JavaScript

Jak ogólnie wygląda nasze środowisko pracy?

Jak działa JS ?

- JavaScript to język interpretowany, to znaczy że kod nie ulega kompilacji. Jest cały czas widoczny w tej samej postaci – którą dostarczymy „na produkcję”
- Interpreter za każdym razem przetwarza kod i zachowuje się zgodnie z jego poleceniami
- W JS – nie ma wątków, to jednowątkowy tzw. event-loop
- JS oryginalnie był dedykowany tylko dla przeglądarki i jego przeznaczeniem w momencie powstawania było zrobienie „Dynamicznego HTMLa” – ożywienie stron WWW. Animacje, eventy etc...

Kontrowersyjny

- Tak, i to bardzo – zwłaszcza jeśli porównujemy go z językami wysokopoziomowymi typu: C++, C#, Java
- Jest podobny do Javy – może nie tylko z nazwy. Jednak pozwala na masę swobody i przez to jest podatny na pisanie – masę bezużytecznego, brudnego, mało czytelnego kodu; Tak, można go w JS popełnić ☺
- Jednak co takiego kontrowersyjnego w JavaScript – wywołuję burze przy porównaniu go z innymi językami?

Typowanie i jego brak

- JavaScript nie określa „jawnie” jakiego typu są dane. To znaczy – nie stosuje się tam tzw. twardego typowania
- Przykładowo w innych językach posiadamy możliwość określenia jakiego typu jest dana zmienna:

```
String hello = "Hello Michał";
```

Kod w Java:
Deklarowanie, że zmienna hello jest typu „ciągu znaków” - String

Ten sam kod – w JavaScript wygląda tak:

```
let hello = "Hello Michał"
```

- Słowo kluczowe **let** wskazuje jedynie – że mamy do czynienia ze zmienną. Nie Określa jej typu. Nie robi tego – jawnie.
- Cała bolączka języka (w porównaniu do innych) zaczyna się w momencie w którym chcemy przypisać do hello coś co nie jest typu String (np. Number) – w JavaScript, jest to możliwe i nie powoduje błędu
- W innych (wymienionych wcześniej językach) jest to operacja niedopuszczalna i powoduje błąd.

Czemu właśnie tak?

- Z jednej strony :
  - – daje nam (programistom) to swobodę w implementacji różnego rodzaju rozwiązań
  - Pozwala również szybko coś napisać – zmodyfikować
  - Obniża „próg wejścia” do języka i sprawia że język jest „dla każdego”

- Jednakże są minusy:
  - W miarę jak aplikacja się rozwija – zmiany w różnych miejscach mogą powodować masę błędów
  - Nie jest lekko z „podpowiadaniem składni” przez IDE

Te minusy są często barierą nie do przejścia dla niektórych programistów
(nie chcą oni pisać w JS)

Gdzie działa JavaScript?

- Głównie: W przeglądarce. Po to został ten język przygotowany i tam sprawdza się od lat 90’ aż do dzisiaj
- Dodatkowo: Na serwerze
  - W 2009 roku – prawdziwy BOOM zrobił: Ryan Dahl ze swoim pomysłem: **Node.js**
  - Dzisiaj większość projektów front-endowych „piszę się” (dewelopuje) w Node. Ponieważ „zaprzęga się go” - do pisania rzeczy pod przeglądarkę.
- I jeszcze: IoT (Internet of Things) / Hardware – coraz więcej rzeczy można już programować z użyciem JS’a – np. Arduino

Czy są różnice w działaniu JS na Browser i na Back-End?

- I tak i nie. Generalnie chodzi o samo środowisko.
- Porównajmy Chrome oraz Node.js:
  - Interpreter JS jest ten sam: V8
  - Jednak w **Chrome** to przeglądarka – ma więc obiekt globalny: `window`, oraz `document` oraz posiada dostęp do DOM (Document Object Model) – tak by można było modyfikować HTML
  - Skrypty w przeglądarce dołączone standardowo za pomocą `<script src=„”>` nawet jeśli są w oddzielnych plikach – „mieszają swoją przestrzeń globalną” – innymi słowy zmienna zdefiniowana w pliku a.js jest dostępna w pliku b.js
  - **Node.js** ma swój własny globalny obiekt „global” i nie znajdziemy tam ani window, ani document
  - Domyślnie plik ma swój własny zakres i nawet jeśli go „`require`’ujemy” w innym pliku – to nie ma on dostępu do niczego – czego inny plik nie „wy’`export`’uje” na zewnątrz. Funkcjonujące tam działanie i jego zapis nazywa się: **CommonJS**.

Język a środowisko

JavaScript w Przeglądarce i w Node.js

JavaScript w przeglądarce

- Potrafi zarządzać DOM
- Ma dostęp do funkcji przeglądarki: pasek adresu (URL), nawigacja, informacje na temat systemu
- Ma dostęp do interface'ów i API związanego np. z: mikrofonem, kamerą, lokalizacją użytkownika, itp. itd.
- Dzięki temu możemy pisać aplikacje web.
- Jednak – jaki jest główny problem tego podejścia?

Standardowe działanie JS in Browser

- Niestety dołączane pliki do strony – „mieszają się” innymi słowy – ich zawartość jest wzajemnie dla siebie dostępna.
- Powoduje to masę problemów, takich jak np. tak samo nazwane zmienne w 2, 3, 5 różnych plikach.
- Dodatkowo nie da się łatwo „dynamicznie” dodać skryptu JS na stronę bez użycia technologii dodatkowych – serwerowych (PHP, ASP, JSP) etc…

- Jest również problem wydajnościowy – nie chcemy żeby nasza aplikacja JS pobierała kilka – kilkanaście plików .js, najlepiej byłoby mieć jeden plik – wtedy będzie to tylko jeden Request ze strony użytkownika końcowego

Rozwiązanie problemów przeglądarki

- Część problemów – dała się rozwiązać jeszcze w latach 2000
- Powstawały różne modele (biblioteki) dołączania zależności plików
- Wykorzystywano konstrukcje typu IIFE (Immediately Invoked Function Expression)
- Dążono w ten sposób do osiągnięcia tzw. „Modułowości JavaScript”.
  - W skrócie: aby plik .js nie udostępniał swoich zasobów innemu jeśli wyraźnie tego nie zadeklarujemy w tym pliku a z 2 strony nie „zaimportujemy” tych zasobów w innym pliku.
- Przełomem okazał się rok 2009 gdzie powstaje Node.js – jego idea **CommonJS** opiera się właśnie o taki model „udostępniania” elementów między plikami.

Modułowość w przeglądarce?

- Dopiero po roku 2015 wchodzą oficjalnie moduły
- W tym układzie do znaczników `<script></script>` w HTML dodajemy atrybut:
  `type="module"` – dzięki temu dane z załączonego skryptu są „hermetyczne” – udostępnione innemu skryptowi jest jedynie to co posiada słowo kluczowe `export`
- Do składni języka dochodzą zapisy: `import` oraz `export`

JavaScript w Node.js

Potrafi wykonywać się server-side

Posiada dostęp do systemu plików - może nimi zarządzać

Dzięki temu możemy napisać np.. serwer lub program konsolowy w JavaScript

Modułowość z automatu

- W Node.js – zawartość plików .js jest automatycznie „hermetyczna”.
- Istnieje tutaj specjalna składnia `module.exports` do eksportowania danych.
- `require('module');` - składnia do importowania danych z modułów lub z bibliotek dołączonych za pomocą managera pakietów (**npm**)

A gdyby tak wykorzystać Node.js – do produkcji front-endu ?!

- Dokładnie to stało się po roku 2009 i rozwija się aż do dzisiaj.
- Powstał mocny ekosystem zależności oraz wiele bibliotek – które wykonują się po stronie serwerowej – czyli w tym układzie na naszym komputerze
- Pomagają one nam w procesie tworzenia oprogramowania
- Dostarczane są za pomocą npm i uruchamiane dzięki Node.js