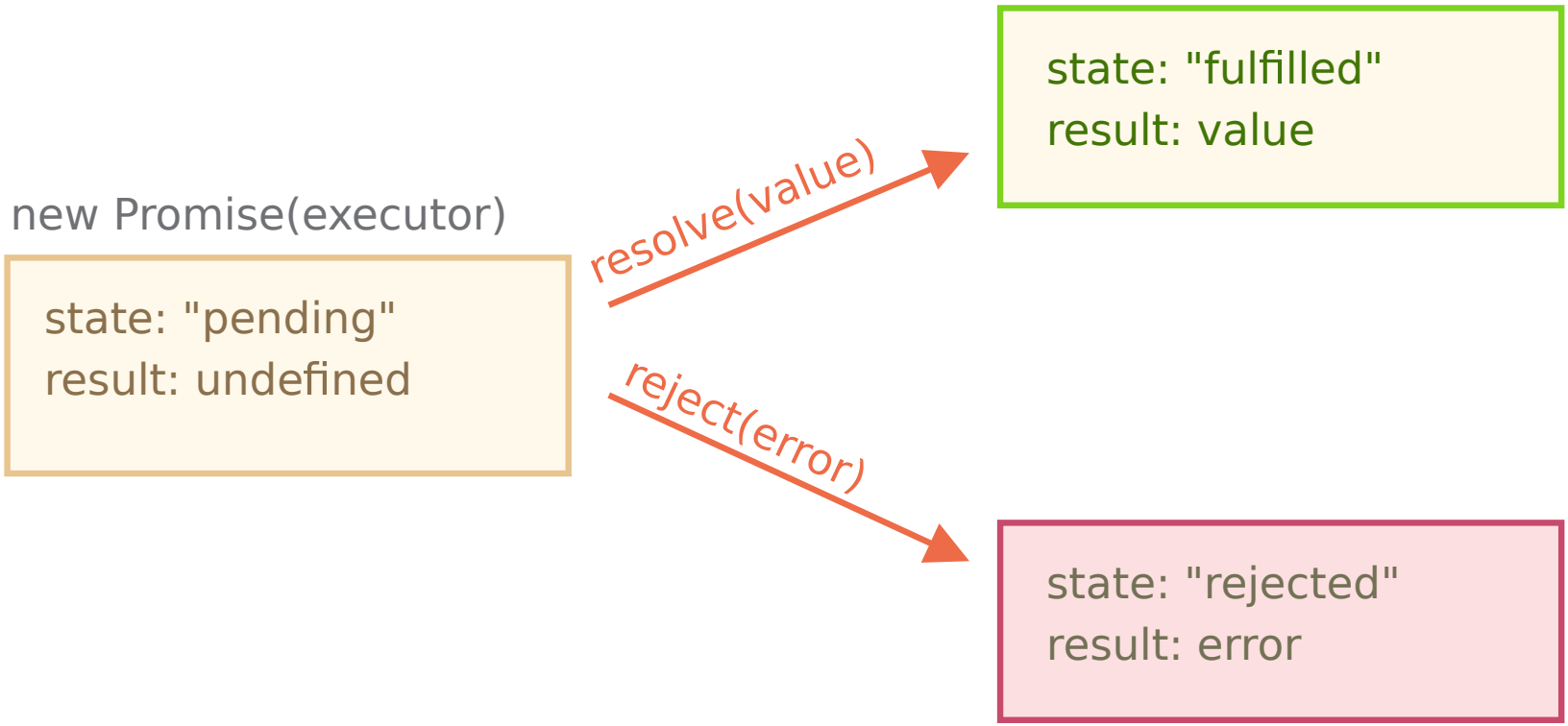
✔ Lepsze rozwiązanie — Promise

Obiekt Promise


```
let promise = new Promise(function(resolve, reject) {
  // funkcja-wykonawca (executor)
});
```

- ✔ Funkcja-wykonawca odpala się w momencie tworzenia promise'a
  - ✘ Po otrzymaniu wyniku `value` powinna wywołać `resolve(result)`
  - ✘ W przypadku błędu wywołuje się `reject(error)`
    - ✓ `error` jest obiektem `Error` bądź potomnym do niego
- ✔ `resolve` i `reject` to są callbacki *systemowe (wbudowane)*

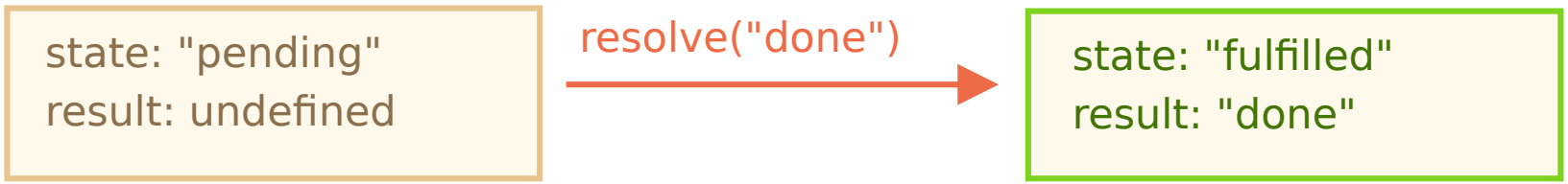
Właściwości wewnętrzne obiektu `Promise`



Właściwości wewnętrzne. Przykłady


```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done"), 1000);
});
```

new Promise(executor)




```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

new Promise(executor)

Właściwości wewnętrzne. Uwagi

- ✔ Stan promise'a może zostać zmieniony tylko jeden raz, wszystkie kolejne wywołania `resolve`/`reject` zostaną zignorowane
- ✔ Callback `reject` powinno się wywoływać tylko z obiektem `Error` (bądź potomnym)
- ✔ Callbacki `resolve`/`reject` mogą zostać wywołane od razu
- ✔ Właściwości `state` i `result` są wewnętrzne, brak bezpośredniego dostępu

Wykorzystanie wyników

- ✔ Wynik Promise'a jest opracowywany w callbacku
- ✔ Callback jest rejestrowany za pomocą metod `.then`, `.catch`, `.finally`

Metoda .then


```
promise.then(
  function(result) { /* fulfilled */ },
  function(error) { /* rejected */ }
);
```

✔ tylko fulfilled:

```
promise.then(
  function(result) { /* fulfilled */ }
);
```

✔ tylko error

```
promise.then(
  null,
  function(error) { /* rejected */ }
);
```

Metoda .then, przykład *resolved*


```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(
  result => alert(result), // wyświetli "done!"
  error => alert(error) // nie będzie wykorzystany
);
```

Metoda .then, przykład *rejected*


```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

promise.then(
  result => alert(result), // nie będzie wykorzystany
  error => alert(error) // wyświetli "Error: Whoops!"
);
```

Metoda .catch

- ✔ Opracowanie błędu: `.catch(errorHandlingFunction)` jest równoważne z `.then(null, errorHandlingFunction)`
- ✔ Przykład:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

promise.catch(
  error => alert(error) // wyświetli "Error: Whoops!"
);
```

Metoda `.finally`

- ✔ Callback odpala się w obu przypadkach
  - ✘ działania, które trzeba wykonać niezależnie od wyniku
- ✔ Sterowanie jest przekazywane na następny callback
- ✔ Przykład:

```
new Promise((resolve, reject) => {
  /* wykonać obliczenia, po czym wywołać
             resolve/reject */
})
  .finally(() => schować pasek postępu)
  .then(result => pokazać wynik,
        err => pokazać błąd)
```

Przykład z załadowaniem skryptu


```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror =
      () => reject(new Error(`Błąd ${src}`));

    document.head.append(script);
  });
}
```

Wykorzystanie


```
let promise = loadScript("https://cdnjs.cloudflare.com/
ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} załadowany!`),
  error => alert(`Błąd: ${error.message}`)
);
promise.then(script => alert('kolejny callback...'));
```

✔ Zobacz

Łańcuchy promise'ów

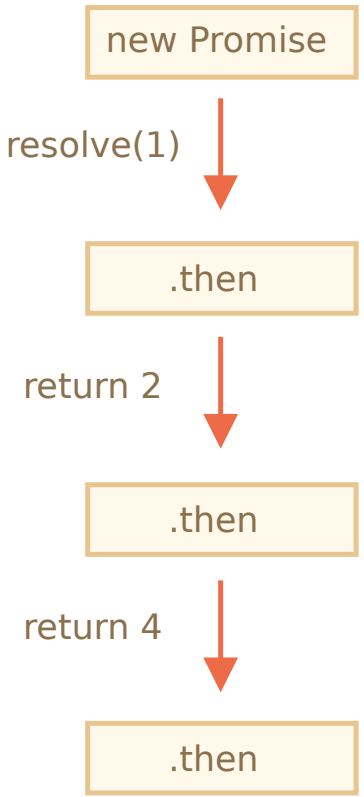
✔ Wykonanie kilka callbacków po kolei

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then(function(result) {
  alert(result); // 1
  return result * 2;
}).then(function(result) {
  alert(result); // 2
  return result * 2;
}).then(function(result) {
  alert(result); // 4
  return result * 2;
});
```

✔ Zobacz

Łańcuchy

✔ Wynik `promise.then()` jest Promise'em

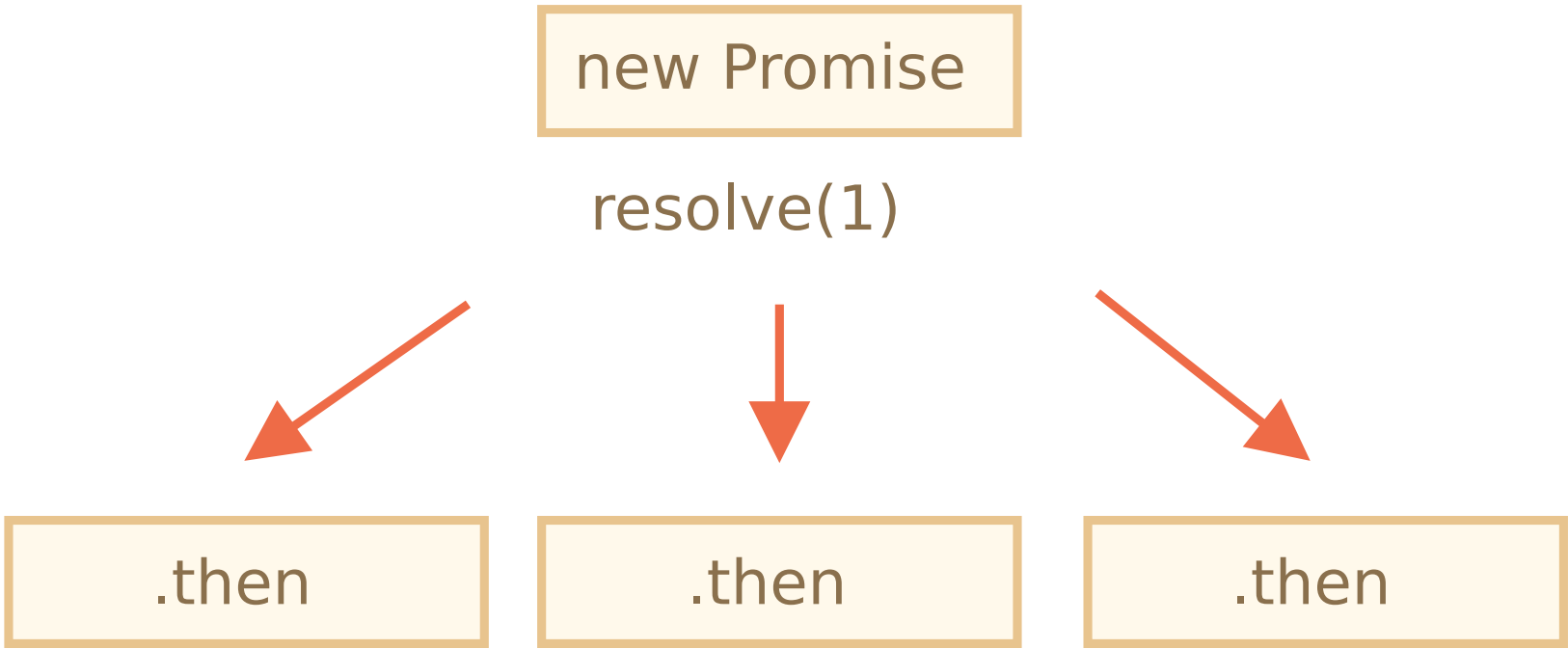


Jakie komunikaty zobaczymy?


```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1024), 1000);
});
promise.then(function(result) {
  alert(result); // 1
  return result / 2;
});
promise.then(function(result) {
  alert(result); // 1
  return result / 2;
});
promise.then(function(result) {
  alert(result); // 1
  return result / 2;
});
```

✔ Zobacz

Podpowiedź



Łańcuchy asynchroniczne

✔ Callback zwraca promise

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
}).then(function(result) {
  alert(result);
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  alert(result);
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  alert(result);
});
```

✔ Zobacz

Przykład z załadowaniem skryptów


```
loadScript("one.js")
  .then(function(script) {
    return loadScript("two.js");
  })
  .then(function(script) {
    return loadScript("three.js");
  })
  .then(function(script) {
    one();
    two();
    three();
  });
```

Przykład i funkcje strzałki


```
loadScript("one.js")
  .then( script => loadScript("two.js"))
  .then( script => loadScript("three.js"))
  .then( script => {
    one();
    two();
    three();
  });
```

Opracowanie błedów

- ✔ Jeżeli promise kończy się błędem (`rejected`), to aktywuje się najbliższy callback na `error` w łańcuchu

```
loadScript("one.js")
  .then( script => loadScript("two.js"))
  .then( script => loadScript("three.js"))
  .then( script => {
    one();
    two();
    three();
  })
  .catch(error => alert(error.message));
```

Niejawny `try...catch`

✔ Wyjątki są równoważne `reject`

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

⇕

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

Dotyczy również calbacków


```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  throw new Error("Błąd!");
}).catch(alert);
```

Dotyczy również błędów wykonania


```
new Promise((resolve, reject) => {
  resolve("ok");
}).then((result) => {
  blabla(); // nie ma tej funkcji
}).catch(alert);
// ReferenceError: blabla is not defined
```

Przerzucanie błędów

✔ Można opracować błąd i przekazać sterowanie na następny `.then`

```
new Promise((resolve, reject) => {
  throw new Error("Błąd!");
}).catch(function(error) {
  alert("Continue");
}).then(() => alert("Kolejny then"));
```

Inny przykład


```
// the execution: catch -> catch -> then
new Promise((resolve, reject) => {
  throw new Error("Błąd!");
}).catch(function(error) { // (*)
  if (error instanceof URIError) {
    // opracować błąd
  } else {
    alert("NIe udało się opracować");
    throw error; // w następny catch
  }
}).then(function() {
  /* nie będzie wykonane*/
}).catch(error => {
  alert(`Bład nieznany : ${error}`);
  // dalej wykonanie w zwykłym trybie
});
```

Błędy nieopracowane


```
new Promise(function() {
  noSuchFunction();
})
  .then(() => {
    // callbacki .then
  }); // brak .catch
```

✔ Generowany błąd globalny, zdarzenie `unhandledrejection`

✔ W przeglądarce można opracować:

```
window.addEventListener('unhandledrejection',
 function(event) {
  alert(event.promise);
  // obiekt Promise, który wygenerował błąd
  alert(event.reason);
  // odpowiedni obiekt Error
});
```

Czy zostanie opracowany błąd?


```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

✔ Zobacz

Promise.all

- ✔ Równolegle uruchomić kilka promise'ów

```
let promise = Promise.all([...promises...]);
```

- ✔ Przykład:

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

- ✔ Kolejność zgadza się z kolejnością na liście
- ✔ Zobacz

Promise.all, błędy

- ✔ W sytuacji `rejected` w jednym z listy promise'ów oblicznie kończy się natychmiast z błędem

```
Promise.all([
  new Promise(
   (resolve, reject) => setTimeout(
            () => resolve(1), 1000)),
  new Promise(
   (resolve, reject) => setTimeout(
            () => reject(new Error("Blad!")), 2000)),
  new Promise(
   (resolve, reject) => setTimeout(
            () => resolve(3), 3000))
]).catch(alert);
```

- ✔ pozostałe promise'y będą obliczane, ale wyniki zostaną ignorowane
- ✔ Zobacz

`Promise.all`, **zastosowanie**

- ✔ Dana jest lista danych
- ✔ Za pomocą funkcji `map` tworzymy listę promise'ów
- ✔ Odpalamy na niej `Promise.all`

```
let urls = [
  'https://api.github.com/users/adenisiuk',
  'https://api.github.com/users/ventrae',
  'https://api.github.com/users/darek'
];
let requests = urls.map(url => fetch(url));
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}:
                       ${response.status}`)
  ));
```

- ✔ Funkcja `fetch()` zwraca promise
- ✔ Zobacz

Promise.all, **przykład**

- ✔ Dana jest lista użytkowników github

```
let names = ['adenisiuk', 'ventrae', 'darek'];
let requests = names.map(
   name => fetch(`https://api.github.com/users/${name}`))
Promise.all(requests)
  .then(responses => {
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`);
    }
    return responses;
  })
  // zamienić response w response.json(),
  .then(
   responses => Promise.all(responses.map(r => r.json())))
  .then(users => users.forEach(user => alert(user.name)))
```

- ✔ Zobacz

Promise.allSettled

- ✔ Czeka na obliczenie wszystkich promise'ów
- ✔ Wynik — tablica struktur
  - ✘ `{status:"fulfilled", value:wynik}`
  - ✘ `{status:"rejected", reason:błąd}`

Promise.allSettled. Przykład


```
let urls = [
  'https://api.github.com/users/adenisiuk',
  'https://api.github.com/users/ventrae',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => {
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

Zobacz

Promise.race

✔ Wykorzystuje wynik (albo błąd) najszybciej obliczonego promise'a

```
Promise.race([
  new Promise(
   (resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise(
   (resolve, reject) => setTimeout(() =>
                        reject(new Error("Błąd!")), 2000)),
  new Promise(
   (resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

Promise.resolve(value)

✔ Tworzy wykonany promise z wynikiem `value`

```
et cache = new Map();

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

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

`Promise.reject(error)`

- ✔ Tworzy odrzucony promise z błędem `error`
- ✔ To samo, co

```
new Promise((resolve, reject) => reject(error));
```

Promisyfikacja

- ✔ Dana jest funkcja, która ma argument-callback
- ✔ Zamiana kodu, aby funkcja zracała promise
  - ✘ na promise'ie rejestrujemy callback
- ✔ Przykładowo

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Error ${src}`));

  document.head.append(script);
}

loadScript('path/script.js', (err, script) => {...})
```

Wersja promisyfikowana


```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}

loadScriptPromise('path/script.js').then(...)
```

„Promisyfikator”


```
function promisify(f) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, result) { // callback dla f
        if (err) {
          return reject(err);
        } else {
          resolve(result);
        }
      }
      args.push(callback);
      f.call(this, ...args);
    });
  };
};
// wykorzystanie:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

„Promisyfikator”, callback z różną ilością argumentów


```
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function callback(err, ...results) {
        if (err) {
          return reject(err);
        } else {
          resolve(manyArgs ? results : results[0]);
        }
      }
      args.push(callback);
      f.call(this, ...args);
    });
  };
};
// wykorzystanie:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...)
```

Kolejność zadań

✔ Który alert pojawi się jako pierwszy?

```
let promise = Promise.resolve();
promise.then(() => alert("promise"));
alert("skrypt");
```

✔ Zobacz

Kolejka zadań

- ✔ Po wykonaniu promise'a jego callbacki umieszcza się w kolejce mikrozadań (Job Queue, microtask queue)

```
promise.then(handler);

...

alert("code finished");
```

handler w kolejce

zakończono wykonanie skryptu

odpala się handler z kolejki

- ✔ Priorytety zadań:
  1. kod synchroniczny
  2. kolejka mikrozadań
  3. kolejka callbacków

Pytanie za sto punktów

✔ Jaka będzie kolejność alertów?

```
alert('Message no. 1');

setTimeout(function() {alert('Message no. 2');}, 0);

var promise = new Promise(function(resolve, reject) {
   resolve();
});
promise.then(function(resolve) {
   alert('Message no. 3');}
)
.then(function(resolve) {alert('Message no. 4');});

alert('Message no. 5');
```

✔ Zobacz

Nieopracowane błędy

- ✔ Zdarzenie `unhandledrejection` jest generowane w momencie gdy promis skończył się niepowodzeniem i kolejka mikrozadań jest pusta.
- ✔ Jakie komunikaty się pojawią?

```
let promise
   = Promise.reject(new Error("Błąd w promisie!"));

window.addEventListener( 'unhandledrejection',
   event => {alert(event.reason);} );
```

- ✔ Zobacz

Jaki komunikat się pojawi?


```
let promise
    = Promise.reject(new Error("Błąd w promisie!"));
promise.catch(err => alert('przechwycony!'));

window.addEventListener( 'unhandledrejection',
    event => {alert(event.reason);} );
```

✔ Zobacz

A teraz jaki komunikat się pojawi?


```
let promise
    = Promise.reject(new Error("Błąd w promisie!"));

setTimeout(
    () => promise.catch(err => alert('przechwycony')),
    1000);

window.addEventListener('unhandledrejection',
    event => alert(event.reason));
```

✔ Zobacz

Funkcje asynchroniczne

- ✔ `async function f` — przekształca wynik funkcji `f` w zakończony promise
- ✔ Przykładowo:

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

- ✔ równoważne

```
async function f() {
  return Promise.resolve(1);
}
```

await

- ✔ `let value = await promise;` — obliczenie się zatrzymuje i zostanie kontynuowane po zakończeniu promise'a
- ✔ JavaScript nie zostanie zablokowany, będzie obliczał inne zadania

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Done!"), 1000)
  });
  let result = await promise;
  alert(result);
}
f();
```

- ✔ Możliwe tylko w funkcji async

`await`, wyjątki

✔ Jeżeli promise zakończy się niepowodzeniem, zostanie wyrzucony wyjątek

```
async function f() {
  await Promise.reject(new Error("Woops!"));
}
```

✘ działa jak

```
async function f() {
  throw new Error("Woops!");
}
```

await, przechwytywanie wyjątków

✔ Można wyjątek przechwycić przez `try..catch`

```
async function f() {
 try {
  let response = await fetch('http://no-such-url');
 } catch(err) {
  alert(err);
 }
}
f();
```

✘ albo przez `catch` promise'a

```
async function f() {
  let response = await fetch('http://no-such-url');
}
f().catch(alert);
```

AJAX

Asynchroniczna wymiana danych z serwerem

- ✔ Asynchronous Javascript And XML — technologia wymiany danych z serwerem bez przeładowywania strony
- ✔ Czas interakcji jest zbliżony do desktopa
- ✔ Przykłady zastosowań:
  - ✘ Małe elementy sterujące (zgoda na użycie cookies, dodać do kosza, etc)
  - ✘ Dynamiczne ładowanie zawartości (drzewo, którego węzły się ładują w miarę potrzeby)
  - ✘ Ciągłe ładowanie informacji z serwera (czat, gełda, etc)
  - ✘ Google suggest, gmail

Podstawowe metody

- ✔ Funkcja `fetch()`
- ✔ Obiekt `XmlHttpRequest`
- ✔ Protokół Websocket

Funkcja fetch

Funkcja `fetch`

- ✔ Następca `XMLHttpRequest`
- ✔ Nie jest obsługiwany przez stare przeglądarki
  - ✘ dostępne są polyfilly
- ✔ Zwraca `Promise`
- ✔ Składnia:

```
let promise = fetch(url[, options]);
```

  - ✘ bez `options` pobiera dane metodą `GET` z podanego adresu

Status odpowiedzi

- ✔ Promise kończy się z obiektem typu Response
  - ✘ niepowodzenie, jeżeli nie udało się wykonać zapytania:
    - ✓ błąd sieciowy, nieprawidłowy adres serwera

    powodzenie: dwie właściwości:

    - ✓ `status` — kod odpowiedzi
    - ✓ `ok` — `true`, jeżeli żądanie http zakończono powodzeniem (kod w zakresie 200–299)

Treść odpowiedzi

- ✔ Metody obiektu Response, zwracają promise'y:
  - ✘ `response.text()` — zwykły tekst
  - ✘ `response.json()` — parsuje odpowiedź w formacie JSON
  - ✘ `response.formData()` — odpowiedź jako obiekt `formData`
  - ✘ `response.blob()` — odpowiedź jako obiekt `Blob` (dane binarne i typ)
  - ✘ `response.arrayBuffer()` — odpowiedź jako obiekt `ArrayBuffer` (dane binarne)
  - ✘ `response.body` — obiekt ReadableStream, można czytać odpowiedź częściami
- ✔ Można wybrać tylko jedną metodę

Przykład


```
fetch('user.json')
 .then(function(response) {
  if(response.ok) return response.json()
  else throw
   new Error("Netork Error: " + response.status);
 })
 .then(function(user) {
  addUser(user.name, user.surname);
 })
 .catch(error => alert(error));
```

✔ Zobacz:

- ✘ OK
- ✘ Not found
- ✘ Network error

Przykład z `await`

✔ Dwa etapy:

```
let response = await fetch(url, options);
let result = await response.json();
```

Nagłówki odpowiedzi

- ✔ W obiekcie `response.headers`
  - ✘ podobny do obiektu `Map`

```
let response = await fetch('http://szuflandia.pjwstk.edu
   ~denisjuk/tin/fetch/user.json');

alert(response.headers.get('Content-Type'));

for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

- ✔ Zobacz

Nagłówki żądania

✔ Opcja `headers` funkcji `fetch()`

```
let response = fetch(url, {
  headers: {
    Authentication: 'secret'
  }
});
```

✘ lista *zakazanych* nagłówków

Żądania POST i inne

- ✔ Parametry `method` oraz `body` funkcji `fetch()`
  - ✘ `method` — metoda http, przykładowo `POST`
  - ✘ `body` — treść żądania:
    - ✓ tekst (na przykład, JSON)
    - ✓ obiekt `formData` — do wysyłania danych w formacie`form/multipart`
    - ✓ `Blob`/`BufferSource` — do wysyłania danych binarnych
    - ✓ `URLSearchParams` — do wysyłania danych w formacie`x-www-form-urlencoded`

Wysyłanie danych w formacie JSON


```
let user = {
   name: 'Aleksander',
   surname: 'Denisiuk'
};
let response = await fetch('user.php', {
   method: 'POST',
   headers: {
      'Content-Type': 'application/json;charset=utf-8'
},
   body: JSON.stringify(user)
});
let result = await response.text();
   alert(result);
```

✔ Zobacz

Skrypt user.php


```php
<?php

$json = file_get_contents('php://input');

// Converts it into a PHP object
$data = json_decode($json, true);

echo <<<OEF
imię: $data[name]
nazwisko: $data[surname]
well done ;-)
OEF;

?>
```

Wysyłanie danych binarnych

✔ Element canvas

```
<canvas id="canvasElem" width="100"
   height="80" style="border:1px solid"></canvas>

<input type="button"
   value="Wyślij" onclick="submit()">
```

Skrypt

```
canvasElem.onmousemove = function(e) {
   let ctx = canvasElem.getContext('2d');
   ctx.lineTo(e.clientX, e.clientY);
   ctx.stroke();
};
async function submit() {
 let blob = await new Promise(
   resolve => canvasElem.toBlob(resolve, 'image/png'));
 let response = await fetch('image.php',
      {method: 'POST', body: blob } );
 let result = await response.text();
 alert(result);
}
```

✔ Zobacz

image.php


```php
<?php
    $my_blob = file_get_contents('php://input');
    file_put_contents('image.png', $my_blob);
    echo "zapisane do image.png";
?>
```

Wysyłanie formularzy

Obiekt `FormData`

- ✔ Formularz html
- ✔ Wysyłany funkcją `fetch()` z nagłówkiem `Content-Type: form/multipart`
  - ✘ dla serwera: zwykły formularz html
- ✔ Konstruktor:

```
let formData = new FormData([form]);
```

  - ✘ Jeżeli podać element `form`, to obiekt automatycznie odczyta wartości pól formularza

Przykład


```
formElem.onsubmit = async (e) => {
 e.preventDefault();
 let response = await fetch('user.php', {
      method: 'POST',
      body: new FormData(formElem)
 });
 let result = await response.text();
 alert(result);
};
```

✔ Zobacz

user.php


```php
<?php
    echo <<<OEF
    imię: $_POST[name]
    nazwisko: $_POST[surname]
    well done ;-)
    OEF;
?>
```

Metody obiektu `FormData`

- ✔ `formData.append(name, value)` — dodaje pole `name` o wartości `value`
- ✔ `formData.append(name, blob, fileName)` — dodaje pole odpowiadające elementowi `<input type="file">`, `fileName` ustawia nazwę pliku
- ✔ `formData.delete(name)` — usuwa pole `name`
- ✔ `formData.get(name)` — wartość pola `name`
- ✔ `formData.has(name)` — `true` / `false`
- ✔ `formData.set(name, value)`, `formData.set(name, blob, fileName)` — podobno do append, ale przed dodaniem usuwa się wszystkie pola `name`
- ✔ `for(let [name, value] of formData) {...}` — pętla po polach

Formularz


```
<form id="formElem">
  <input type="text" name="firstName"
                              value="Aleksander">
  Avatar: <input type="file" name="picture"
                              accept="image/*">
  <input type="submit">
</form>
```

Skrypt

```
formElem.onsubmit = async (e) => {
  e.preventDefault();

  let response = await fetch('avatar.php', {
    method: 'POST',
    body: new FormData(formElem)
  });

  let result = await response.text();

  alert(result);
};
```

✔ Zobacz

avatar.php


```php
<?php
$filename = $_FILES['picture']['name'];
$filesize = $_FILES['picture']['size'];
echo <<<OEF
imię: $_POST[firstName]
avatar: $filename
rozmiar: $filesize bajtów
well done ;-)
OEF;
?>
```

Canvas


```
<canvas id="canvasElem" width="100" height="80"
   style="border:1px solid"></canvas>
<input type="button" value="Wyślij"
                     onclick="submit()">
<script>
   canvasElem.onmousemove = function(e) {
   let ctx = canvasElem.getContext('2d');
   ctx.lineTo(e.clientX, e.clientY);
   ctx.stroke();
   };
</script>
```

Skrypt

```
async function submit() {
 let imageBlob = await new Promise(
   resolve => canvasElem.toBlob(resolve, 'image/png'));
 let formData = new FormData();
 formData.append("firstName", "Aleksander");
 formData.append("image", imageBlob, "image.png");
 let response = await fetch('image.php', {
   method: 'POST',
   body: formData
 });
 let result = await response.text();
 alert(result);
}
```

✔ Zobacz

image.php

```php
<?php
    $filename = $_FILES['image']['name'];
    $filesize = $_FILES['image']['size'];
echo <<<OEF
imię: $_POST[firstName]
avatar: $filename
rozmiar: $filesize bajtów
well done ;-)
OEF;
?>
```

Cross-Origin Resource Sharing

Same origin policy

- ✔ *Zazwyczaj* nie można wysłać zapytania fetch na inny serwer
  - ✘ Same-origin policy (reguła tego samego pochodzenia)
    - ✓ domena/port/protokół

```
fetch('http://example.com').catch(alert);
```

- ✔ Zobacz

Obejście same origin policy

- ✔ Formularze: zapytania POST i GET

```
<iframe name="iframe"></iframe>
<form target="iframe" method="POST"
          action="http://another.com/...">
  ...
</form>
```

  - ✘ problem z odczytywaniem danych z `iframe`: do pokonania

- ✔ Skrypty: można załadować i wykonać skrypt z dowolnego miejsca:

```
<script src="http://another.com/..."></script>
```

  - ✘ JSONP (JSON with Padding)

JSON with Padding

- ✔ Deklarujemy callback do opracowania danych, przykładowo `function f(x, y){...}`
- ✔ Tworzymy skrypt

```
let script = document.createElement('script');
script.src = `http://another.com/api?parametry`;
document.body.append(script);
```

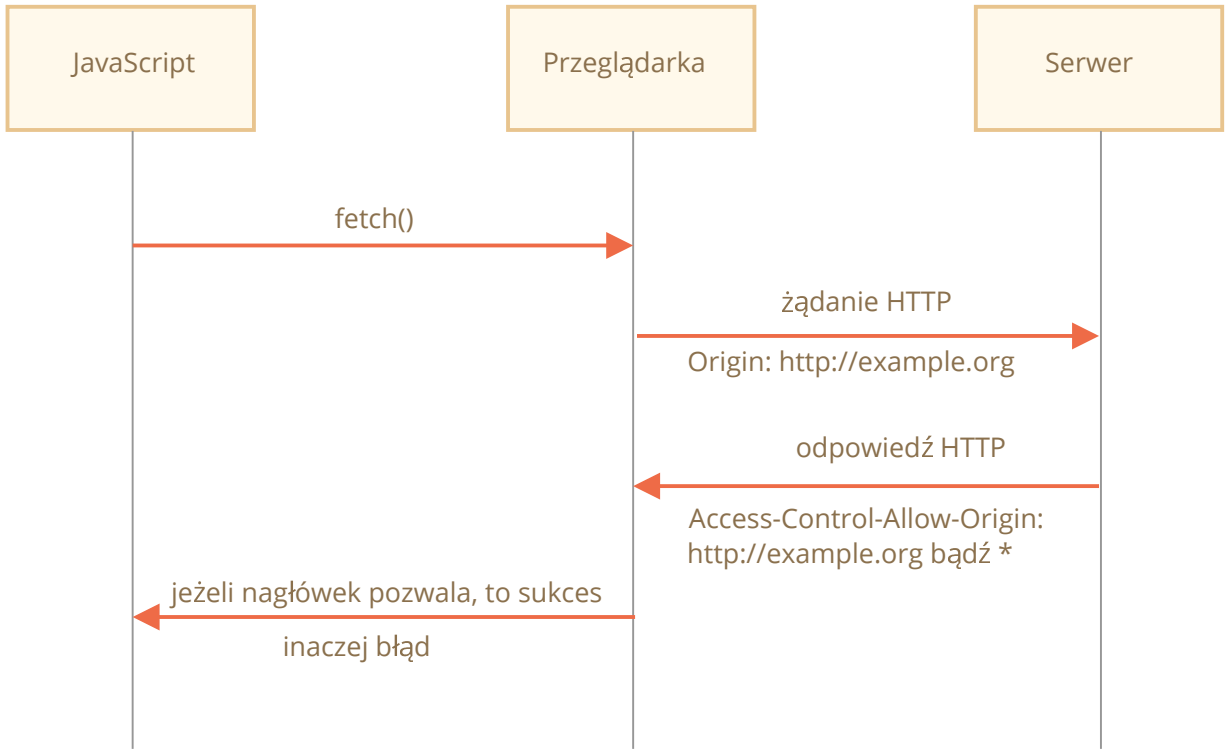
- ✔ Serwer zwraca tekst `"f(x, y)"`, zawierający wartości przekazywanych parametrów `(x, y)`
- ✔ Na kliencie odpala się `f(x, y)`
- ✘ w parametrach można przekazać również callback
- ✔ Zobacz przykład

Zapytanie proste (Simple Request)

- ✔ Metoda http: `GET`, `POST` lub `HEAD`
- ✔ Nagłówki http: `Accept`, `Accept-Language`, `Content-Language` lub `Content-Type`
  - ✘ ostatni wyłącznie `application/x-www-form-urlencoded`, `multipart/form-data` lub `text/plain`
- ✔ I tak można zrobić za pomocą `<form>` bądź `<script>`

Schemat zapytania prostego

- ✔ Przeglądarka ustawia nagłówek `Origin`
- ✔ Serwer ustawia nagłówek `Access-Control-Allow-Origin`
  - ✘ `*` oznacza każdy serwer
- ✔ Przeglądarka sprawdza, czy dostęp jest dozwolony



Nagłówki odpowiedzi na zapytanie proste

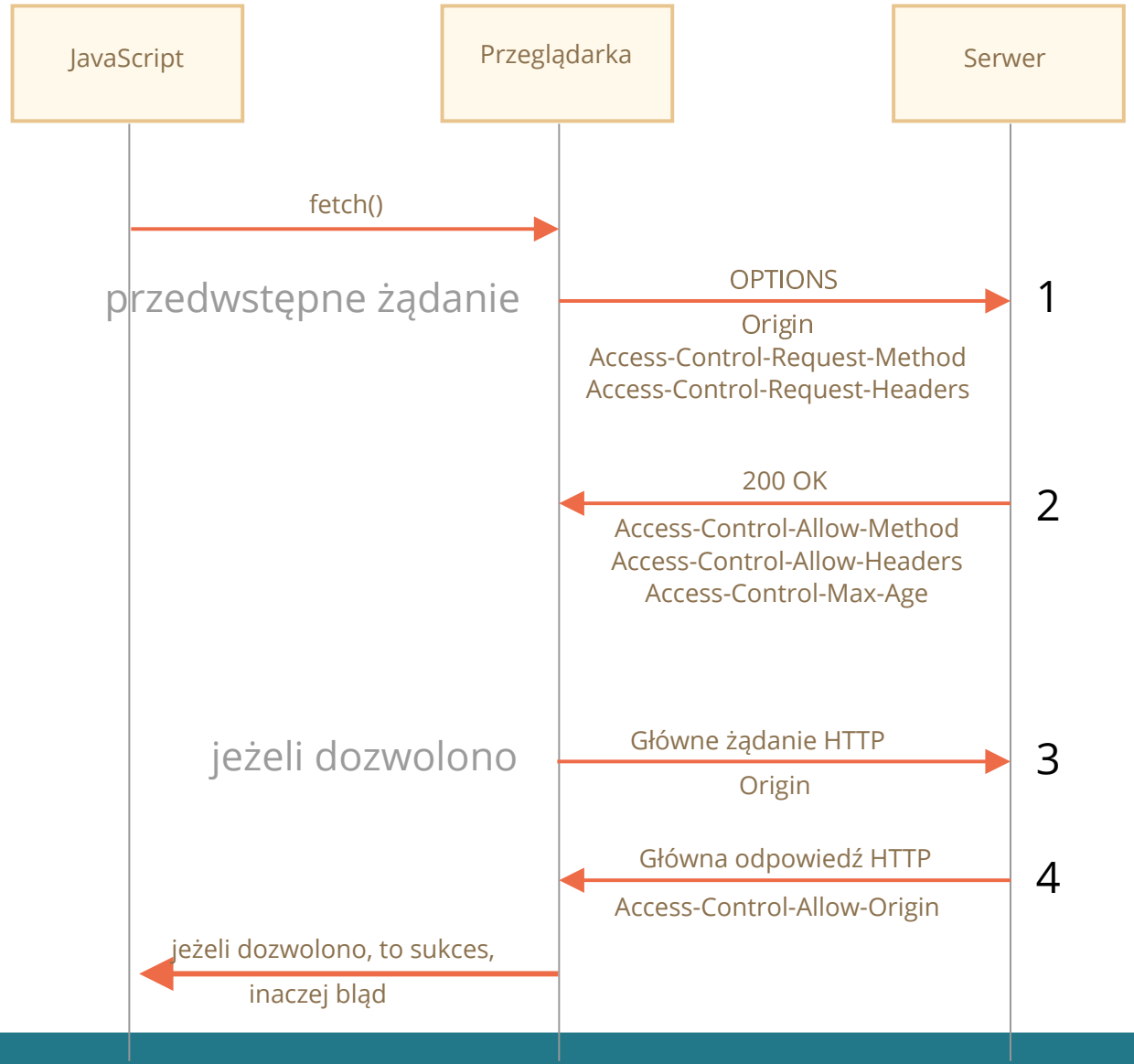
- ✔ Skrypt ma dostęp do nagłówków odpowiedzi: `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified` oraz `Pragma`
- ✔ Aby dozwolić dostęp do innych nagłówków, serwer powinien ustawić nagłówek `Access-Control-Expose-Headers`, przykładowo:

```
200 OK
Content-Type:text/html; charset=UTF-8
Content-Length: 12345
API-Key: 2c9de507f2c54aa1
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: Content-Length,API-Key
```

Zapytanie nie aż takie proste (Not-So-Simple Requests)

- ✔ Dowolna metoda http, dowolne nagłówki
- ✔ Dwa etapy:
  1. Żądanie przedwstępne
     - ✘ klient ustawia nagłówki http: `Access-Control-Request-Method`, `Access-Control-Request-Headers`
       - ✓ wykorzystywana metoda `OPTIONS`
     - ✘ serwer odpowiada z kodem 200 i nagłówkami http: `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`
       - ✓ opcjonalnie dodaje `Access-Control-Max-Age`
  2. Żądanie główne

Schemat zapytania nieprostego



Przykład zapytania nieprostego


```
let response = await
      fetch('https://site.com/service.json', {
  method: 'PATCH',
  headers: {
    'Content-Type': 'application/json'
    'API-Key': 'secret'
  }
});
```

✔ Szczegóły zapytania są *za kulisami*

Żądanie przedwstępne


```
OPTIONS /service.json
Host: site.com
Origin: https://javascript.info
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type,API-Key
```

Odpowiedź przedwstępna


```
200 OK
Access-Control-Allow-Methods: PUT,PATCH,DELETE
Access-Control-Allow-Headers: API-Key,Content-Type,
        If-Modified-Since,Cache-Control
Access-Control-Max-Age: 86400
```

- ✔ 86400 sekund (jeden dzień)
- ✔ łamanie wiersza tylko dla potrzeb prezentacji

Żądanie główne


```
PATCH /service.json
Host: site.com
Content-Type: application/json
API-Key: secret
Origin: https://example.com

.....................................
```

Odpowiedź główna


```
.....................................
Access-Control-Allow-Origin: https://example.com

.....................................
```

Dane autoryzacyjne

- ✔ Domyślnie przeglądarka nie wysyła danych autoryzacyjnych (np. Cookies)
- ✔ Dodać w sposób jawny:

```
fetch('http://another.com', {
  credentials: "include"
});
```

Serwer odpowiada w sposób jawny

```
200 OK
Access-Control-Allow-Origin: https://examle.com
Access-Control-Allow-Credentials: true
```

✘ w `Access-Control-Allow-Origin` nie może być `*`

Fetch API

API

- ✔ Funkcja `fetch()` ma inne możliwości
  - ✘ indykacja postępu otrzymywania odpowiedzi
  - ✘ przerywanie zapytania
  - ✘ etc

Obiekty URL

Konstruktor

- ✔ Można przekazać jako parametr do większości metod
- ✔ `new URL(url, [base])`
  - ✘ `url` — adres (bądź adres względny)
  - ✘ `base` — adres bazowy

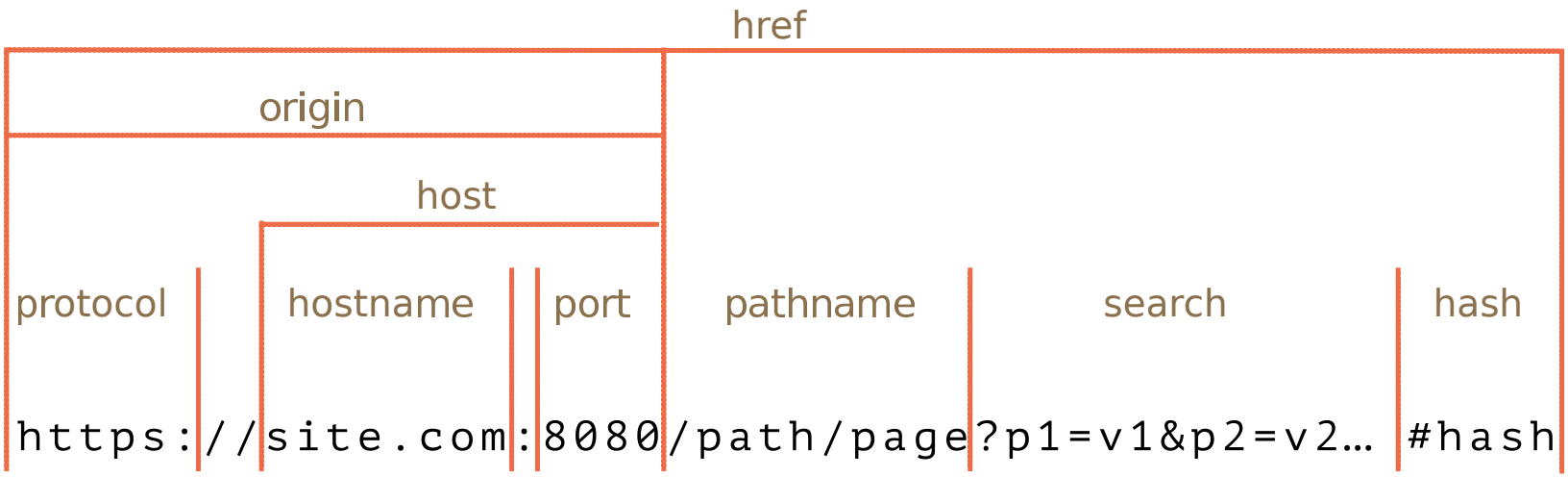
```
let url1 = new
   URL('http://szuflandia.pjwstk.edu.pl/~denisjuk/tin');
let url2 = new
   URL('tin', 'http://szuflandia.pjwstk.edu.pl/~denisjuk/');
```

- ✔ Względem innego obiektu:

```
let url3 = new URL('gk', url1);
```

Składowe obiektu URL

- ✔ `protocol`, `host`, etc
- ✔ Dostępne do odczytania oraz zmiany



- ✔ Dodatkowo `user` oraz `password` do uwierzytelniania http:
  `http://user:password@site.com`

Właściwość `searchParams`

- ✔ Obiekt typu `URLSearchParams`
- ✔ Metody:
  - ✘ `append(name, value)`
  - ✘ `delete(name)`
  - ✘ `get(name)`
  - ✘ `getAll(name)`
  - ✘ `has(name)`
  - ✘ `set(name, value)`
  - ✘ `sort()` — sortowanie według `name`
  - ✘ jest iterowalnym podobno do `Map`

Kodowanie znaków specjalnych

✔ `searchParams` automatycznie koduje parametry zgodnie z RFC3986

```
let url = new URL('https://www.google.com/search');
url.searchParams.set('q', 'Elbląg');
alert(url);
//https://www.google.com/search?q=Elbl%C4%85g
```

✘ funkcja `encodeURI` jest zgodnia ze starym RFC2396

```
let url = 'http://[2607:f8b0:4005:802::1007]/';
alert(encodeURI(url));
// http://%5B2607:f8b0:4005:802::1007%5D/
alert(new URL(url));
// http://[2607:f8b0:4005:802::1007]/
```

XMLHttpRequest

Schemat

- ✔ tworzymy obiekt `XmlHttpRequest`
- ✔ rejestrujemy funkcję na zradzenie `readystatechange` (odpowiedź serwera)
- ✔ otwieramy połączenie z serwerem — `open`
- ✔ wysyłamy zapytanie `send`
- ✔ odpowiedź serwera jest przyjmowana asynchronicznie funkcją, reagującą na `readystatechange`

Konstruktor, inicalizacja, wysyłanie zapytania

- ✔ Konstruktor nie ma parametrów

```
let xhr = new XMLHttpRequest();
```

- ✔ Inicjalizacja

```
xhr.open(method, URL, [async, user, password])
```

  - ✘ `method` — `GET`, `POST`, etc
  - ✘ `URL` — adres, tekst bądź obiekt `URL`
  - ✘ `async` — jeżeli `true`, zapytanie zostanie wysłane synchronicznie
  - ✘ `user`, `password` — dane do uwierzytelniania HTTP Basic

- ✔ Wysyłanie zapytania

```
xhr.send([body])
```

  - ✘ `body` — wysyłane dane

Odpowiedź

- ✔ Callback na zdarzenia
  - ✘ `load` — otrzymana odpowiedź (możliwe, że z kodem błędu HTTP)
  - ✘ `error` — bład sieciowy
  - ✘ `progress` — okresowo podczas załadowania odpowiedzi
  - ✘ `timeout` — zapytanie nie udało się wykonać w określonym czasie

Przykład


```
xhr.onload = function() {
  alert(`Otzrymano: ${xhr.status} ${xhr.response}`);
};
xhr.onerror = function() {
  alert(`Błąd połączenia`);
};
xhr.onprogress = function(event) {
  // event.loaded - ilość otrzymanych bajtów
  // event.lengthComputable - true, jeżeli serwer
  //     przysłał nagłówek Content-Length
  // event.total - rozmiar odpowiedz
  //     (jeżeli lengthComputable == true)
  alert(`Otrzymano ${event.loaded} z ${event.total}`);
};
```

Status odpowiedzi

- ✔ `xhr.status` — kod odpowiedzi
  - ✘ 200 powodzenie
  - ✘ 404 zasób nie znaleziony
- ✔ `xhr.statusText` — opis słowny kodu odpowiedzi
  - ✘ OK
  - ✘ File not found
- ✔ `xhr.response` — dane odpowiedzi

Typ odpowiedzi

- ✔ Ustawia się w `xhr.responseType`
  - ✘ `""` — tekst (domyślnie)
  - ✘ `"text"`
  - ✘ `"ArrayBuffer"` — *surowe* dane binarne
  - ✘ `"Blob"` — dane binarne z typem
  - ✘ `"document"` — dokument XML (można korzystać z XPath)
  - ✘ `"json"` — automatycznie parsowany

Przykład


```
let xhr = new XMLHttpRequest();
xhr.open('GET', 'hello.php');
xhr.responseType = 'json';
xhr.send();
xhr.onload = function() {
  let responseObj = xhr.response;
  alert(responseObj.message);
};
```

✔ Zobacz

Ponadto

- ✔ Stan zapytania
- ✔ Indykacja postępu otrzymywania/wysyłania odpowiedzi
- ✔ Przerywanie zapytania
- ✔ Zapytania synchroniczne
- ✔ Obsługa nagłówków
- ✔ CORS
- ✔ etc

Long Polling Chat

Na kliencie

1. subskrypcja — wysyłamy zapytanie i czekamy na odpowiedź (komunikat)
2. w razie przerwania połączenia subskrybujemy ponownie
3. przy otrzymaniu komunikatu wyświetlamy i subskrybujemy ponownie
4. wysyłamy swój komunikat

Na serwerze

1. przy subskrypcji dodajemy klienta do listy i nic nie odpowiadamy
2. przy zerwaniu połączenia usuwamy klienta z listy
3. przy otrzymaniu komunikatu wysyłamy wszystkim klientom i usuwamy ich z listy

HTML


```
<!DOCTYPE html>
<html><head>
  <meta charset="utf-8">
</head><body>
<p class="lead">Witaj w chacie!</p>

<form id="publish" >
  <input type="text" name="message"/>
  <input type="submit" value="Wyślij"/>
</form>

<div id="subscribe">
</div>
<script>...</script>
</body>
</html>
```

Subskrypcja


```
async function subscribe() {
   let response = await fetch(url);
   if (response.status == 502) {//timeout
   await subscribe();
   } else if (response.status != 200) {
   showMessage(response.statusText);
   await new Promise(
      resolve => setTimeout(resolve, 1000));
   await subscribe();
   } else {
   let message = await response.text();
   showMessage(message);
   await subscribe();
   }
}
subscribe();
```

Wysłanie komunikatu


```
function sendMessage(message) {
    fetch(url, {
        method: 'POST',
        body: message
    });
}

form.onsubmit = function() {
    let message = form.message.value;
    if (message) {
        form.message.value = '';
        sendMessage(message);
    }
    return false;
};
```

Subskrypcja


```
function onSubscribe(req, res) {
  let id = Math.random();

  res.setHeader('Content-Type',
                'text/plain;charset=utf-8');
  res.setHeader("Cache-Control",
                "no-cache, must-revalidate");
  subscribers[id] = res;
  req.on('close', function() {
    delete subscribers[id];
  });

}
```

Publikacja


```
.............................
if (urlParsed.pathname == '/publish'
        && req.method == 'POST') {
    req.setEncoding('utf8');
    let message = '';
    req.on('data', function(chunk) {
        message += chunk;
    }).on('end', function() {
        publish(message);
        res.end("ok");
    });

.............................
```

Funkcja `publish()`


```
function publish(message) {
  for (let id in subscribers) {
    let res = subscribers[id];
    res.end(message);
  }
  subscribers = Object.create(null);
}
```