AJAX

Hello

Kamil Richert

Junior Software Engineer at Spartez

Gdańsk, 27 października 2019 roku

AGENDA

1. HTTP
2. ASYNCHRONICZNOŚĆ
3. CALLBACK HELL
4. PROMISE
5. FETCH
6. ASYNC / AWAIT
7. EVENT LOOP

HTTP

HyperText Transfer Protocol - protokół przesyłania dokumentów hipertekstowych w sieci WWW

Zadaniem stron WWW jest publikowanie informacji
- natomiast protokół HTTP to umożliwia

HTTP

Kiedy to się w ogóle zaczeło?

Pierwsza udokumentowana wersja (0.9) pochodzi z 1991 roku.
Wersja 1.0 pochodzi z 1996.
Wersja 1.1 pochodzi z 1997.
Powstawały kolejne modyfikacje aż do 2014 roku.
HTTP/2 został zatwierdzony w 2015.

HTTP

URL

Źródło: http://antezeta.com/news/campaign-tracking

Z czego się składa HTTP?

Metoda

1. GET – żądanie pobrania zasobu
2. POST – żądanie utworzenia zasobu
3. PUT – żądanie zmiany / utworzenia zasobu
4. DELETE - żądanie usunięcia zasobu
5. PATCH – żądanie zmiany części zasobu
6. OPTIONS – żądanie sprawdzenia dostępnych opcji
7. HEAD – żądanie pobrania nagłówków dla metody GET

Z czego się składa HTTP?

Request headers (nagłówki)

- Accept
- Accept-Langaguge
- Host
- User-Agent
- Cookie

Z czego się składa HTTP?

Response headers (nagłówki)

- Expires
- Content-Type
- Set-Cookie
- Content-Length
- Cache-Control

Z czego się składa HTTP?

Status

- 1xx – informacyjne
- 2xx – status
- 3xx – przekierowanie
- 4xx – błąd po stronie klienta
- 5xx – błąd po stronie serwera

Zobaczmy to w akcji

curl – popularny klient HTTP

wpisz komendę do konsoli:

`curl --verbose http://google.pl`

Podsumowanie HTTP

- Protokół HTTP jest bezstanowy
  każdy request musi zawierać tyle informacji ile to konieczne, aby odtworzyć stan aplikacji
- Protokół HTTP to request i response
  Request – nagłówki, dane
  Respone – nagłówki, status, dane
- Ciasteczka umożliwaiają wprowadzenie „stanowości”
- HTTP jest tekstowy, HTTPS pozwala na szyfrowanie danych między klientem a serwerem

AJAX

Asynchronous JavaScript and XML -

- technika tworzenia aplikacji internetowych
- bez przeładowywania całego dokumentu
- bardziej dynamiczna interakcja z użytkownikiem niż w tradycyjnym modelu
- każde żądanie nowych danych nie wiąże się z przeładowaniem całej strony

Synchroniczność

JavaScript jest sam w sobie synchroniczny. Nie możemy w nim pisać kodu asynchronicznego.

Jednak możemy korzystać z API, które dostarczają nam np. przeglądarki (setTimeout, setInterval czy XMLHttpRequest).

```
for (var i = 0; i < 10000; i++) {
    console.log('wykonam sobie 10000 razy logowanie')
}

console.log('wywołam się dopiero po 10000 logach')
```

Asynchroniczość

Dzięki asynchroniczności, możemy wykonywać różne procesy w tle, podczas gdy nasz program robi co innego.

Poznaliśmy już metody, które są asynchroniczne: setTimeout, setInterval czy XMLHttpRequest.

```
setTimeout(function() {
    for (var i = 0; i < 10000; i++) {
        console.log('wykonam sobie 10000 razy logowanie')
    }
}, 1000);

console.log('nie muszę czekać na wywołanie 10000 logów')
```

Synchroniczność vs Asynchroniczność

Stara szkoła

Pobieranie danych z XMLHttpRequest.

Metoda open przyjmuje 3 argumenty:

1. Metodę
2. URL
3. Boolean (czy zapytanie ma być asychroniczne)

```
var req = new XMLHttpRequest();
req.open('GET', 'https://randomuser.me/api/', false);
req.send(null);
if(req.status == 200) {
    console.log(req.responseText)
}
```

```
var req = new XMLHttpRequest();
req.open('GET', 'https://randomuser.me/api/', true);
req.onreadystatechange = function (aEvt) {
  if (req.readyState == 4) {
     if(req.status == 200)
      console.log(req.responseText);
     else
      console.log("Błąd podczas ładowania strony\n");
  }
};
req.send(null);
```

Callback hell

Pobieranie paru zasobów lub odczytywanie plików w kolejce z użyciem starej szkoły często kończyły się czymś co nazwano callback hell.

```
queryDatabase({ username: 'Arfat'}, (err, user) => {
  // handle errors database querying failure
  const image_url = user.profile_img_url;
  getImageByURL(`someServer.com/q=${image_url}`, (err, image) => {
    // handle errors fetching failure
    transformImage(image, (err, transformedImage) => {
      // handle errors transformation failure
      sendEmail(user.email, (err) => {
        // handle errors of email failure
        logTaskInFile('transformed the file and sent user an email', (err) => {
          // handle errors of logging failure
        })
      })
    })
  })
})
```

AJAX

Jak skorzystać z AJAX i pobrać dane?

Na przykład przy pomocy bilboteki jQuery lub bardziej nowoczesnego rozwiązania z Fetch API.

```
const fetchData = () => {
    return $.ajax({
        url: 'https://randomuser.me/api/'
    });
}

fetchData().then(data => console.log(data))
```

```
const fetchData = () => {
    return fetch('https://randomuser.me/api/')
}

fetchData().then(r => console.log(r))
```

Promise

Co to ten then?

then można wywołać na Promise

Promise to obiekt, który reprezentuje wartość, która może zostać wykorzystana w przyszłości. Dzięki niemu możemy przechwycić wynik operacji asynchroniczej i obsłużyć ją w pożądany sposób.

```
▼ Promise {<pending>} i
  ▼ __proto__: Promise
    ▶ catch: ƒ catch()
    ▶ constructor: ƒ Promise()
    ▶ finally: ƒ finally()
    ▶ then: ƒ then()
      Symbol(Symbol.toStringTag): "Promise"
    ▶ __proto__: Object
  [[PromiseStatus]]: "resolved"
  ▶ [[PromiseValue]]: Response
```

Promise vs callback

Promise rozwiązują wiele problemów - przedewszytkim pozbywa się problemu callback hell.

```
queryDatabase({ username: 'Arfat'})
  .then((user) => {
    const image_url = user.profile_img_url;
    return getImageByURL(`someServer.com/q=${image_url}`)
      .then(image => transformImage(image))
      .then(() => sendEmail(user.email))
  })
  .then(() => logTaskInFile('...'))
  .catch(() => handleErrors()) // handle all errors
```

Promise

Stany:

- **resolved (fulfilled)** – promise został spełniony, reprezentowana wartość jest dostępna
- **rejected** – promise odrzucony, np.. przez wyjątek, błąd serwera
- **pending** – oczekujący, pozstaje w takim stanie do momentu spełnienia lub odrzucenia

Promise

Promise możemy stworzyć z pomocą słówka **new**:

new Promise (handler)

Konstruktor Promise’a przyjmuje funkcję handler, której zadaniem jest stwierzdzenie czy dany Promise został spełniony czy odrzucony. Handler ma 2 argument – metodą resolve i reject.

Promise

Przykładowe uwtorzenie Promise’a:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(function(){
    resolve("Success!"); // Yay! Everything went well!
  }, 250);

  setTimeout(function(){
    reject("Failure!"); // Ups! Something went wrong!
  }, 500);
});

promise
    .then((successMessage) => {
        console.log("Yay! " + successMessage);
    })
    .catch((failureMessage) => {
        console.log("Ups! " + failureMessage);
    });
```

Fetch API

Wróćmy na chwile do fetch. Jako drugi argument metoda fetch przyjmuje obiekt opcji i zwraca nam Response. Jeżeli chcemy otrzymać dane w formacie czytelnym dla JavaScript musimy wykonać na odpowiedzi wykonać metodę *json()*.

```
fetch("...", {
        method: "post",
        body: "name=Marcin&surname=Nowak"
    })
    .then(res => res.json())
    .then(res => {
        console.log("Dodałem użytkownika:");
        console.log(res);
    });
```

Promise

Promise prócz then, catch, resolve i reject udostępniam nam jeszcze inne metody:

1) all – czekamy aż wszystkie promisy zostaną spełnione
2) race – rozwiązuj się, gdy pierwszy promise zostanie spełniony
3) finally – wykonane na samym końcu, nie przyjmuje parametru

```
var promises = [];

for (var i = 0; i < 10; i++) {
  promises.push(new Promise(function(resolve, reject) {
    var timeout = i * 10;
    setTimeout(function() {
      resolve('resolving after ' + timeout + ' milliseconds');
    }, timeout);
  }));
}

Promise.race(promises).then(function(result) {
  console.log('first result: ' + result)
});

Promise.all(promises).then(function(result) {
  console.log('first result: ' + result)
});

Promise.resolve('Rozwiązany').then(function(result) {
  console.log(result)
});

Promise.reject('Odrzucony').catch(function(result) {
  console.log(result)
});

Promise.resolve('Rozwiazny').finally(function(result) {
  console.log(result) // undefined
});
```

Async / await

Jednak developerów męczyło ciągłe then, then … Na ratunek powstało async / await, które pozwala pisać kod asynchroniczny w sposób bardziej synchroniczny.

```
function foo() {
    return new Promise((resolve, reject) => {
        resolve(1);
    });
}

foo().then(console.log); // 1
```

LUB:

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

foo().then(console.log); // 1
```

```
async function bar() {
    return 1; // „resolwuj” wartość
}

bar().then(console.log); // 1
```

Async / await

Zasady:

- await może wsytąpić tylko w funkcji oznacznej async
- wartość zwrócona w async funkcji oznacza tyle co Promise.resolve danej wartości
- async funkcja zwraca Promise
- brak oznaczenia funckji jako async spowoduje error w await

```
async function getUsers() {
    const response = await fetch('https://randomuser.me/api/');
    const users = await response.json();
    console.log(users); // tablica użytkowników
    return users;
}

const users = await getUsers();
```

Async / await

Ale co z errorami? Możemy użyć starego dobrego try catch!

```
async function getUsers() {
    try {
        let response = await fetch('https://randomuser.me/api/');
        let users = await response.json();
        return users;
    } catch(err) { // tutaj przechwytujemy błąd z Promise fetch
        alert(`ALERT MESSAGE ${err}`); // TypeError: failed to fetch
    }
}
```

Async / await

Tworzenie promiesów też jest prostsze!

```
const checkDay = day => {
    return new Promise((resolve, reject) => {
        if (day === 'Sunday') {
            resolve({time: '10:20:30'})
        } else {
            reject({message: 'Error!'})
        }
    });
};

async function checkDayAsync(day) {
    if (day === 'Sunday') {
        return {time: '10:20:30'};
    } else {
        throw {message: 'Error!'};
    }
}
```

Event loop

Kolejka wywołania (task queue) + stos wywołania (call stack):

- makrotaski (setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering)
- mikrotaski (process.nextTick, Promises, Object.observe, MutationObserver)

Event loop – w pigułce

1. Jeśli coś znajduje się w kolejce zadań to weź pierwszy element i wrzuć go na stos wywołań.
2. Wykonuj kolejne instrukcje ze stosu wywołań, do momentu aż będzie on pusty.
3. Wykonuj kolejne instrukcje z kolejki mikrozadań, do momentu aż będzie ona pusta.
4. Wykonaj instrukcje wskazane przez requestAnimationFrame, przelicz style, wyrenderuj stronę.

Dzieki

Znajdziecie mnie:

https://www.linkedin.com/in/kamil-richert/

https://github.com/krichert