# Transkrypcja webinaru:

# Podstawy NodeJS

Artykuł ten powstał w oparciu o **webinar *Podstawy NodeJS***, który odbył się 20 lutego 2018 roku.

Pierwsza część to troszeczkę więcej mówienia z mojej strony. Przedstawię Wam, czym jest Node, czym się charakteryzuje, jakie są korzyści używania Node'a oraz jakie są minusy - na co trzeba uważać. Troszeczkę opowiem o architekturze Node'a i następnie przedstawię Wam, jak można zainstalować Node'a i jak zacząć z niego korzystać.

W następnych krokach spróbuję Wam troszeczkę przedstawić na kodzie, jak wykorzystujemy system modułów w Node.js'ie.Troszeczkę zaprezentuję i opowiem na temat NPM'a - menedżera pakietów dla Node'a. Chciałem też przedstawić kwestię, jak mamy sobie radzić z asynchronicznym programowaniem w Node.js'ie, co okazywało się troszeczkę problematycznym przedsięwzięciem, zwłaszcza dla osób, które wywodziły się z takich języków, jak Java czy .Net. Widziałem, że stanowiło to dla nich problem. Mam nadzieję, iż uda mi się pokazać, że nie jest to nic strasznego.

Na koniec spróbujemy stworzyć prosty serwer http. Jeżeli wszystko dobrze pójdzie, to może uda nam się go zdeployować, przynajmniej na jeden z serwerów w chmurze - Heroku albo Azure.

Zacznijmy.

NodeJS jest to środowisko uruchomieniowe JavaScript typu open-source, które możemy uruchomić na dowolnej platformie. Dzięki NodeJS możemy uruchamiać kod JavaScript bez wykorzystywania przeglądarki.

NodeJS jest jednym z niewielu rozwiązań serwerowych opartych na koncepcji programowania sterowanego zdarzeniami, która pozwala tworzyć wysoce skalowane serwery bez udziału wątków - tak, jak ma to miejsce w tradycyjnych rozwiązaniach wielowątkowych.

Ponadto Node obsługuje wiele jednoczesnych równoległych żądań i operacji za pośrednictwem wywołania synchronicznych i nieblokujących operacji wejścia/wyjścia.

Podstawą działania Node'a jest silnik V8, który został stworzony przez firmę Google na potrzeby przeglądarki Chrome, dzięki czemu Node w ekstremalnie szybki sposób jest w stanie wykonać kod JavaScript.

NodeJS istnieje już od 2009 roku. Stał się już dosyć dojrzałą platformą. Na samym początku był stworzony tylko pod Linuxa. Na szczęście od 2011 Microsoft wspomógł platformę i pomógł stworzyć wersję dla Windowsa.

Pomimo, że jest to projekt open-source'owy, to czuwa nad nim fundacja Node.js będąca też członkiem fundacji Linux Foundation. Wraz ze swoimi członkami ma ona za zadanie promowanie i przyśpieszenie rozwoju projektu i technologii NodeJS.

Pomimo, że NodeJS jest projektem typu open-source jest bardzo poważnie traktowany dzięki zaangażowaniu się liderów IT w jego rozwój.

Jak tutaj zaznaczyłem, kilku z głównych członków fundacji to bardzo duże firmy, takie jak Google, Microsoft czy IBM, które wspierają platformę Node chociażby poprzez to, że ich pracownicy są jednymi z głównych kontrybutorów platformy. Na przykład Microsoft rozwija wersję Node ze swoim silnikiem Chakra znanym z przeglądarki Edge, co pozwoliło na rozwój platformy. Nawet zespół od silnika V8 z firmy Google od ubiegłego roku traktuje bardzo poważnie Node'a, co zadeklarowali na jednej z konferencji, gdzie oświadczyli, że podczas wysyłania *commita* przy budowaniu kodu silnika V8 żaden *commit* nie pójdzie do repozytorium, jeżeli podczas weryfikacji wysyłania tego kodu okazałoby się, że *bild* Node'a nie przechodzi - że występują błędy podczas budowania Node'a. Wówczas taki *commit* będzie wycofywany i naprawiany.

Kiedyś było to nie do pomyślenia, gdyż zespół V8 skupiał się tylko na Chrome'ie, ale od niedawna bardzo poważnie traktuje też Node'a.

Przejdźmy do głównych korzyści wynikających z Node'a.

NodeJS jest jednowątkowy. Jego architektura oparta jest na zdarzeniach i nieblokujących operacjach wejścia/wyjścia, które pozwalają na maksymalizację wykorzystania pojedynczego procesora i pamięci komputera, dzięki czemu serwery są szybsze i bardziej wydajne.

Architektura sterowania zdarzeniami oparta na synchronicznych wywołaniach umożliwia serwerom NodeJS przetwarzanie większej liczby równoległych żądań niż w przypadku większości konwencjonalnych serwerów wielowątkowych - jak np. Apache, który jest właśnie takim wielowątkowym serwerem.

Ponadto nieblokujące operacje wejścia/wyjścia, które nie blokują wykonywania programu oraz silnik V8 wpływają na to, że platforma NodeJS jest bardzo szybkim rozwiązaniem po stronie serwera. Jeżeli mówimy o szybkości, to ważnym aspektem w przypadku platformy NodeJS jest fakt, że aplikacja jest w stanie bardzo szybko wystartować. Nie mamy tutaj jakiegoś procesu kompilowania, co ma np. Java. Tutaj, jeżeli potrzebujemy wystartować projekt, jest to praktycznie natychmiastowe, co ma w niektórych środowiskach bardzo istotne znaczenie.

Wysoka skalowalność. Dzięki wbudowanym modułom w NodeJS'a jesteśmy w stanie w bardzo łatwy sposób zeskalować naszą aplikację, dostosować dzięki temu do większego obciążenia. Możemy utworzyć dodatkowe procesy Node'a. Pomimo, że Node domyślnie wykorzystuje tylko jeden procesor, to w łatwy sposób możemy zeskalować go na liczbę procesorów, którą mamy w komputerze.

NPM jest to menedżer pakietów. W sumie pod słowem *NPM* można rozumieć dwie kwestie. Jest to menedżer pakietów - narzędzie wiersza poleceń oraz ogólny rejestr w Internecie, gdzie mamy dostęp do całej masy pakietów, dzięki czemu nie musimy sami wymyślać wielu rozwiązań. Sporo z nich zostało już napisanych przez innych i możemy to wykorzystać. Jest to też spora zaleta, gdyż znacznie może przyspieszyć wytwarzanie oprogramowania.

W przypadku Node'a istotną kwestią jest także społeczność, która stoi za tą platformą, ponieważ wpływa ona w znaczący sposób na jej rozwój. Sam projekt Node'a ma ponad 1 500 kontrybutorów, bardzo dużo powstaje też tutoriali, zasobów oraz przykładowych projektów. W ostatnich latach ta aktywna społeczność powoduje, że platforma zyskuje na popularności. Jest ona obecnie jedną z najczęściej używanych technologii, po którą się coraz częściej sięga.

Kolejnym ważnym aspektem z mojego punktu widzenia jest JavaScript, czyli język, który tak naprawdę jest łatwy do nauki, zyskuje na popularności, jest w ostatnim czasie mocno rozwijany i

udoskonalany. Ten sam język na frontendzie co na backendzie oznacza też tak naprawdę, że możemy dysponować mniejszym zespołem, który będzie bardziej wydajny, może się lepiej komunikować i w efekcie dostarczyć zadanie znacznie szybciej.

Tutaj jest jednak mała uwaga, że owszem jest to możliwe. Potwierdzeniem tego jest użycie Node'a przez PayPal'a, gdzie oni właśnie mniejszym zespołem byli w stanie szybciej wytworzyć oprogramowanie, aczkolwiek tutaj muszą być to doświadczone osoby, które będą wiedziały, jak się na tej platformie poruszać.

Do czego możemy wykorzystać NodeJS'a?

W Internecie najczęściej podawanymi przykładami wykorzystania dla Node'a pojawiają się trzy użycia:

- strumieniowanie danych - Node ze względu na swój asynchroniczny charakter jest idealny do obsługiwania transmisji strumieniowej, tak do aplikacji w czasie rzeczywistym, do przesyłania mediów, różnych źródeł plików czy też jako serwer WebSocket'ów;
- wykorzystanie go jako serwer API jest też bardzo naturalnym rozwiązaniem - nie dość, że może obsłużyć wiele połączeń naraz, to format JSON jest w naturalny sposób wykorzystywany przez JavaScript i z łatwością możemy przekonwertować obiekty JavaScript do formatu JSON;
- Mikroserwisy - tutaj tak samo jesteśmy w stanie łatwo komunikować się z innymi serwisami także poprzez użycie formatu JSON, a dodatkowo, ponieważ Node jest szybki i nie obciąża tak systemu, jesteśmy w stanie stworzyć mikroserwisy w łatwy i w miarę szybki sposób.

Jeżeli spojrzymy sobie trochę szerzej, to - patrząc na popularność platformy (sama liczba modułów w rejestrze NPM, których przybywa w tempie trochę takim szalonym, bo w sumie przybywa 400 modułów na dzień) - można powiedzieć o platformie NodeJS, że można ją znaleźć wszędzie. Co prawda, jakość udostępnianych pakietów jest bardzo zróżnicowana - trzeba mieć to na uwadze - ale niosą też one bardzo wiele ciekawych, nowych rozwiązań, dzięki czemu umożliwiają nam wejście do bardzo wielu różnych obszarów.

Tak na przykład NodeJS w bardzo mocnym stopniu wspomógł rozwój narzędzi stricte dla programistów frontendowych. Wszelkiego rodzaju narzędzia typu: Babel, Eslint, Typescript, Webpack - to wszystko bazuje jakby pod spodem na NodeJS'ie.

Tworzenie aplikacji mobilnych - wszelkiego rodzaju frameworki, jak: Ionic, React native, NativeScript wykorzystują w jakiś sposób Node'a. Bardzo często mają swoje dodatkowe narzędzia z wiersza poleceń do wspomagania.

Kolejna kwestia - aplikacje desktopowe. Coś, co by się mogło wydawać, że trochę rzadko już występuje, ale też dzięki powstałemu frameworkowi Electron możemy pisać aplikacje desktopowe - używając właśnie Node'a - na dowolną platformę. Aplikacja stworzona w Electronie będzie działała na Windowsie, na Macu i na Linuxie.

Wiele popularnych rozwiązań zostało już napisanych przy użyciu tego frameworka - np. Visual Studio Code o środowisku programistycznym stworzonym przez Microsoft, czy też Slack

Communicator - np. często stosowany przez zespoły programistyczne.

Kolejny obszar, który stał się możliwy jak gdyby trochę dzięki Node'owi, gdzie jest łatwiejszy dostęp, to jest Internet rzeczy…

Dominik napisał (na czacie) akurat ciekawy komentarz, z którym się zgadzam - jak klient nie chce webaplikacji to można to wykorzystać. Możemy sobie ładnie ostylować, stworzyć… Tam tak naprawdę jest jak gdyby strona zaszyta. Ale jest dostarczone w takiej paczce - jak taką standardową aplikację byśmy instalowali.

Internet rzeczy. Możemy sobie tutaj oprogramować roboty - coś, co generalnie raczej kiedyś wymagałoby bardziej specjalistycznej wiedzy. Wcześniej się nigdy tym nie interesowałem. Troszeczkę zacząłem się tym bawić, kiedy miałem do czynienia z Node'em, Gdzieś się pojawiła zabawa trochę z Raspberry Pi i też właśnie powstał framework Johny five, który w znaczny sposób ułatwia oprogramowywanie robotów.

Kolejny obszar, który jest teraz bardzo intensywnie wspierany, jeśli chodzi o Node'a, to wszelkiego rodzaju usługi chmurowe. Różni dostawcy - AWS, Microsoft Azure czy Google Cloud Platform - praktycznie wspierali chyba od samego początku Node'a.

Jeżeli chodzi o kwestię serverless - w ostatnim czasie dosyć popularną - gdzie budujemy swoją dyskretną funkcję osadzoną właśnie na środowisku chmurowym. Node do takiego use case'u bardzo dobrze się nadaje, ponieważ jest zoptymalizowany pod kątem działania w pojedynczym procesie z jak najlepszą wydajnością. Potrzebuje on mało przestrzeni pamięci, jest w stanie wykorzystać to środowisko w bardzo efektywny i wydajny sposób. Dzięki temu, że może zostać bardzo szybko ponownie uruchomiony możemy tworzyć całe aplikacje i rozwiązania w chmurach, co umożliwiają dostawcy chmurowi.

Te zróżnicowane obszary są elementem, który bardzo lubię, jeśli chodzi o Node'a . Interesowało mnie zawsze to, że dzięki tej platformie przed programistami otwierają się możliwości dotknięcia obszarów, o których wcześniej by nie pomyśleli, żeby robić tego typu rozwiązania - chociażby tworzyć, programować jakieś roboty… Mamy teraz swobodny dostęp do takich usług chmurowych, więc pod tym względem bardzo fajne możliwości otwiera ta platforma.

W ostatnim czasie często sięgają po Node'a różnego rodzaju firmy, start-up'y, ale też i duże przedsiębiorstwa. Tutaj wstawiłem kilka firm, które udanie wdrożyły u siebie Node'a. Są to bardzo duże firmy, których historie wdrożenia NodeJS możemy przeczytać w Internecie, gdzie z reguły bardzo pozytywnie o tym mówią - o wszelkich korzyściach, które z tego wynikają.

Przykładowo Netflix po wdrożeniu Node'a do swojego środowiska był w stanie skrócić czas uruchomienia o 70% w stosunku do poprzedniego rozwiązania na Javie. Nowa aplikacja Linkedin jest bodajże dziesięć razy szybsza niż jej poprzedniczka oparta o Ruby. Linkedin twierdził, że był w stanie zmniejszyć liczbę używanych serwerów z 30 do 3. Node jest w stanie bardziej efektywnie wykorzystać dostępne zasoby serwera.

Ale żeby tak nie było "różowo"... Node z uwagi na jego jednowątkową naturę wiąże się z ograniczeniami. Został on stworzony z myślą o bardzo dużych obciążeniach, jeśli chodzi o operacje wejścia/wyjścia. W tym jest świetny. Jeżeli będziemy chcieli wykonywać operacje, które będą wymagały sporo mocy obliczeniowej procesora, to tutaj może zajść sytuacja, że zablokujemy sobie naszą aplikację i nikt nie będzie w stanie z niej skorzystać do czasu aż ten wątek zostanie odblokowany.

Do takich sytuacji, które mogą spowodować nasze problemy wydajnościowe możemy m.in.

zaliczyć obsługę synchronicznych operacji wejścia/wyjścia. W kodzie raczej należy unikać tego typu rozwiązań. Możemy jak najbardziej zastosować - i to się robi najczęściej - synchroniczne operacje wejścia/wyjścia na starcie aplikacji. Zanim wystartujemy serwer WWW, mamy jakieś elementy konfiguracyjne, wówczas używamy sobie synchronicznego I/O, to nie robi dla nas żadnej różnicy. Gdy jednak mamy już serwer webowy, to takiej sytuacji należy unikać.

Skomplikowane szablony do renderowania. Sam jeszcze nie spotkałem się z tak bardzo skomplikowanym szablonem, żeby Node miał jakiś problem z wyrenderowaniem. Z reguły bardzo dobrze sobie radzi, aczkolwiek - z tego, co słyszałem - były też takie sytuacje. Jeżeli ktoś by bardzo przesadził z bardzo dużym skomplikowanym szablonem, jego wyrenderowanie mogłoby przysporzyć duże problemy dla Node'a.

Pasowanie dużych obiektów JSON. Tutaj tak naprawdę duże, kilkumegowe obiekty mogłyby przytkać nam troszeczkę Node'a, ponieważ jest tam wykorzystywana metoda JSON parse, która jest synchroniczna, więc tutaj moglibyśmy sobie zaszkodzić. Dlatego jeśli udostępniamy jakiegoś JSON'a, wystawiamy na jakimś API, należy dobrze zweryfikować, jakie właściwości wysyłamy do klienta.

Wyrażenia regularne mogą się okazać problematyczne na przykład w sytuacji, gdzie mamy serwer webowy, mamy zdefiniowany jakiś endpoint - jakąś trasę, na której mamy zdefiniowane jakieś działanie. Tą trasę mamy jednak zdefiniowaną przy pomocy wyrażenia regularnego i w przypadku jego błędnego zapisania moglibyśmy narazić naszą aplikację na atak ReDoS, czyli Regular Expression Denial Of Service, co spowodowałoby, że nasza aplikacja mogłaby przestać odpowiadać.

Wszelkiego rodzaju obliczenia kryptograficzne należy wykonywać asynchronicznie, ponieważ wymagają one czasu.

Mam jeszcze uwagę, jeśli chodzi o Node'a, która czasami jest postrzegana jako zaleta, czasami jako wada. Node nie narzuca nam, w jaki sposób będziemy tworzyć naszą aplikację, daje pełną swobodę w jej tworzeniu. Ta jego elastyczność postrzegana jest czasami jako super zaleta, ale z drugiej strony - sam tego doświadczyłem wraz ze swoim zespołem - nie do końca wiedząc na początku, jak sobie poradzić z projektem node'owym możemy zostać trochę przytłoczeni. Przez to, że nie mamy stricte wytycznych możemy czuć się zagubieni. Dlatego ważne jest, jeżeli zacznie się jakiś projekt w Node'zie, raczej nie porywać się na początku na coś dużego, skupić się na jakimś małym fragmencie, jakiejś istotnej implementacji, istotnej funkcji - niech ona będzie mała. Niech działa.

Jeżeli chcemy pisać aplikacje w Node'zie, warto jest zacząć od czegoś mniejszego. My akurat kiedyś zrobiliśmy trochę na odwrót - zaczęliśmy od większego… Nie polecam.

Kilka słów na temat architektury.

Jeżeli chodzi o silnik JavaScript stosowany w NodeJS'ie, to domyślnym silnikiem jest V8. Ale od niedawna, dzięki współpracy z Microsoftem, jest też dostępna wersja Node'a z silnikiem Chakra, gdzie cały czas jest ona rozwijana. Z tego co wiem, nawet Microsoft też używał już tej wersji na swoich stronach, więc działa już to całkiem nieźle.

Żeby jeszcze trochę przejść do kodu…

NodeJS jest nie tylko jakimś fraperem odnośnie silnika V8. Mamy dostępne natywne moduły Node'a napisane w JavaScripcie, które pod spodem tak naprawdę komunikują się z warstwą

napisaną w C++. Sam silnik V8 i sporo kodu w NodeJS'ie zostało napisane w C++. Dzięki modułom natywnym mamy możliwość interakcji systemu plików z sieciami, timerami i innymi obszarami systemu operacyjnego.

Node również wspiera czekanie na synchroniczne zdarzenia używając biblioteki libuv, która została napisana w C specjalnie dla niego. Kiedy Node kończy czekanie na operacje wejścia/wyjścia lub jakieś timery, to ma najczęściej do wykonania jakąś funkcję zwrotną. Kiedy jest wykonanie tej funkcji, Node przekazuje kontrolę do silnika V8. Kiedy silnik V8 kończy wykonywanie kodu, kontrola jest wracana do Node'a.

Silnik V8 jest jednowątkowy, dlatego kiedy wykonuje kod JavaScript NodeJS nie wykona innego kodu, nie zważając na to, ile abstrowanych jest do wykonania. Node będzie czekał aż uzyska ponownie kontrolę, przez co będzie mógł zlecić silnikowi V8 kolejne operacje.

Moduł jednowątkowy - nie musimy martwić się o blokowanie zmiany kontekstu procesora czy jakiś race condition. Mamy tylko jeden wątek - tam, gdzie działa nasz kod.

Biblioteka libuv została zapisana w C i stanowi ona ujednolicony interfejs dla synchronicznych operacji wejścia/wyjścia, wykorzystując funkcje systemowe dostępne w systemach operacyjnych, pozwalające wykonać operacje na ich poziomie bez wątków. Operacje synchronicze są implementowane na poziomie systemów operacyjnych.

Ponieważ nie wszystkie operacje są wspierane asynchronicznie na poziomie systemu operacyjnego, libuv dostarcza pulę wątków do operacji, które nie mogą zostać obsłużone w ten właśnie sposób. Do takich operacji należą na przykład operacje na plikach. libuv tworzy domyślnie pulę z czterech wątków, ale jest to jak gdyby poza nami i nie musimy tego dotykać.

NodeJS, jak każdy system, ma też zbiór swoich różnych zależności…

Przejdę jeszcze do kolejnych elementów, żeby zdążyć przedstawić Wam kod w działaniu.

Jeśli sobie wejdziemy na stronę nodejs.org, możemy sobie ściągnąć wersję - akurat używam Windowsa, więc mam dostępne dwie wersje, są to wersja LTS i Current. W LTS jest to typ wydania, który jest wspierany przez trzydzieści miesięcy - jest to wersja zalecana na serwery produkcyjne. Wersja LTS ma gwarancję stabilności - nie pojawią się tutaj nigdy jakieś zmiany w API, które mogłyby nam coś popsuć na serwerze. Dodawane są tutaj również jakieś łatki i odnośnik bezpieczeństwa.

Wersja Current jest aktualnie rozwijana. Zmieniać się może tutaj API. Jeżeli chcemy trochę poeksperymentować z tym, co udostępnia Node w najnowszych wersjach możemy sobie ściągnąć na potrzeby developmentu, aczkolwiek o wdrożeniu z pójściem na produkcję, to tutaj musimy się trzymać LTS''a.

Pojawi się wersja 10 Current, która w październiku stanie się kolejną wersją LTS.

Na stronie możemy znaleźć też pakiety instalacyjne dla dowolnej platformy. Jest jeszcze taki typ wydania Nightly builds. Co 24 godziny pojawiają się eksperymentalne wydania, jeżeli były jakieś zmiany na głównej gałęzi Node'a, na Githubie. Ale to już jest jak gdyby zupełnie do eksperymentów z tym, co się dzieje na bieżąco z platformą.

Jeżeli chodzi o instalację, warte rozważenia jest użycie takiego rozwiązania jak NVM - Node Version Manager. Jeżelu chcemy mieć zainstalowanych jednocześnie więcej wersji Node'a - LTS, Current lub inne - możemy użyć właśnie tego narzędzia…

Jest oddzielny pakiet pod Windowsa. Jest tutaj na Githubie link MVM fet Windows. Jak instalowałem tą wersję pod Windowsa, miałem akurat problem, że nie działał mi NPM. Musiałem to odinstalować. Być może teraz to się już troszeczkę usprawniło i jest lepiej.

Jeżeli zainstalujemy sobie nowszego Node'a, z poziomu linii poleceń będziemy mieli dostępne polecenie "Node". W momencie, kiedy wpiszemy sobie bezpośrednio samo polecenie "Node" zostanie odpalony interaktywny terminal zwany Rep, gdzie możemy sobie pisać kod JavaScript, podobnie jak w devtools'ach, w Chromie. Możemy tworzyć sobie dowolne fragmenty.

Fajne jest tutaj autouzupełnienie - poprzez wpisanie kropki i wciśnięcie tabulacji mamy podpowiedź, jakie są dostępne metody na ten cel (co możemy wykonać).

W Node'zie - podobnie trochę jak w przeglądarkach - mamy dostępny globalny obiekt, który nazywa się "global". W przypadku interaktywnego terminala Rep ma on załadowane wszystkie natywne moduły Node'a. To jak gdyby w regularnym wykonaniu skryptu node'owego się nie dzieje, wówczas te moduły można zaciągać samemu, ładować w miarę potrzeb, ale tutaj, jak zrobiłem "tab, tab", to wszystkie dostępne tutaj moduły zostały zaprezentowane.

Ten tryb udostępnia nam kilka dodatkowych poleceń. Jeżeli wpiszemy sobie . i użyjemy tabulacji, mamy możliwość na przykład wyczyszczenia.

Kolejną formą szybkiego wywołania kodu poprzez Node'a jest podanie argumentu "-p". Możemy sobie tutaj podać na przykład kawałek kodu, który nam zwróci liczbę procesorów na naszym komputerze. To jest rzadko używane, ale czasami - jakbyśmy potrzebowali na szybko tego typu informacje - można w ten sposób wykonać sobie kod.

Ostatnim sposobem wywoływania kodu jest podanie nazwy pliku. Czyli mamy "node index.js", i jak wywołamy nasze polecenie, mamy wywołany kod i wyświetlony fragment na konsoli.

Chciałem teraz troszeczkę wspomnieć o systemie modułów używanych w Node'zie.

NodeJS posiada system ładowania modułów, który jest oparty na standardzie Commonjs. Są tam minimalne różnice w stosunku do wyjściowego standardu. Ideałem jest tutaj, że moduł jest w korespondencji "jeden do jednego" z plikiem - jeden plik jest traktowany jak moduł.

Chcąc załadować moduł w NodeJS używamy funkcji **require("nazwa modułu")**.

Każdy moduł używa obiektu **module.exports**, aby udostępnić swoje API dla innych modułów.

Funkcja **require** - w momencie, gdy chcemy sobie załadować jakiś moduł do naszego kodu - przechodzi przez kilka faz. Na początku Node musi znaleźć absolutną ścieżkę do modułu, następnie musi zostać ustalone, jaki typ pliku chcemy załadować.

Jest następnie taki element, jak **Wrapping**, dzięki czemu nasz moduł ma stworzony prywatny zasięg. Jeżeli zadeklarujemy sobie zmienną w naszym module, nie tworzy się ona jako zmienna globalna, tylko ma prywatny zasięg w ramach naszego modułu.

Kolejny element to jest **Evaluating**, czyli wykonanie przez silnik V8 naszego kodu.

I ostatni punkt to jest **cachowanie rezultatu**, dzięki czemu, jeżeli będziemy kilka razy ładowali nasz moduł, to kolejne wywołania będą używały cach.

Spróbujemy teraz przez to przejść…

Mam tutaj stworzony plik “index.js” i jeżeli chciałbym w tej chwili załadować sobie jakiś moduł - powiedzmy *webinar* - używam funkcji **require** i podaję nazwę modułu - *webinar*. Jeżeli będziemy chcieli teraz uruchomić nasz kod, Node “krzyknie” nam: *Przepraszam, nie znalazłem tego modułu! Gdzie on jest?!*

Jak Node szuka tego modułu?

Możemy sobie odpowiedzieć zaglądając do obiektu *module*, który jest dostępny lokalnie dla naszego modułu. Jeżeli w tej chwili wykonamy nasz kod, to wyświetli nam się zawartość tego obiektu *module*. Istotna z punktu widzenia ładowania modułów jest tablica *paths*. Zawiera ona ścieżki, po których NodeJS będzie poszukiwał występowania modułu przekazanego do funkcji require.

Poszukiwanie zaczyna się od naszego katalogu w folderze *node_modules*. Jeżeli moduł nie zostanie tam znaleziony, to poszukiwanie idzie wzwyż - aż do samej góry.

Jeżeli stworzę folder *node_modules* i plik *webinar.js*, to w tym momencie Node był w stanie zlokalizować moduł, który żądaliśmy, ponieważ na podstawie ścieżek, które miałem tutaj zdefiniowane, najpierw poszukał sobie folderu *node_modules* w bieżącym katalogu, tutaj znalazł plik *webinar.js* i był już w stanie go załadować.

Oprócz takiej formy podania modułów możemy podać także w ten sposób natywne moduły Node’a, np. FS do systemu plików, aczkolwiek nie przechodzą one przez te fazy, o których wspominałem, są one dostępne natychmiast.

Kolejnym sposobem wskazania na moduł może być podanie relatywnej ścieżki, czyli możemy sobie wskazać na przykład *webinar.js*. Jeżeli przeniosę sobie folder *webinar* na ten sam poziom, co mam swój plik *index*, to wynik załadowania modułu powinien być taki sam. Dzięki temu Node jest w stanie zlokalizować ścieżkę absolutną do modułu.

Kolejnym obszarem w momencie ustalania ścieżki załadowania modułu jest ustalenie, jaki typ pliku chce załadować użytkownik. Tutaj podałem rozszerzenia, aczkolwiek nie musiałbym tego robić. Node w pierwszej kolejności będzie szukał pliku *webinar.js*. Jeżeli go nie by znalazł, to będzie szukał *webinar.json*. Trzeci będzie *webinar.addon* - moduł napisany w C++. W takiej kolejności Node będzie szukał plików do załadowania.

Jeżeli chodzi o utworzenie prywatnego zasięgu, to tutaj nasz moduł jest owinięty przez specjalny wrapper, który możemy sobie podejrzeć odpalając *REPL*. Nasz kod jest opakowywany, jak gdyby wstawiany do środka funkcji, którą tutaj widzicie. Żeby zobaczyć sobie na żywo, możemy tutaj zrobić *console.log(arguments);*.

Skoro to jest funkcja, powinniśmy być w stanie podejrzeć wszystkie argumenty. Na początku jest obiekt *exports* używany do wyeksportowania naszego API z modułu. *Exports* jest tak naprawdę referencją do obiektu *module.exports*, o którym wspominałem. Następnie mamy funkcję **require**, która jest jak gdyby lokalna dla danego modułu. Obiekt *module*…

Ostatnimi parametrami są parametr *final*, czyli absolutna ścieżka do naszego modułu, oraz *dirname* - ostatnia ścieżka do folderu, w którym znajduje się nasz moduł.

Jeśli chodzi o ładowanie modułów możemy też wskazać nie tylko bezpośrednio plik, ale też naszym modułem może być folder. Możemy stworzyć folder *webinar* i w nim domyślnie stworzyć sobie plik *index.js*. W ten sposób możemy też wskazać na folder i załadować go jako moduł.

Pojawiło się pytanie o edytor.

Ja na przykład używam WebStom'a, ale obecnie polecam też Visual Studio Code. Jest to darmowe rozwiązanie Microsoftu, które jest bardzo rozbudowane. Polecam sprawdzić, gdyż jest to bardzo aktywnie rozwijane narzędzie. Jest bardzo dużo dodatków - np. do współpracy z Microsoft Azure.

Jeśli chodzi o ładowanie modułów, w JavaScript pojawiły się natywne moduły w standardzie ES6 - zostały zdefiniowane - aczkolwiek w Node'zie nie ma jeszcze natywnego wsparcia dla tych modułów. W wersji 8.5.0 pojawiło się na razie eksperymentalne wsparcie dla nich, gdzie pliki będące modułem ES6 używały rozszerzenia .mjs. Sądzę, że w tym roku temat wsparcia natywnego tych modułów powinien się domknąć. Planowane wsparcie dla modułów ES6 jest przewidziane na Node'a w wersji 10.

Jeżeli ktoś chce mimo wszystko używać składni ES6, to polecam korzystać z TypeScript'a czy z Babela.

Ogólnie TypeSctipt'a polecam też osobom, które pisały w .Net czy w Javie, z uwagi na zbliżone konstrukcje obiektowe, które tam występują. Osoby te na pewno łatwiej odnajdą się, programując w Node'zie.

Ostatnim elementem, który chciałem troszeczkę przedstawić jest *NPM*.

Jest to menedżer pakietów…

Jeżeli wejdziemy sobie na stronę www.npmjs.com, to możemy sobie wyszukać dowolny pakiet, którego chcielibyśmy użyć w naszym projekcie. Na ten moment jest dostępnych 644 000 pakietów, mamy zatem w czym wybierać. Możemy sobie wejść na dany moduł, gdzie z reguły mamy informacje, jak go zainstalować, jakie jest użycie - pojawia się w miarę szczegółowy opis wykorzystania.

Od strony linii poleceń mamy narzędzie nazwane również *NPM*, którego nie musimy oddzielnie instalować, ponieważ jest ono instalowane wraz z Node'em, warto jest jednak aktualizować je niezależnie do nowszych wersji, ponieważ wydania NPM'a są częstsze niż Node'a.

Jeżeli chcemy sobie zainstalować jakąś zależność do naszego projektu, musimy posłużyć się poleceniem *npm install* i podać nazwę pakietu. W momencie, gdy podam przykładowy pakiet (np. *express* - taki web framework) i wciśniemy *Enter* NPM ściągnie go z rejestru i zainstaluje lokalnie.

W momencie, gdy zainstalowałem tą zależność, mógłbym stworzyć swój plik *index* i korzystać z zainstalowanego modułu,

Jak korzystać z NPM pod WebStorm'em?

Jest tu wsparcie… Jak sobie stworzę na przykład pakiet JSON, to WebStorm w menu kontekstowym udostępni do tego dodatkowe narzędzia.

Jeżeli teraz byśmy chcieli udostępnić ten projekt komuś w zespole, to mielibyśmy problem. *node_modules* nie jest folderem, który powinniśmy udostępniać, Jest duży, jest tam bardzo dużo plików, z reguły instalujemy na nim dużo zależności, zatem udostępnianie tego modułu w jakikolwiek sposób nie jest dobrą praktyką.

W celu udostępnienia naszego projektu, co należy robić bez folderu *node_modules*, musimy posiadać w nim plik *pakiet JSON*, który jest swoistą dokumentacją naszych projektów, gdzie mamy

informacje o nazwie i wersji naszego projektu oraz o zainstalowanych w nim zależnościach.

Żeby tworzyć sobie plik *pakiet JSON*, możemy się posłużyć poleceniem *npm init*. W tym momencie narzędzie zada nam kilka podstawowych pytań dotyczących naszej aplikacji. Tu należy wziąć wszystko domyślnie… I w efekcie powstał nam plik *pakiet JSON*. Mamy tutaj nazwę, która jest domyślna (jak nazwa folderu. w którym jesteśmy), wersję, plik *main* - wskazywany jako główny plik, który byłby uruchamiany w naszym projekcie - i sekcja *dependencies*, która jest tutaj najważniejsza.

Jest to sekcja wskazująca na zainstalowane zależności w naszym projekcie. Ponieważ zainstalowałem już lokalnie *express'a*, NPM od tworzenia *pakiet JSON* wykrył to i od razu wstawił go jako zależność dla naszego projektu.

Kolejnym obszarem jest *devdependencies* - miejsce na zależności deweloperskie wykorzystywane podczas developmentu. Może to być na przykład jakiś framework do testów. Żeby zainstalować jakieś rozwiązanie na potrzeby developmentu, musimy wywołać podobną komendę *npm install* - może też być skrót *npmi* - i przekazać parametr *save-dev*, ewentualnie -D. I teraz podajemy nazwę framework'u, jaki chcemy zainstalować. Tu dla przykładu podam framework *mocha* - do uruchamiania testów.

W tym momencie pojawiła nam się zależność sekcji *devdependencies*. Jeżeli instalujemy projekt na serwerze produkcyjnym, to ta sekcja *devdependencies* nie jest nam potrzebna. Żeby zainstalować tylko zależności z tej sekcji, musimy zrobić *npm install* z flagą *production*. Wówczas, gdy wywołamy to polecenie, zostaną zainstalowane tylko elementy z sekcji *dependencies*.

Tak samo, jeślibyśmy z naszego repozytorium kodu pobrali aplikację, mielibyśmy ją w takim stanie jak w tej chwili. Powiedzmy, że mam jakiś plik wykonywany, jest jakiś tam kod, dostajemy pakiet *JSON* z jakimiś zależnościami… Żeby móc z tego korzystać lokalnie, pierwszą czynnością, jaką musimy wykonać jest wywołanie komendy *npm install*. Wywołując tą komendę NPM wykryje wszystkie zależności zdefiniowane w pliku *pakiet JSON* i momentalnie nam je zainstaluje. W tym momencie pojawił nam się folder *node_modules* i zainstalowane nasze zależności.

Kolejnym ważnym obszarem w *pakiet JSON* jest sekcja *scripts*. Umożliwia nam ona definiowanie dodatkowych skryptów budujących nasz projekt, uruchamiających go. Mamy tutaj jak gdyby dowolność. Są dwa szczególne skrypty - *npm start* i *test*. Żeby uruchomić nasz zdefiniowany skrypt, musimy wywołać komendę *npm start* - w tym momencie nasz kod został wywołany.

Drugą specjalną komendą jest właśnie *npm test*.

Obie komendy są wyjątkowe, przez to, iż wywołujemy je w taki sposób, że zdefiniujemy tylko *npm* i podamy nazwę skryptu. Jeżeli zdefiniujemy nasz dowolny skrypt - np. dev - to musimy go wywołać poprzez *npm run dev*.

Definiowanie takiego skryptu - zwłaszcza start - jest istotne w momencie, gdy chcemy deployować nasze rozwiązanie na jakiś serwer, platformę lub serwis w chmurze. Te platformy, żeby wystartować nasz serwer, używają najczęściej polecenia *npm start*. Warto mieć zatem tutaj zdefiniowany ten skrypt.

Jeśli chodzi o NPM'a, są dla niego przynajmniej dwie alternatywy, które znam. Pierwsza to jest Yarn - menedżer pakietów stworzony przez Facebooka. W momencie, gdy został on stworzony był znacznie szybszy od NPM'a. Pojawił się w wersji 3. Faktycznie była kolosalna różnica, jeśli chodzi o szybkość. Obecnie NPM w wersji 5 jest również szybki.

Dostępną alternatywą jest także narzędzie *pnpm*, które podaje bardzo ciekawe rozwiązanie. Jeżeli będziemy mieli np. sto różnych projektów na naszym dysku i każdy będzie używał tego samego pakietu z tej samej wersji, to pakiet ten zostanie ściągnięty tylko raz. Wszystkie moduły będą linkowane do pierwszego ściągniętego modułu, dzięki czemu zyskujemy na szybkości instalacji oraz więcej przestrzeni dyskowej.

Wykorzystanie NPM w WebStorm'ie.

WebStorm wykrywa, jakie skrypty zostały zdefiniowane w pliku *pakiet JSON*, dzięki czemu można uruchamiać je z jego poziomu. Program ten umożliwia po prostu wywołanie kodu - jeżeli sobie ściągnę jakiś projekt, w którym nie ma jeszcze zainstalowanych zależności, to WebStorm zapyta, czy nie chcę ich zainstalować, ponieważ wykrył *pakiet JSON* i może dokonać tego automatycznie.

Następnym elementem, który chciałem dotknąć jest sposób obsługi asynchronicznych wywołań. Jak sobie radzić z asynchronicznością w Node'zie? Jak to rozumieć?

Przykładem może być tutaj McDonald's. W przypadku McDrive'a musimy czekać w kolejce na złożenie zamówienia. Gdy natomiast zdecydujemy się wejść do środka restauracji, możemy od razu złożyć zamówienie, które jest delegowane dalej - w momencie przyjmowania innych zamówień - a następnie czekamy na jego odbiór. Po prostu szybciej dostajemy informację zwrotną o tym, że nasze zamówienie jest już gotowe.

Jeżeli spojrzymy na to, co oferuje Node, a tak naprawdę JavaScript, jeśli chodzi obsługę operacji asynchronicznych, to mamy trzy główne obszary:

- Callbacks - zwykłe funkcje przekazywane do innej funkcji jako argument. Od samego początku Node'a są one elementem do obsługi wywołań asynchronicznych;
- Promise - pojawiły się od czwartej wersji Node'a i reprezentują jakiś obiekt, który zostanie wykonany w przyszłości;
- Async/Await - składnia, która pojawiła się w zeszłym roku, od S2017, ułatwiająca korzystanie z operacji asynchronicznych.

Operacje Promise i Async/Await służą do tego, aby uciekać od Callbacks, które mogą w wyniku zagnieżdżeń powodować powodować tzw. efekt Callback Hell, gdzie kod staje się bardzo nieczytelny i ciężko się w tym poruszać.

Async/Away pojawiły się w ósmej wersji Node'a.

W Node'zie wszystkie API synchroniczne mają końcówkę *.sync* - po to, żeby było jawnie widać, że jest to operacja synchroniczna, natomiast użycie interfejsu było świadome.

W NodeJS jest stosowane podejście *error first callback*. Większość API natywnych Node'a, używanych przez inne pakiety w NPM, stosuje się do tego wzorca.

Jeżeli chodzi o Promise, to w ósmej wersji Node;a pojawiła się pomocnicza funkcja w module *util*, która nazywa się *promisefy*. Pozwala to zmodyfikować nasze API do wersji zwracającej Promise.

Wśród planów na przyszłość, jakie ma Node jest to, żeby moduły natywne wspierały nie tylko Callbacks, ale dawały też dostęp do wersji Promise. Pierwszy eksperyment z natywnym wsparciem dla Promise pojawia się dla modułu FS.

Ostatnia kwestia dotyczy obsługi składni Async/Away. W celu jej użycia musimy posłużyć się

słowem *async* przed zdefiniowaniem naszej funkcji i następnie mamy wywołanie *away*. Dzięki temu zapis naszego kodu jest czytelniejszy. Operator *away* oczekuje obiektu, który zwraca Promise'a.

W NodeJS wśród natywnych modułów jest dostępny moduł http, który w szybki sposób umożliwia nam stworzenie serwera webowego. Po jego załadowaniu musimy wywołać funkcję *createServer*, której przekazujemy funkcję wywołania zwrotnego otrzymującego dwa parametry.

Użycie natywnego modułu http i pisanie pełnego web server'a byłoby dosyć karkołomnym przedsięwzięciem. Trzeba byłoby przetwarzać samemu przychodzące parametry, co przysporzyłoby dużo pracy. Dlatego też powstały wszelkiego rodzaju frameworki. Jednym z popularnych frameworków webowych jest *express*. Ułatwia on tworzenie naszych web server'ów.

Jego mocą - i innych frameworków mu podobnych - jest koncept *middleware*. Jest to oprogramowanie pośredniczące, które umożliwia podpięcie różnego rodzaju dodatkowych funkcji, które procesują nasze, a więc przychodzące żądanie zanim trafi do docelowej funkcji przetwarzania…

Oprogramowanie pośredniczące pozwala nam przetwarzać żądania przychodzące właśnie takim łańcuchem - w kolejności, jaką sobie zdefiniujemy. Charakteryzuje się ono otrzymaniem trzech parametrów:

- req - żądanie;
- res - odpowiedź;
- next - specjalna funkcja wywołania zwrotnego, którą musimy wywołać, jeżeli chcemy, aby *express* wykonywał w dalszym ciągu łańcuch kolejnych wywołań - żeby to żądanie było dalej procesowane.

Moc programowania pośredniczącego - tych *middleware'ów* - jest w tym, że podczas jego przetwarzania możemy sobie zmodyfikować na przykład obiekt *request*.

Są specjalne narzędzia monitorujące nasz serwer - pliki w naszym projekcie. Jednym z takich popularnych pakietów jest *nodemon*, który monitoruje zmiany na naszych plikach i jest nam w stanie zrestartować serwer, jeżeli wprowadzimy jakąkolwiek modyfikację,

Jak debugować serwer Node'a?

Większość środowisk programistycznych typu WebStorm czy Visual Studio Code ma bardzo dobre wsparcie, jeżeli chodzi o debugowanie Node'a. Jeżeli chcemy zdebugować nasz serwer, możemy przekazać parametr *inspect - brk*. Uruchomi to Node'a w specjalnym trybie - uruchomiony zostanie *inspector*, czyli dodatkowy serwer silnika V8 na WebSocket'ach umożliwiających debugowanie. Komenda *inspect - brk* powoduje też, że jak zaczniemy debugować nasz kod użytkownika, to zostanie on wstrzymany na pierwszej linii zanim się jeszcze wykona. Jest to wygodne, jeżeli chodzi o kwestię debugowania.

Jak wyłączyć niektóre moduły z debugowania w WebStorm'ie?

W *dev tools'ach* to się nazywa *black box* - można sobie w menu kontekstowym na przykład na jakimś module natywnym określić, żeby go wrzucić do takiego *black box'a*. W WebStorm'ie jest też taka możliwość…

Wdrożenie na Heroku jest stosunkowo proste. Jego dokumentacja - tak samo, jak Azure - prowadzi nas za rękę. Jest to dostępne z linii poleceń. W przypadku Azure'a należy pamiętać o zdefiniowaniu sobie darmowego planu, żeby nic nie płacić. Jeżeli chodzi natomiast o Heroku, to ściągamy sobie *Heroku CLI* i raczej nie powinno być już żadnych problemów. Wywołujemy komendę *heroku create*, która tworzy miejsce dla naszej aplikacji oraz zdalnego *remote'a* w naszym repozytorium Git.