npm

Kilka słów o node.js

Czym jest Node.js?

Node.js

Platforma, która pozwala uruchamiać JavaScript bez przeglądarki – tak najprościej można opisać Node.js. Nie byłoby w tym nic nadzwyczajnego, gdyby nie moduły, które dostarcza nam **Node.js**. Oto niektóre z nich:

- moduł HTTP (możesz stworzyć swój serwer),
- moduł File System (możesz obsługiwać pliki).

Node.js jest zbudowany w oparciu o ten sam silnik co Google Chrome czyli V8. W przyszłości będzie mógł wykorzystywać inne silniki takie jak na przykład Chakra od MS Edge.

Node.js to środowisko niebędące przeglądarką do uruchamiania JavaScriptu.

Czym jest npm?

Menadżer pakietów (Package Manager)

Menadżer pakietów to program, którego zadaniem jest zarządzanie zależnościami potrzebnymi do uruchomienia jakiegoś programu (np. pisanej przez nas aplikacji internetowej).

Ten program istnieje po to,
żeby zautomatyzować:

- instalację wielu pakietów,
- usuwanie niepotrzebnych pakietów,
- utrzymywanie zależności (czyli potrzebnych nam pakietów) w odpowiedniej wersji.

Menadżery pakietów istnieją dla praktycznie każdego środowiska programistycznego.

Są to np.:

- dla systemów Unix: apt, packman, dpkg,
- dla języka PHP: composer,
- dla języka JavaScript: npm, bower, volo, jspm, yarn.

Na zajęciach nauczymy się najpopularniejszego menedżera dla JS – npm.

Menadżer pakietów – przykład

Załóżmy, że mamy stronę, która do działania potrzebuje biblioteki **Chart.js**. Jest to biblioteka ułatwiająca tworzenie wykresów (więcej możecie przeczytać o niej tutaj):

https://github.com/chartjs/Chart.js

Biblioteka **Chart.js** sama do działania potrzebuje następujących zależności:

- **`chartjs-color`**,
- **`moment`**.

Biblioteka **chartjs-color** do działania potrzebuje:

- **`chartjs-color-string`**,
- **`color-convert`**.

Menadżer pakietów – przykład

Próba instalowania ręcznie wszystkich potrzebnych zależności mogłaby doprowadzić do pominięcia plików albo wgrania innej wersji niż jest nam potrzebna. Zazwyczaj problemem są wersje starsze.

Menadżer pakietów zainstaluje nam wszystkie potrzebne biblioteki, i to w odpowiedniej wersji.

Dzięki temu szybciej możemy zacząć pracę nad projektem.

Czym jest npm?

Npm (Node Package Manager) jest menadżerem pakietów stworzonym dla środowiska **Node.js**. Składa się z dwóch głównych dla nas części:

CLI – Command Line Interface

Jest to narzędzie konsolowe, które umożliwia nam łatwą pracę z npm. Dzięki temu narzędziu będziemy mogli w łatwy sposób zarządzać pakietami w naszym projekcie.

Repozytorium paczek npm

Jest to repozytorium, w którym są dostępne do ściągnięcia wszystkie paczki (i praktycznie wszystkie ich wersje), do których mamy dostęp. To z tego repozytorium narzędzie konsolowe będzie pobierać potrzebne pliki. W 2016 r. w repozytorium było ponad 280 000 paczek.

Oczywiście możemy dodawać własne paczki do repozytorium npm.

Paczka a moduł

W zrozumieniu działania npm bardzo ważne jest zrozumienie dwóch nowych pojęć:

Paczka npm

Paczką jest każdy plik lub katalog opisany przez plik **`package.json`** (o tym za chwilę).

Paczką npm będą tworzone przez nas programy, które będą korzystać z npm. Nawet gdy nie będziemy ich publikować w repozytorium npm.

Większość paczek npma jest jego modułami!

Moduł npm

Modułem npm jest wszystko, co możemy dołączyć do naszego programu za pomocą funkcji **`require()`**. Będą to na przykład:

- folder z plikiem **`package.json`** i jakimkolwiek plikiem zawierającym funkcję **`main`**,
- folder z plikiem **`index.js`**,
- osobny plik **JS**.

Czym jest JSON?

JSON – format wymiany danych

Wyobraź sobie, że jedziesz do jednej z chińskich prowincji. Nikt nie zna tam ani angielskiego, ani – tym bardziej – polskiego. Chcesz się porozumieć, ale nie zawsze dogadywanie się na migi pomaga. Potrzebujesz kogoś, kto będzie rozumiał język chiński i angielski.

Podobnie sytuacja wygląda, kiedy dochodzi do komunikacji dwóch (lub więcej) aplikacji napisanych w różnych językach programowania. Potrzebują jakiegoś formatu, który będzie przez nie rozumiany.

JSON – JavaScript Object Notation

JSON – to specjalny format tekstowy, dzięki któremu aplikacje mogą w łatwy i przyjemny sposób wymieniać się danymi. Spójrz na przykład obok.

Ten format jest czytelny nie tylko dla człowieka, lecz także także dla komputera.

JSON – jak już wiesz – jest niezależny od języka programowania, tzn. możemy go wykorzystywać nie tylko w JavaScripcie, lecz także w C, C++, Javie, C#, Perlu, Pythonie i wielu innych.

```json
{
  "id": 1,
  "name": "T-shirt",
  "price": 16.80,
  "tags": ["clothes", "tshirts"]
}
```

JSON – do czego go używamy?

JSON możemy wykorzystywać do przesyłania danych na serwer, odbierania danych z serwera.

Wiele narzędzi używa go również do konfiguracji, np. **npm** do konfiguracji paczek (wkrótce o tym powiemy). Dzięki temu w łatwy sposób możemy powiedzieć menadżerowi paczek jaką konfigurację ma nasz projekt.

Nie tylko **npm** korzysta w ten sposób z JSON-a, również **Webpack**, który poznamy na zajęciach.

Przykład pliku package.json

```
{
  "name" : "mój projekt",
  "version" : "1.2.3",
  "description" : "Moja Aplikacja :D",
  "tags": ["clothes", "tshirts"]
}
```

JSON – struktura

JSON powstał w oparciu o dwie znane nam już struktury:

- **Obiekt** (ale w innych językach programowania może mieć inną nazwę np. rekord, struktura, słownik itp.) – czyli zbiór par **klucz i wartość**,
- **Tablica** (również w innych językach może mieć inną nazwę np. wektor, lista) – czyli uporządkowana lista wartości.

Klucz jest zawsze stringiem! Wartość może być: stringiem, liczbą, obiektem, tablicą, wartością boolean (true, false) lub null.

Obiekt

```
{
  "name" : "Puszek",
  "age" : 2
}
```

Tablica

```
[
  {
    "name" : "Puszek",
    "age" : 2
  },
  {
    "name" : "Pusheen",
    "age" : 1
  }
]
```

JSON – jak go sprawdzić?

Jeśli sami tworzymy JSON, to warto sprawdzić jego poprawność, zanim zaczniemy się nim posługiwać. Możemy do tego celu wykorzystać JSON validator:

- http://jsonlint.com
- http://jsonformatter.curiousconcept.com

Możemy również skorzystać z narzędzi online do generowania losowych danych np.:

- http://www.json-generator.com
- http://www.mockaroo.com

Instalacja i korzystanie z npm

Instalacja środowiska Node.js

Najpierw należy zainstalować środowisko **Node.js**, a dopiero potem korzystać z **npm**.

Preferowanym sposobem instalacji **Node.js** jest ściągnięcie instalatora odpowiedniego dla naszego systemu ze strony:
https://nodejs.org/en/download

Po zainstalowaniu powinniśmy sprawdzić, czy Node poprawnie się zainstalował. Robimy to przez wpisanie komendy:

```
node -v
```

Wskazana wersja powinna być wyższa niż 0.10.32 (najlepiej najnowsza wersja LTS - 6.X.X).

Instalacja npm

Po zainstalowaniu **Node.js** na naszym komputerze będzie od razu dostępny **npm**. Możemy to sprawdzić przez wpisanie komendy:

```
npm -v
```

Wskazana wersja powinna być wyższa niż 2.1.8

Jeżeli wersja jest starsza, to możemy ją uaktualnić do najnowszej wersji dzięki następującej komendzie:

```
npm install npm@latest -g
```

Więcej na temat powyższej komendy dowiesz się na zajęciach.

Instalacja paczek npm

Paczki npm możemy doinstalować do naszego projektu na dwa sposoby:

Instalacja lokalna

Instalacja lokalna polega na zainstalowaniu paczki w folderze, z którego została uruchomiona komenda instalacji.

Paczka ta będzie dostępna tylko i wyłącznie dla projektu, w którym została zainstalowana.

Zazwyczaj stosujemy taką instalację dla modułów, które będziemy załączać w naszym kodzie za pomocą funkcji **`require()`**.

Instalacja globalna

Instalacja globalna polega na instalacji danej paczki w ścieżce systemowej.

Paczka ta będzie wtedy dostępna dla całego systemu.

Zazwyczaj stosujemy instalację globalną, jeżeli paczka udostępnia nam narzędzie konsolowe (CLI). Dzięki instalacji globalnej będziemy mogli z niego korzystać w konsoli.

Lokalna instalacja paczki

Aby zainstalować paczkę lokalnie, należy użyć komendy:

```
npm install
```

lub

```
npm i
```

Npm wyświetli nam informację na temat zainstalowanej paczki (np. jej drzewko zależności, wersję, która została zainstalowana).

Paczka zostanie zainstalowana do katalogu **node_modules** znajdującego się w miejscu, z którego wywołaliśmy komendę instalacji.

Komendę powinniśmy zawsze wywoływać z głównego katalogu naszej aplikacji!

Globalna instalacja paczki

Aby zainstalować paczkę globalnie, należy użyć jednej z komend (litera **`g`** to **`global`**):

```
npm install -g
```

lub

```
npm i -g
```

Tak samo jak w przypadku lokalnej instalacji npm wyświetli nam informacje na temat zainstalowanej paczki (np. jej drzewko zależności, zainstalowaną wersję). Paczka zostanie zainstalowana do katalogu, z którego nasz system będzie miał dostęp do tej paczki i będzie mógł uruchomić CLI, które do niej należy. Zazwyczaj jest to **`/usr/local/lib/node_modules`** dla systemów Unix oraz **`C:/Program Files`** dla systemów Windows.

Komendę możemy użyć z dowolnej lokalizacji.

Instalacja dokładnej wersji paczki

Czasami chcemy zainstalować dokładną wersję jakiejś paczki. Możemy to zrobić następująco:

```
npm install @
```

lub

```
npm i -g @
```

Jeżeli nie wpiszemy wersji paczki, to zawsze zostanie ściągnięta najnowsza wersja dostępna w repozytorium.

Wyświetlanie zainstalowanych paczek

Aby wyświetlić zainstalowane paczki, musimy użyć komendy:

```
npm list
```

lub (jeżeli chcemy wyświetlić paczki zainstalowane globalnie):

```
npm list -g
```

Npm pokaże nam pełne drzewo zainstalowanych zależności. Jeżeli chcemy ograniczyć wyświetlane drzewo tylko do jakiejś głębokości, to możemy użyć opcji:

```
npm list --depth=
```

Gdzie głębokość jest liczbą całkowitą (zaczynamy liczyć od **0**).
Działa też oczywiście do wyświetlania paczek zainstalowanych globalnie.

Usuwanie zainstalowanych paczek

Aby usunąć zainstalowaną paczkę, musimy użyć komendy:

```
npm uninstall
```

lub (jeżeli chcemy usunąć paczki zainstalowane globalnie):

```
npm uninstall -g
```

Uaktualnianie zainstalowanych paczek

Aby uaktualnić zainstalowaną, paczkę musimy użyć komendy:

npm update

lub (jeżeli chcemy uaktualnić paczki zainstalowane globalnie):

npm update -g

Możemy podać wersję, do której chcemy uaktualnić daną paczkę (tak samo jak przy instalacji nowej paczki).

Jeżeli nie podamy wersji, to paczka zostanie uaktualniona do najnowszej dostępnej na repozytorium wersji.

Wyszukiwanie paczek

Czasami wiemy, jaką paczkę chcemy zainstalować, ale nie pamiętamy jej pełnej nazwy. Możemy wtedy skorzystać z dwóch opcji:

Komenda konsolowa

npm search

Npm wyświetli nam wtedy wszystkie paczki mające szukaną frazę w nazwie. Npm przedstawi nam też krótki opis, nazwę twórcy i najnowszą dostępną wersją.

Strona internetowa

https://www.npmjs.com

Na górze strony jest wyszukiwarka paczek.

Zarządzanie zależnościami

Zarządzanie zależnościami

Zarządzanie zależnościami jest kolejną cechą menedżerów pakietów.

Przy większym projekcie będziemy korzystać z wielu pakietów, innych dla wersji produkcyjnej naszego projektu, innej dla wersji testowej, a jeszcze innej dla wersji deweloperskiej.

Nieefektywne byłoby ręczne instalowanie każdego pakietu za każdym razem, kiedy nowa osoba trafia do projektu albo kiedy projekt jest migrowany na inny serwer.

Plik package.json

Npm rozwiązuje ten problem dzięki plikowi o nazwie **package.json** (plik musi się tak nazywać).

Plik ten zbiera wszystkie informacje na temat naszego projektu. W nim wpisane będą też wszystkie paczki potrzebne do działania.

Poprawne przygotowanie i korzystanie z pliku **package.json** pozwoli szybko zainstalować wszystkie paczki, i to po wpisaniu tylko jednej komendy.

Inicjalizacja pliku package.json

Plik **package.json** możemy napisać ręcznie sami albo skorzystać z generatora, który go dla nas stworzy. Aby uruchomić generator należy użyć komendy:

```
npm init
```

Komenda ta musi zostać wywołana w głównym katalogu naszego projektu.

Po uruchomieniu tej komendy będziemy musieli podać jej wszystkie potrzebne informacje. Odbywa się to w interaktywny sposób.

Pod koniec zostanie nam wyświetlone podsumowanie wygenerowanego pliku.

Najważniejsze pola w pliku package.json

name

Nazwa naszej paczki. Pole obowiązkowe, które musi składać się z liter (małych) i ew. podkreśleń.

version

Wersja naszej paczki. Pole obowiązkowe, które powinno składać się z trzech liczb całkowitych oddzielonych kropkami.

author

Dane autora tej paczki.

licence

Informacje na temat licencji, pod którą rozpowszechniana jest ta paczka.

dependencies

Nazwy paczek (w postaci tablicy JSON), które są potrzebne do działania dla wersji produkcyjnej naszego projektu.

devDependencies

Dodatkowe paczki (w postaci tablicy JSON), które są potrzebne przy tworzeniu naszego projektu.

Najważniejsze pola w pliku package.json

`main`

Plik, który oznaczamy jako plik startowy naszego projektu. Jego rola będzie wytłumaczona przy okazji tłumaczenia funkcji **`require()`**.

Pełen opis wszystkich pól pliku **`package.json`** możemy znaleźć po wpisaniu komendy:

```
npm help json
```

Zależności i ich wersje

Zależności w pliku **package.json** wpisujemy w następujący sposób:

```
{
  "dependencies" : {
    "foo" : "1.0.0",
    "nazwa_paczki" : "wersja_paczki"
  }
}
```

Zależności i ich wersje

Istnieje wiele sposobów podawania wersji paczki. Najpopularniejsze to:

Dokładna wersja

Musimy podać dokładną wersję w formacie **"X.Y.Z"**.

Najnowsza wersja z rodziny

Możemy pominąć któryś z numerów wersji i wstawić tam gwiazdkę. Dzięki temu **npm** ściągnie najnowszą wersję z tej rodziny.

Wersja co najmniej

Jeżeli przed numerem wersji dodamy znak **>=**, to zostanie ściągnięta co najmniej podana wersja (ale może być też każda nowsza – domyślnie zainstaluje najnowszą).

Wersja najwyżej

Jeżeli przed numerem wersji dodamy znak **<=**, to zostanie ściągnięta co najwyżej podana wersja (ale może być też każda starsza).

Zależności i ich wersje

Zależności w pliku **package.json** wpisujemy w następujący sposób:

```
{
  "dependencies" : {
    "foo" : "1.0.0", <- paczka o nazwie foo w wersji 1.0.0
    "bar" : ">= 2.3.5", <- paczka o nazwie bar w wersji co najmniej 2.3.5
    "baz" : "4.*", <- paczka o nazwie baz w najnowszej wersji z rodziny 4
    "baz2" : "1.2.*", <- paczka o nazwie baz2 w najnowszej wersji z rodziny 1.2
    "foobar" : "*", <- paczka o nazwie foobar w najnowszej wersji  (ogólnie)
  }
}
```

Więcej o wersjonowaniu semantycznym (bo tak nazywa się ten sposób wersjonowania) możesz przeczytać tutaj:

- http://semver.org
- http://docs.npmjs.com/misc/semver

Dodawanie zależności do package.json

Do pliku **package.json** możemy dodawać zależności ręcznie, po prostu je wpisać. Możemy też skorzystać z poznanej wcześniej komendy konsolowej **install**:

```
npm install <nazwa paczki> --save
```

Lub:

```
npm install <nazwa paczki> --save-dev
```

Flaga **--save** automatycznie dołączy naszą paczkę do pola **dependencies** w naszym pliku.

Flaga **--save-dev** automatycznie dołączy naszą paczkę do pola **devDependencies** w naszym pliku.

Instalacja wszystkich zależności

Aby zainstalować wszystkie zależności w przypadku, w którym mamy istniejący (i wypełniony) plik **package.json**, musimy użyć komendy:

```
npm install
```

Zostaną automatycznie doinstalowane wszystkie brakujące zależności.

Korzystanie z zainstalowanych paczek

Korzystanie z zainstalowanych paczek

Jak pamiętacie paczki **npm** można instalować na dwa sposoby.
Jest to silnie powiązane z tym, jak będziemy chcieli z nich korzystać:

Instalacja lokalna

Paczka ta będzie dostępna tylko i wyłącznie dla projektu, w którym została zainstalowana.

Zazwyczaj stosujemy taką instalacje dla modułów, które będziemy załączać w naszym kodzie za pomocą funkcji **`require()`**.

Instalacja globalna

Instalacja globalna polega na instalacji danej paczki w ścieżce systemowej. Zazwyczaj stosujemy instalację globalną, jeżeli paczka udostępnia nam narzędzie konsolowe (CLI). Dzięki instalacji globalnej będziemy mogli z npm korzystać w konsoli.

Korzystanie z globalnych paczek

Paczki globalne instalujemy po to, żeby korzystać z ich narzędzi konsolowych **(CLI – Command Line Interface)**. Są to małe programy napisane w JavaScript, które możemy uruchomić z linii komend.

Jednym z przykładów takiego programu jest `jslint`. Jest to linter do języka JavaScript.

Użycie jslinta zostanie pokazane na ćwiczeniach.

Lintery to programy sprawdzające poprawność napisanego kodu w danym języku.
Nie sprawdzają go jednak pod względem poprawności programistycznej, ale poprawności składniowej konwencji pisanego kodu.

Oznacza to, że linter nie zauważy pętli nieskończonej w waszym kodzie, ale pokaże wam wszystkie miejsca, w których macie źle napisane wcięcia lub używacie złej konwencji nazywania zmiennych.

Korzystanie z lokalnych paczek

Paczki lokalne instalujemy po to, żeby korzystać z nich w naszym kodzie jak z bibliotek. Dzięki temu możemy w łatwy sposób rozszerzać nasz projekt o dodatkowe funkcje, napisane już przez kogoś innego.

Aby dołączyć kod jakiegoś modułu, musimy użyć funkcji **`require()`**, która jest częścią npm i środowiska Node.js.

Node.js jest technologią serwerową (służy do pisania aplikacji serwerowych w JS zamiast w PHP, RoR albo innych językach), funkcja **`require()`** nie jest dostępna z poziomu przeglądarki.

Aby skorzystać z doinstalowanych bibliotek, będziemy musieli poznać jedno z narzędzi do budowania kodu (**build tool**). Więcej o nich dowiecie się podczas wprowadzenia do Webpacka.

Przykład

Łączenie plików lokalnych

Stwórz dwa pliki:

- **parent.js**,
- **child.js**.

W pliku **child.js** wpisz:

```
console.log("Witaj w pliku child.js");
```

W pliku **parent.js** wpisz:

```
require("./child.js");
console.log("Witaj w pliku parent.js");
```

Łączenie plików lokalnych

Uruchom konsolę i wpisz:

```
$ node ./parent.js
Witaj w pliku child.js
Witaj w pliku parent.js
```

Jak widzisz uruchomiliśmy plik **`parent.js`**, do którego został załączony plik **`child.js`**.

Dodajmy do naszego pliku **parent.js** paczkę **validator**.
Użyjemy z niej funkcji **isEmail(string)**.

W pliku **parent.js** dopisz następujący kod:

```
require("./child.js");
var validator = require("validator");
console.log("Witaj w pliku parent.js");
var isOk = validator.isEmail("foo@op.pl");
console.log(isOk);
```

Uruchom konsolę i wpisz:

```
node ./parent.js
```

Powinien pokazać Ci się błąd:

```
$ node  ./parent.js
  Witaj w pliku child.js
  module.js:341
  throw err;
  ^
  Error: Cannot find module 'validator'
  at Function.Module._resolveFilename (module.js:339:15)

  /.../
```

Dodawanie paczek npm do pliku

Dlaczego wystąpił błąd? To dlatego, że nie mamy zainstalowanej paczki **`validator`**.

Wpisz zatem w konsolę:

```
npm install --save-dev validator
```

```
$ node ./parent.js
Witaj w pliku child.js
Witaj w pliku parent.js
true
```

Ewentualne problemy

Globalna instalacja paczek – problemy z dostępem

Czasami podczas globalnej instalacji paczek możemy dostać błąd **EACCES** (Access error – błąd dostępu). Spowodowany on jest niemożliwością zainstalowania paczki w katalogu systemowym bez odpowiednich uprawnień.

Możemy go rozwiązać na trzy sposoby:

1. Zmienić katalog, do którego instalujemy paczki globalne (zalecane).
2. Zmienić uprawnienia dostępu do katalogu, w którym są instalowane paczki globalne (niezalecane).
3. Instalować paczki z uprawnieniami administratora (bardzo niezalecane).

W naszej prezentacji zostanie opisana tylko metoda numer 1 (**zalecana**).

Zmiana ścieżki instalacji

Aby zmienić ścieżkę, w której globalnie instalujemy paczki, musimy wykonać następujące kroki (przez wpisanie w terminalu podanych komend):

1. Stworzyć nową ścieżkę, w której będą trzymane paczki:

   mkdir ~/.npm-global

2. Poinformować npm gdzie (od teraz) powinien instalować globalne paczki:

   npm config set prefix '~/.npm-global'

3. Upewnić się, że system i npm będą w stanie znaleźć zainstalowane globalnie paczki:

   export PATH=~/.npm-global/bin:$PATH

4. Przeładować nasz terminal, żeby korzystał z nowej ścieżki:

   source ~/.profile