Programowanie aplikacji internetowych

2019/20

Instrukcja laboratoryjna cz.6

Backend

Prowadzący: Tomasz Goluch

Wersja: 1.0

I. Wprowadzenie

Cel: Przekazanie podstawowych informacje o zasobach laboratorium.

Laboratorium odbywa się na maszynach fizycznych, z wykorzystaniem preferowanego edytora lokalnego (np. *Atom*, *Sublime text*, *Neovim*…) albo on-line (np. https://jsfiddle.net, https://codepen.io, https://codesandbox.io).

II. MongoDB

Cel: Zapoznanie z dokumentową bazą danych MongoDB.

Instalator można pobrać z: https://www.mongodb.com/download-center/enterprise, po zainstalowaniu uruchamiamy mongo poleceniem:

```
mongo[1]
```

Tworzymy nową bazę danych:

```
use <nazwa_bazy_danych>
```

Dodajemy kolekcję:

```
db.users.insert({ name: "student", password: "student"})
```

Sprawdźmy czy baza oraz kolekcja została dodana:

```
show dbs
show collections
```

oraz czy element kolekcji został poprawnie dodany do bazy:

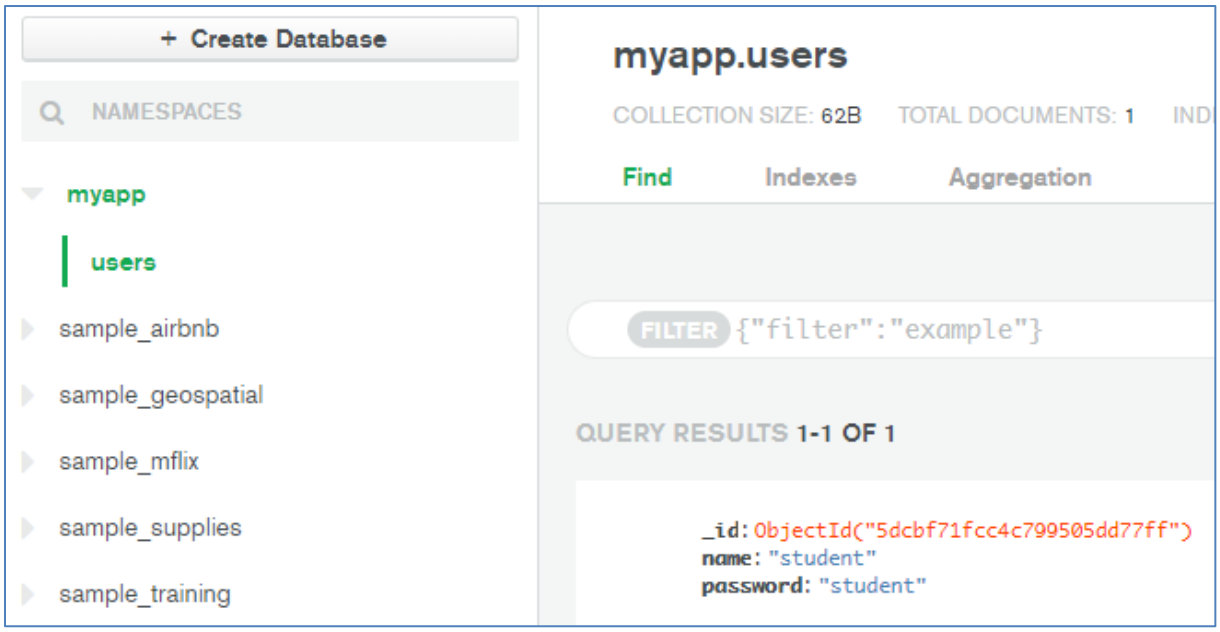
```
db.users.find()
```

MongoBD dostępny jest również w wersji darmowej w chmurze: https://www.mongodb.com/cloud.

Po założeniu konta należy dodać nowy klaster (może to zająć nawet kilka minut). Z klastrem możemy połączyć się za pomocą Mongo Shell, aplikacji MongoDB Compass albo bezpośrednio z naszej aplikacji.

Przykłady połączenia można znaleźć (po założeniu konta i zalogowaniu się) w menu https://cloud.mongodb.com/: ATLAS → Clusters → przycisk: CONNECT. Należy pamiętać aby wcześniej dodać adres IP z którego się łączymy do białej listy (menu: SECURITY → Network Access → IP Whitelist), ewentualnie można zezwolić na dostęp z dowolnego IP (0.0.0.0/0), przycisk: ALLOW ACCESS FROM ANYWHERE. Po dodaniu bazy danych oraz kolekcji będzie ona widoczna w menu: ATLAS → Clusters → COLLECTIONS:

[1] Może być wymagane dodanie ścieżki do zmiennej środowiskowe PATH: PATH=%PATH%;"C:\Program Files\MongoDB\Server\X.X\bin", gdzie X.X to numer wersji MongoDB.



III. Mongoose[2]

Cel: Zapoznanie z biblioteką modelowania danych Mongoose.

Mongoose to nakładka komunikująca się z bazą danych MongoDB. Zapewnia proste, oparte na schematach modelowanie danych aplikacji. Instalacja Mongoose za pomocą npm:

```
npm install mongoose
```

Mongoose pozwala od razu zacząć korzystać ze modeli danych, bez czekania, na ustanowienie połączenia z MongoDB[3]. Połaczenie z lokalną bazą danych (przykłady):

```
var mongoose = require('mongoose');
var mongoDB = 'mongodb://127.0.0.1/my_app';
mongoose.connect(mongoDB,
        { useNewUrlParser: true , useUnifiedTopology: true });
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));
```

Przykładowy connection string do bazy danych w chmurze:

```
var mongoDB = 'mongodb+srv://student:student@cluster0-
ujjjo.mongodb.net/test?retryWrites=true&w=majority';
```

W kolejnym kroku tworzymy model dokumentu/schemat odwzorowywany na kolekcję w bazie MongoDB. Schematy przechowujemy w folderze models. Nazwa schematu wg konwencji wykorzystuje notację camelCase i zaczyna się od nazwy modelu z dodanym postfix'em Schema, oto przykład:

[2] https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose
[3] https://mongoosejs.com/docs/connections.html

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const documentSchema = new Schema({
  first: Number,
  second: { type: String, required: true },
                  ...
});
```

Domyślnie MongoDB tworzy unikalny indeks w polu _id podczas tworzenia kolekcji. Właściwości mogą być różnych typów: `String`, `Number`, `Date`, `Buffer`, więcej tutaj. Domyślnie właściwości nie są oznaczone jako wymagane ale można to zmienić za ustawiając wartość klucza `required: true` . Utworzenie schematu, obiektu i zapisanie do kolekcji bazy danych:

```
var Kolekcja = mongoose.model('Kolekcja', kolekcjaSchema);
const Document = require('<sciezka/do/models/Kolekcja>');

const newDocument = new Document({
  first: 1,
  second: 'second',
        ...
});

newDocument.save((err) => {
  mongoose.disconnect();
  if (err) throw err;
  console.info('User saved successfully!');
});
```

Wyszukanie kolekcji z bazy danych:

```
Document.find(function (err, documents) {
    if (err) return console.error(err);
    console.log(documents);
})
```

IV. WEB API (Express, Cors)

Cel: Zapoznanie z frameworkiem Express.js oraz Cors pozwalającym na łatwe tworzenie interfejsów API.

Instalacja frameworka Express przy pomocy npm:

```
npm install express
```

W celu umożliwienia pobrania zasobów pochodzących z różnych źródeł (domen) wymagane jest skorzystanie z mechanizmu CORS. Instalacja pakietu cors.js w npm:

```
npm install cors
```

Użycie Express.js z Cors.js:

```
var express = require('express')
var cors = require('cors')
var app = express()
app.use(cors())
```

Uruchomienie aplikacji:

```
app.listen(3000, () => {
 console.log("Server running on port ” + 3000);
});
app.get('/', (req, res) => {
  res.json({app: 'Run app’});
});
```

Serwer dostępny jest pod adresem: http://localhost:3000/.

V. Zadanie domowe

W ramach otwartych danych ZTM w Gdańsku mamy następujące API:

https://ckan.multimediagdansk.pl/dataset/c24aa637-3619-4dc2-a171-a23eec8f2172/resource/4c4025f0-01bf-41f7-a39f-d156d201b82b/download/stops.json – **lista przystanków**, zawiera **stopId** – identyfikator słupka przystankowego unikalny w skali Trójmiasta.

http://ckan2.multimediagdansk.pl/delays?stopId={stopId} – **estymowane czasy przyjazdów na przystanek**, gdzie {stopId} jest identyfikatorem słupka – wartość stopId z zasobu Lista przystanków. Przykładowe zapytanie dla przystanku na Miszewskiego (stopId=2019): http://ckan2.multimediagdansk.pl/delays?stopId=2019. Dostajemy kolekcję pojazdów które pojawią się na przystanku w najbliższym okresie. Dane są zapisane z użyciem formatu JSON. Mamy tam kilka właściwości zawierających interesujące aktualne informacje, takie jak: numer linii:"routeId", opóźnienie:"delayInSeconds", czas przyjazdu wg rozkładu: "theoreticalTime", rzeczywisty czas odjazdu: "estimatedTime".

Dane można je porównać ze wybrana podstroną ZTM: https://www.ztm.gda.pl/rozklady/rozklad-002_20191014-21-1.html.

Rysunek 1 - Pretty View JSON'a w Postman'ie z estymowanymi czasami przyjazdów na przystanek Miszewskiego.

W ramach zadania laboratoryjnego proszę zaimplementować prostą aplikację pozwalającą na zalogowanie się użytkownika. Po zalogowaniu wyświetla użytkownikowi listę wybranych przez niego tablic przystankowych. Tablica przystankowa zawiera listę niebawem odjeżdżających z tego przystanku, pojazdów ZTM.

Wymagania techniczne (backend):

- Dane użytkownika (login i hasło) przechowywane w bazie danych dokumentowej albo relacyjnej (np. MongoDB),
- Wykorzystanie odpowiedniego JavaScript ODM (Object-Document Mapper) albo ORM (Object-Relational Mapper) dla wybranej bazydanych.
- Zbudowanie prostego API w node.js (np. z wykorzystaniem frameworka Express.js albo Sails.js) udostępniającego dane logowania użytkownika oraz listę zapisanych przez niego przystanków.

Wymagania techniczne (frontend):

- Implementacja przynajmniej 2 multikomponentów (składające się z co najmniej 2 różnych komponentów).
- Przynajmniej 1 komponent i 1 multikomponent powinny być jednoplikowe.
- Jeden bądź więcej komponentów zawieranych w multikomponentach powinny posiadać powiązane dwukierunkowo zmienne dyrektywa `x-on`.
- Dodanie własnego filtru ładnie wyświetlającego czas odjazdu w formacie: HH:MM:SS, np.: 04:03:45.
- Wykorzystanie dyrektywy `x-bind:class` lub `x-bind:style`.
- Wykorzystać przynajmniej 2 wtyczki (np. omówiona w instrukcji:
  - `vue-good-table`,
  - `vue-resource` pozwalająca na łatwe konstruowanie żądań internetowych i obsługi odpowiedzi za pomocą XMLHttpRequest lub JSONP i współpracująca

np. z `json-serwer` – prostym serwerem HTTP przechowywującego dane w plikach JSON .)

- Implementacja magazynu Vuex, zawierającego przynajmniej 3 zmienne i 3 mutacje, zarówno zmienne i mutacje powinny być odczytywane/popełniane w komponentach.

Mile widziane dodatkowe wymagania techniczne:

- Wykorzystanie tokenów JWS przy autentykacji urzytkownika
- Implementacje i użycie własnej wtyczki
- Własne testy jednostkowe
- Własne testy E2E