MEAN Stack - Node.js, express

Tworzenie serwisów Web 2.0

dr inż. Robert Perliński
rperlinski@icis.pcz.pl

23 lutego 2019

Plan prezentacji

Spis treści

1 MEAN Stack dla średniozaawansowanych **1**
1.1 Node.js . . . 2
1.2 Express.js . . . 3

2 Struktura projektu i moduły **6**
2.1 express-generator . . . 6
2.2 Moduły w Node.js . . . 8
2.3 Silniki szablonów . . . 10

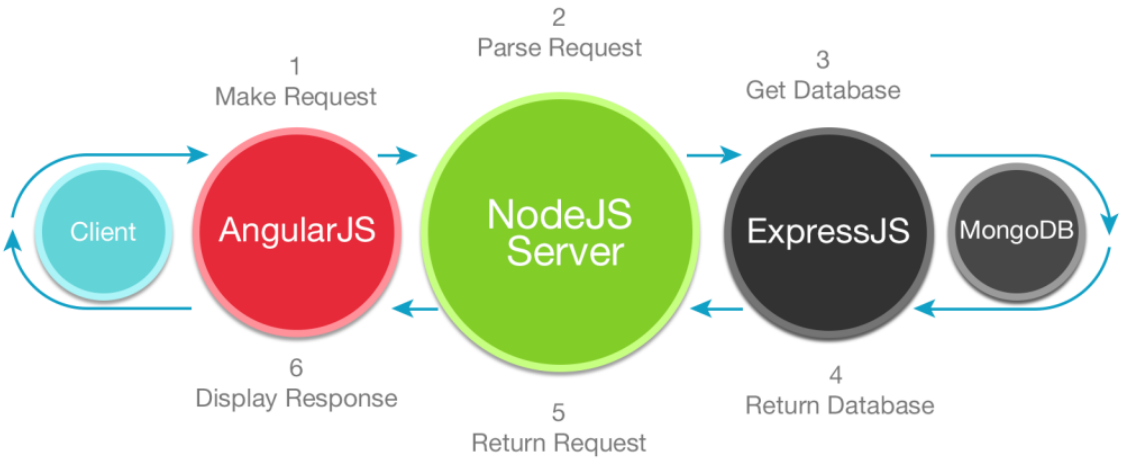
3 Tworzenie własengo API **11**
3.1 Biblioteka mongoose . . . 11
3.2 API dla kolekcji użytkowników . . . 15

4 Źródła **18**

1 MEAN Stack dla średniozaawansowanych

MEAN Stack

MEAN Stack



1.1 Node.js

Node.js

https://nodejs.org/en/

- platforma działająca po stronie serwera na bazie silnika Google Chrome V8,
- zaprojektowany przez Ryana Dahl w 2009 roku,
- ma służyć prostemu tworzeniu szybkich i skalowalnych aplikacji internetowych,
- open source,
- całkowicie darmowe,
- aplikacje napisane w Node.js działają na OS X, MS Windows i Linux'ie,
- używane przez tysiące programistów na całym świecie.

```
/* Program Hello World w Node.js */
console.log("Hello World!");
```

1.2 Express.js

Express

`http://expressjs.com/`

Express to szybki, elastyczny (nie wymuszający konkretnych rozwiązań), minimalistyczny szablon aplikacji internetowych i mobilnych dla Node.js.

Express:

- udostępnia solidną funkcjonalność do budowania aplikacji,
- pozwala na szybkie budowanie aplikacji bazujących na Node.js,
- pozwala na utworzenie warstw pośredniczących odpowiadających na żądania HTTP,
- dostarcza mechanizm trasowania (ang. `routing`) - różne akcje dla różnych metod i adresów URL,
- pozwala na dynamiczne renderowanie stron HTML poprzez przekazywanie argumentów do szablonów.

Express.js i warstwy pośrednie

Instalacja: `npm install express`

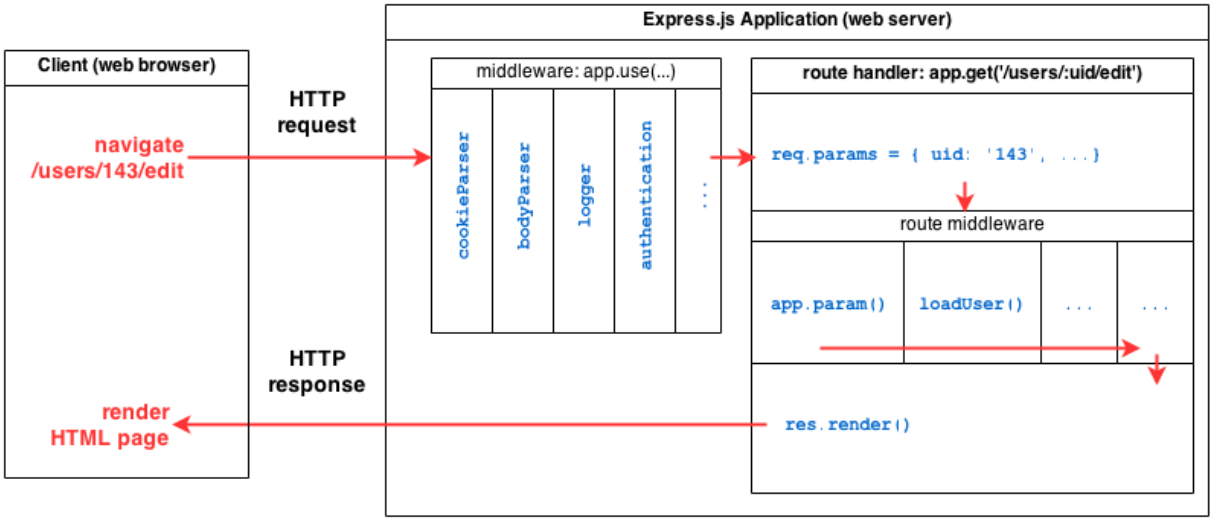
- ExpressJS to komplenty framework.
- Działa z szablonami (jade, ejs, handlebars, hogan.js)
- i kompilatorami CSS (less, stylus, compass).
- Dzięki wartstwom pośredniczącym (oprogramowanie pośredniczące) obsługuje również: ciasteczka, sesje, buforowanie, CSRF, kompresję i wiele innych.

Oprogramowanie pośredniczące:

- łańcuch/stos programów przetwarzających każde żądanie do serwera.
- Takich programów w stosie może być dowolna liczba,
- przetwarzanie odbywa się jeden po drugim.
- Niektóre z nich mogą zmieniać żądanie, tworzyć logi czy inne dane,
- przekazywać je do następnych (`next()`) programów w strumieniu.

Express-middlewares

Warstwy pośredniczące dodajemy do ExpressJS używając `app.use` dla dowolnej metody albo `app.VERB` (np. `app.get`, `app.delete`, `app.post`, `app.update`, ...)



Express - instalacja

```
$ npm install express --save
```

Inne ważne moduły, które warto od razu zainstalować:

- **body-parser** - warstwa pośrednia obsługująca JSON, Raw, Text i dane formularza przekazane w URL,
- **cookie-parser** - przetwarza nagłówki ciasteczek (cookie header) i dodaje obiekt do `req.cookies`, w którym klucze to nazwy przesłanych ciasteczek,
- **multer** - warstwa pośrednia w Node.js do obsługi `multipart/form-data` (kodowanie danych z formularza),
- **mongoose** - połączenie z bazą MongoDB, praca na schematach i obiektach.

Express, Hello World

hello.js

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(5000, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Przykładowa aplikacja nasłuchuje na http://%s:%s", host, port)
})
```

Wynik: `$ node hello.js`
Otwieramy w przeglądarce: http://localhost:5000/

```
Przykładowa aplikacja nasłuchuje na http://0.0.0.0:5000
```

Express używa funkcji zwrotnych z argumentami req i res (obiekty **request** i **response**), które zawierają bardzo dużo informacji i żądaniu i odpowiedzi.

Express, trasowanie I

Reakcje na żądania użytkowników w punktach końcowych (URI, metody protokołu HTTP).

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   console.log("Otrzymano żądanie GET dla strony głównej");
   res.send('Hello GET');
})
app.post('/', function (req, res) {
   console.log("Otrzymano żądanie POST dla strony głównej");
   res.send('Hello POST');
})
app.delete('/usun', function (req, res) {
   console.log("Otrzymano żądanie DELETE dla strony /usun");
   res.send('Hello DELETE');
})
app.get('/user_list', function (req, res) {
   console.log("Otrzymano żądanie GET dla strony /user_list");
   res.send('Lista użytkowników');
})
app.get('/ab*cd', function(req, res) {   //  wzorzec strony: abcd, abxcd, ab123cd, ...
   console.log("Otrzymano żądanie GET dla strony /ab*cd");
   res.send('Wzorzec strony dopasowany');
})

var server = app.listen(5000, function () {
  var host = server.address().address
  var port = server.address().port
  console.log("Przykładowa aplikacja nasłuchuje na http://%s:%s", host, port)
})
```

Express, trasowanie II

Kolejne żądania:

```
GET    - http://localhost:5000/           // Hello GET
POST   - http://localhost:5000            // Hello POST
DELETE - http://localhost:5000/usun       // Hello DELETE
GET    - http://localhost:5000/user_list  // Lista użytkowników
GET    - http://localhost:5000/abcd       // Wzorzec strony dopasowany
GET    - http://localhost:5000/ab123cd    // Wzorzec strony dopasowany
GET    - http://localhost:5000/abXXcd     // Wzorzec strony dopasowany
GET    - http://localhost:5000/abcdef     // Cannot GET /abcdef
```

Reakcja serwera:

```
http://localhost:5000/  -
Przykładowa aplikacja nasłuchuje na http://0.0.0.0:5000
Otrzymano żądanie GET dla strony głównej
Otrzymano żądanie POST dla strony głównej
Otrzymano żądanie DELETE dla strony /usun
Otrzymano żądanie GET dla strony /user_list
Otrzymano żądanie GET dla strony /ab*cd
Otrzymano żądanie GET dla strony /ab*cd
Otrzymano żądanie GET dla strony /ab*cd
```

Trasowanie - dostępne metody

Express udostępnia metody do rootingu (trasowania) zgodne z metodami protokołu HTTP:

- get,
- post,
- put,
- head,
- delete,
- options,
- trace,
- copy,
- lock,
- mkcol,
- move,
- purge,
- propfind,
- proppatch,
- unlock,
- report,
- mkactivity,
- checkout,
- merge,
- m-search,
- notify,
- subscribe,
- unsubscribe,
- patch,
- search,
- connect.

Dla metod, których nazwy nie są poprawnymi nazwami JavaScript używamy notacji z nawiasami kwadratowymi:

```
app['m-search']('/', function ...
```

2 Struktura projektu i moduły

2.1 express-generator

express-generator

- narzędzie generujące szkielet aplikacji do frameworka express.
- instalacja: `npm install express-generator -g`

Użycie: `express [opcje] [katalog]`

Opcje:

```
    --version        wyświetla numer wersji narzędzia
-e, --ejs            wsparcie dla silnika szablonów ejs
    --pug            wsparcie dla silnika szablonów pug
    --hbs            wsparcie dla silnika szablonow handlebars
-H, --hogan          wsparcie dla silnika szablonow hogan.js
-v, --view <engine>  wsparcie dla silnika szablonów <engine>, dostępne
                     są (dust|ejs|hbs|hjs|jade|pug|twig|vash),
                     domyślnie ciągle jest jade
-c, --css <engine>   wsparcie dla silnika stylów <engine>, dostępne są
                     (less|stylus|compass|sass) - prekompilatory css,
                     domyślnie wygląd określamy w zwykłych plikach css
    --git            tworzy plik .gitignore z potrzebnymi wpisami
-f, --force          wymuszenie utworzenie projektu w niepustym katalogu
-h, --help           output usage information
```

Domyślny silnik szablonów to jade ale niedługo się to zmieni.

express-generator bez parametrów

Tworzenie pustej aplikacji: `express hello`

```
hello
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

Pliki widoków w `views`, plik stylu `style.css`, moduły z trasaowaniem w katalogu `routes`,...

Następnie:

- instalacja zależności: `cd hello && npm install`
- uruchimienie aplikacji: `DEBUG=hello:* npm start`

express-generator z parametrami

Tworzenie pustej aplikacji: `express --view hbs --css sass --git hello2`

```
hello2
├── app.js
├── bin
│   └── www
├── .gitignore
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.sass
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.hbs
    ├── index.hbs
    └── layout.hbs
```

Jest inny system szablonów (pliki `*.hbs`), styl trzeba przekompilować(plik `style.sass`), jest plik `gitignore`.

Fragment pliku *app.js*

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var index = require('./routes/index');
var users = require('./routes/users');

var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', index);
app.use('/users', users);
```

express-generator, jak to działa?

Plik wejściowy jest w bin/www.

DEBUG=hello:* npm start - uruchamia skrypt start z pliku package.json czyli:

```
...
"scripts": {
  "start": "node ./bin/www"
},
```

W pliku bin/www jest:

- obsługa całej aplikacji w `../app.js`
- wczytanie modułu `http`
- uruchomienie serwera na porcie 3000, rejestracja dwóch funkcji: do obsugi błędów i nasłuchiwania

W app.js mamy 6 innych modułów: express, path, serve-favicon, morgan, cookie-parser, body-parser. Mamy też użycie dwóch plików JS:

```
var routes = require('./routes/index');
var users = require('./routes/users');
```

Dokładnie tak samo projekt jest generowany przez WebStorm.

express-generator, jak to działa? II

Rooting ze strony głównej i dla adresu `/users`:

```
app.use('/', index);       // ten jest dla stron
app.use('/users', users);  // ten jest pod API
```

- W pliku `app.js` jest podpięcie trasowania z index.js i users.js.
- Ścieżki trasowania są podzielone na dwa pliki, najpierw początek adresu jest ustalony w `app.js`, reszta jest w `index.js` i `users.js`.
- Następnie jest sporo warstw pośredniczących (matoda `app.use()`) do obsługi JSONa, wysyłania formularzy metodą POST, obsługi ciasteczek, dostarczania statycznych stron,...
- Jest też obsługa błędów, przekazywanie ich do wyświetlenia.

Ustawienia aplikacji

Tworzymy właściwości przypisując wartości do nazw zmiennych (nazwa, wartość), których nazwy są ustalone przez express. Można tworzyć też własne zmienne, np.:

```
app.set('tytul', 'Moja stronka'); // utworzenie zmiennej tytul
var zmienna = app.get('tytul');   // odczyt zmiennej
console.log(zmienna);             // wypisanie wartość, Moja stronka
```

Niektóre zmienne pozwalające ustawić działanie aplikacji express:

- `view` - nazwa katalogu (String) albo tablicy katalogów (Array) gdzie mieszczą się szablony widoków. W przypadku tablicy katalogi są przeglądanie w kolejności występowania w tablicy. Domyślna wartość to: `process.cwd() + '/views'`
- `view engine` - nazwa określa domyślny silnik, który będzie użyty jeśli nie określono jawnie później.

```
// ustawienie silnika dla widoków
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

2.2 Moduły w Node.js

Node.js - eksportowanie modułów

Moduły w Node.js są związane z plikami - jeden plik to jeden moduł. Wczytywanie modułów odbywa się za pomocą polecenia `require()`, np.:

kolo.js

```
var PI = Math.PI;

exports.pole = function (r) {
  return PI * r * r;
};
module.exports.obwod = function (r) {
  return 2 * PI * r;
};
```

main.js

```
var kolo = require('./kolo.js');
console.log('Pole koła wynosi ' + kolo.pole(2));
console.log('Obwód koła wynosi ' + kolo.obwod(3));
```

```
Pole koła wynosi 12.566370614359172
Obwód koła wynosi 18.84955592153876
```

Zmienna PI jest prywatna, widoczna tylko w zakresie modułu.

Node.js - eksportowanie modułów

Moduł jako funkcja (np. konstruktor) albo kompletny obiekt:

kwadrat.js

```
// przypisanie do exports nie zmieni modułu, trzeba użyć module.exports
module.exports = function(a) {
  return {
    pole: function() {
      return a * a;
    }
  };
}
```

main.js

```
var kwadrat = require('./kwadrat.js');
var mojKwadrat = kwadrat(4);
console.log('Pole kwadratu wynosi ' + mojKwadrat.pole());
```

```
Pole kwadratu wynosi 16
```

Node.js - eksportowanie modułów

Osobny moduł połączenia się z bazą danych:

db.js

```
var MongoClient = require( 'mongodb' ).MongoClient;
var _db;
var _client;
module.exports = {
 connectToServer: function( dbName, callback ) {
  MongoClient.connect("mongodb://localhost:27017/local", function(err, client) {
    _client = client;
    console.log("Nawiązano połączenie z serwerem");
    _db = client.db(dbName);
    return callback( err );
  } );
 },
 getDb: function() {
  return _db;
 },
 closeConnection: function() {
  _client.close();
 }
};
```

Node.js - eksportowanie modułów

Połączenie się z bazą danych:

app.js

```
...
var db = require( './db.js' );

db.connectToServer("local", function( err ) {
  assert.equal(null, err);  // console.log(err);
} );

app.get('/liczba-stud', function (req, res) {
  var db2 = db.getDb();
  var collection = db2.collection('studenci');
  collection.count(function(err, count) {
    assert.equal(err, null);
    res.send('Liczba studentów: ' + count);
  });
});

app.get('/close', function (req, res) {
  db.closeConnection();
  res.send('Zamykanie połączenia...');
});
```

2.3 Silniki szablonów

Silniki szablonów

Zarówno express-generator jak i WebStorm oferują do wyboru jeden z kilku silników szablonów: `ejs|hbs|hjs|jade|pug|twig|vash` Domyślnym silnikiem szablonów jest Jade.

- Jade, obecnie Pug - `https://pugjs.org`
- EJS - `http://ejs.co/`
- Handlebars - `http://handlebarsjs.com/`
- Hogan.js - `http://twitter.github.io/hogan.js/`

Silniki szablonów - EJS

EJS - `<% Effective JavaScript %>` `http://ejs.co/`

Cechy:

- wyrażenia umieszczanie w znacznikach `<% %>`
- modyfikacja znaczników HTML: `<%= %>` (możliwość konfiguracji funkcji zastępowania znaczników)
- brak modyfikacji znaczników, wyjście nieprzetworzone: `<%- %>`
- tryb usuwania znaków nowej linii, kończy się znacznikiem `-%>`
- tryb usuwania białych znaków, oganieczniki: `<%_ _%>`
- dwolne ograniczniki, np. użycie `<? ?>` zamiast `<% %>`
- wspiera dziedziczenie szablonów,
- wsparcie szablonów po stronie serwera i klienta (przeglądarka),
- statyczne buforowanie pośredniego JavaScriptu,
- statyczne buforowanie szablonów,
- kompiluje się z systemem widoków Expressa.

Silniki szablonów - Handlebars

Handlebars - `http://handlebarsjs.com/`

Cechy:

- w dużej mierze kompatybilny z Mustache (`http://mustache.github.io/`), zawiera dodatkową funkcjonalność
- wyrażenia umieszczamy w parze znaków `{{ ... }}`
- komentarze w szablonie: `{{!-- --}}` albo `{{! }}`
- kompilacja szablonu w przeglądarce albo wcześniejsza prekompilacja (szybsze działanie strony)
- zawiera przekształcanie znaczników HTML (dla bezpieczeństwa, `{{napis}}`), można je wyłączyć korzystając z zapisu `{{{napis}}}`,
- przy użyciu bloku rozszerzeń można dodać obsługę pętli, instrukcji warunkowych, itp.
- w znaczniku szablonu można używać ścieżek, dostępu do zagnieżdżonych pól obiektu,
- pozwala na używanie literałów,
- pozwala na tworzenie szablonów cześciowych wielokrotnego użytku.

Silniki szablonów - Hogan.js

Hogan.js
JavaScript templating from Twitter.

`http://twitter.github.io/hogan.js/`

Cechy:

- kompilator dla silnika szablonów Mustache (`http://mustache.github.io/`),
- zintegrowany z Express - pakiet hogan-express,
- wspiera podział szablonów na cześci (działa podobnie do include),
- pozwala na tworzenie układów (layout) często występujących na stronie - coś jak szablon rodzica,
- wspiera buforowanie - aplikacja działa szybciej unikając niepotrzebnego renderowania,
- inaczej niż Handlebars, Hogan.js nie zawiera dodatkowych funkcjonalności, np. pętli czy instrukcji warunkowych,
- pozwala na tworzenie własnych filtrów.

Silniki szablonów - Pug (Jade)

Pug, wcześniej Jade (zastrzeżony znak towarowy)

`https://pugjs.org` tutorial - `http://learnjade.com/tour/intro/`

Cechy Pug/Jade:

- słowo Pug to mops :)
- silnik szablonów wysokiej wydajności,
- zaimplementowany w JavaScript dla Node.js i dla przeglądarek,
- pełna integracja z Express.js jako jeden z wspieranych silników szablonów,
- zespół pracował nad niekompatybilną wersją Jade 2.0.0 kiedy zmieniono nazwę na Pug,
- instalacja Pug oznacza używanie nowej wersji 2.0.0 (ciągle jeszcze beta),
- nazwa `jade` jest ciągle wspierana w repozytoriach npm.

3 Tworzenie własengo API

3.1 Biblioteka mongoose

Biblioteka mongoose

mongoose
elegant mongodb object modeling for node.js

`http://mongoosejs.com/`

Mongoose:

- biblioteka dla Node.js udostępniająca mapowanie obiektowe (podobne do ORM) z interfejsem znanym z Node.js,
- opiera się na Object Data Mapping (ODM) - zmiana danych z bazy do obiektów JavaScript, któreh można użyć w aplikacji,
- dostarcza gotowe rozwiązanie do modelowania danych aplikacji,
- zawiera wbudowane rzutowanie typów, walidację, budowanie zapytań, gotowe, praktyczne rozwiązania dla logiki biznesowej i wiele innych.

Schemat

- Wszystko w Mongoose zaczyna się od schematu.
- Każdy schemat przekłada się na kolekcje w MongoDB, określa budowę i zawartość dokumentów w tej kolekcji.
- Przykład:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  }
});
```

Schemat

- Każdy klucz schematu określa pole dokumentu w bazie i typ tego pola.
- W ten sposób można również definiować zagnieżdżone obiekty dokumentu.
- Możliwe jest dodanie pól do schematu już po jego utworzeniu (`Schema.add()`).
- Dozwolone typy danych: String, Number, Date, Buffer, Boolean, Mixed, ObjectId, Array.
- Schematy określają/definiują też:
  - nasze własne metody dla modelu (opróczy tych wbudowanych),
  - statyczne metody dla modelu,
  - dodatkowe i złożone indeksy utworzone dla kolekcji.
- Do schematów można też dodawać wirtualne pola, które np. będą zwracać napis z kilku pól ale samo pole wirtualne nie będzie odzwierciedlone w bazie.

Model

- Modele są specjalnymi konstruktorami tworzonymi na bazie schematu.
- Instancje modelu reprezentują dokumenty, które mogą być odczytane i zapisane do bazy.
- Wszystkie operacje na dokumentach w bazie są wykonywane za pośrednictwem modelu.
- Konstruktor ma dwa parametry:
  - **liczbę pojedynczą** nazwy kolekcji, w której będą dane,
  - schemat, na bazie którego powstanie model,

```
var schema = new mongoose.Schema({ name: 'string', size: 'string' });
var Tank = mongoose.model('Tank', schema);
```

- pozyższy kod stworzy w bazie kolekcję **tanks**.
- na modelu można robić chyba wszystko: pobierać, tworzyć, usuwać, aktulizować, ...

Instalacja, połączenie, schemat i model

- Instalujemy mongoose w projekcie: `npm i mongoose --save`
- Dołączamy mongoose do projektu i łączymy się z bazą danych, np. `test` (zostanie utworzona jeśli takiej nie było):

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

- Sprawdzamy czy połączenie się udało:

```
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'błąd połączenia...'));
db.once('open', function() {
  // połączenie udane!
});
```

- Schemat i model:

```
var friendSchema = mongoose.Schema({
  nazwa: String
});

var Friend = mongoose.model('Friend', friendSchema);
```

Tworzenie dokumentów, własna metoda i jej użycie

- Na bazie modelu tworzymy dokumenty (zawierają pola i typy jak w schemacie):

```
var franek = new Friend({ nazwa: 'Franek' });
console.log(franek.nazwa); // 'Franek'
```

- Przyjaciele mogą się witać - zobaczmy, jak dodać funkcjonalność do naszych dokumentów:

```
// metody należy dodać do schematu ZANIM utworzy się z niego model
friendSchema.methods.sayHello = function () {
  var powitanie = this.nazwa
      ? "Cześć, mam na imię " + this.nazwa
      : "Witaj, nie wiem jak się nazywam ...";
  console.log(powitanie);
}
```

- Funkcja dodana do pola `methods` schematu i wykorzystana w modelu jest dostępna w każdym utworzonym dokumencie

```
var jola = new Friend({ nazwa: 'Jolanta' });
jola.sayHello(); // "Cześć, mam na imię Jolanta"
```

Operacje wykonywane na modelu I

Wybrane metody wykonywane na modelu:

- `increment()` - zwiększa o jeden wersję dokumentu,
- `model(name)` - zwraca dodatkową instancję modelu,
- `remove([fn])` - usuwa bieżący dokument z bazy danych,
- `save(...)` - zapisuje bieżący dokument w bazie,
- `count(conditions, [callback])` - zwraca liczbę dopasowanych dokumentów,
- `create(doc(s), [callback])` - skrót dla wygodniejszego tworzenia dokumentów, wykonuje: `new MyModel(doc).save()` dla każdego dokumentu w `docs`,
- `deleteMany(conditions, [callback])` - usuwa wszystkie dopasowane dokumenty,
- `deleteOne(conditions, [callback])` - usuwa pierwszy dopasowany dokument,

Operacje wykonywane na modelu II

Wybrane metody wykonywane na modelu:

- `find(conditions, [projection], [options], [callback])` - zwraca dokumenty spełniające kryterium,
- `findById(id, [projection], [options], [callback])` - zwraca jeden dokument o podanym id, niemal równoznaczne z `findOne({ _id: id })`,
- `findByIdAndRemove(id, [options], [callback])` - usuwa dokument o podanym id,
- `findByIdAndUpdate(id, [update], [options], [callback])` - aktualizuje dokuemnt o podanym id,
- `findOne([conditions], [projection], [options], [callback])` - zwraca pierwszy dokument spełniający kryterium,
- `findOneAndRemove(conditions, [options], [callback])` - usuwa pierwszy dopasowany dokument,
- `findOneAndUpdate([conditions], [update], [options], [callback])` - aktualizuje pierwszy dopasowany dokument,

Operacje wykonywane na modelu III

Wybrane metody wykonywane na modelu:

- `insertMany(doc(s), [options], [callback])` - sprawdza poprawność dokumentów (docs) i jeśli są poprawne dodaje je wszystkie do bazy w jednym zapytaniu,
- `remove(conditions, [callback])` - usuwa dokument(y) spełniający kryterium,
- `replaceOne(conditions, doc, [options], [callback])` - zastępuje dokument spełniający kryterium, różni się od `update()` tym, że nie pozwala na operatory atomowe, np. `$set`,
- `update(conditions, doc, [options], [callback])` - aktualizuje dokumenty spełniające kryterium,
- `updateOne(conditions, doc, [options], [callback])` - aktualizuje pierwszy dokument spełniający kryterium,
- atrybuty modelu: `db`, `collection`, `schema` - zwraca: połączenie, kolekcję czy schemat, z którego korzysta model.

Przykłady wbudowanych metod

- Zapis dokumentów w bazie, metoda `save()`:

```
jola.save(function (err, jola) {  // pierwszy argument odpowiada za błędy
  if (err) return console.error(err);
  jola.sayHello();
});
```

- Odczyt dokumentów zapisanych w bazie, metoda `find()`:

```
Friend.find(function (err, przyjaciele) {
  if (err) return console.error(err);
  for(var i=0; i<przyjaciele.length; i++) {
    console.log('%s', przyjaciele[i].nazwa);
  }
})
```

- Wyszukiwanie można wykonać po dowolnym polu: find({ nazwa: /^Jol/ }

```
Friend.find({ nazwa: /^Jol/ }, function (err, przyjaciele) {
  if (err) return console.error(err);
  console.log("====================\n");
  for(var i=0; i<przyjaciele.length; i++) {
    console.log('%s', przyjaciele[i].nazwa);
  }
}); // lista przyjaciół nazywających sie Jol*
```

3.2 API dla kolekcji użytkowników

API dla kolekcji użytkowników

Adresy dostępne w API i ich znaczenie:

Adres (URI)	Metoda	działanie
`/users`	GET	lista wszystkich uzytkowników
`/users/:id`	GET	użytkownik o podanym ID
`/users`	POST	dodanie użytkownika do kolekcji
`/users/:id`	PUT	aktualizacja danych użytkownika o padanym ID
`/users/delete-all`	DELETE	usunięcie wszystkich użytkowników z kolekcji
`/users/:id`	DELETE	usunięcie użytkownika o padanym ID

Z głównego pliku aplikacji, `app.js`, interesuje nas:

app.js

```
var users = require('./routes/users');
...

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
...

app.use('/users', users);
```

Przygotowanie, połączenie z bazą, schemat i model

Przygotowanie mongoose, połączenie z bazą, schemat i model:

routes/users.js

```
var mongoose = require('mongoose');
...

// wszystkie dane będą w kolekcji users bazy ob-tur
mongoose.connect('mongodb://localhost/ob-tur', function(err) {
    if(err) {
        console.log('błąd połączenia', err);
    } else {
        console.log('połączenie udane');
    }
});
```

```
var UsersSchema = new mongoose.Schema({
  username: String,
  password: String,
  admin: { type: Boolean, default: false }
});

var Users = mongoose.model('users', UsersSchema);
...
```

Pobieranie danych, metoda GET

Pobieranie całej kolekcji:

routes/users.js

```
/* GET /users */
router.get('/', function(req, res, next) {
  Users.find(function (err, docs) {
    if (err) return next(err);
    res.json(docs);
  });
});
```

Pobieranie wybranego użytkownika:

routes/users.js

```
/* GET /users/:id */
router.get('/:id', function(req, res, next) {
  Users.findById(req.params.id, function (err, doc) {
    if (err) return next(err);
    res.json(doc);
  });
});
```

Dodawanie i aktualizacja danych, metody POST i PUT

Dodawanie nowego dokumentu do kolekcji:

routes/users.js

```
/* POST /users */
router.post('/', function(req, res, next) {
  Users.create(req.body, function (err, doc) {
    if (err) return next(err);
    // console.log(JSON.stringify(doc));
    res.json(doc);
  });
});
```

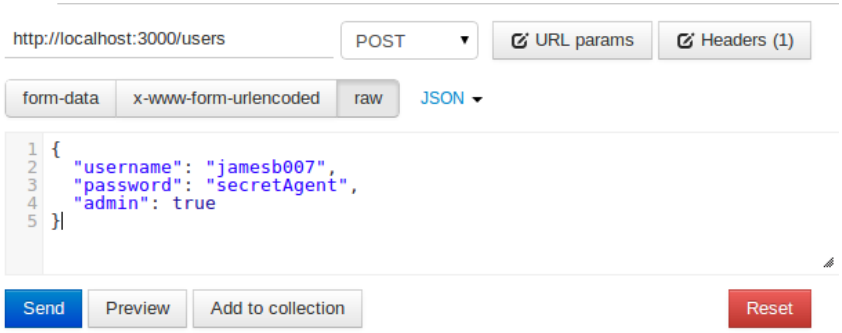
Aktualizacja wybranego użytkownika:

routes/users.js

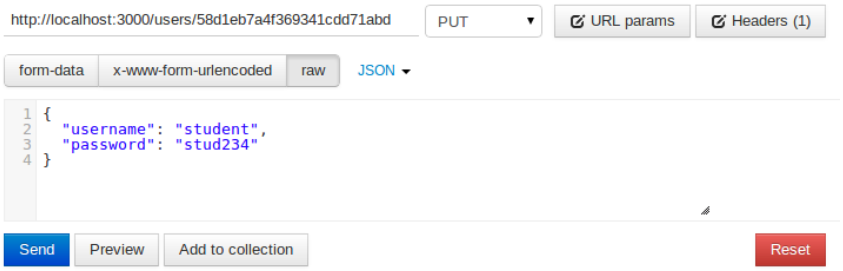
```
/* PUT /users/:id */
router.put('/:id', function(req, res, next) {
  Users.findByIdAndUpdate(req.params.id, req.body, function (err, doc) {
    if (err) return next(err);
    res.json(doc);
  });
});
```

Dodawanie i aktualizacja danych, metody POST i PUT

Dodawanie użytkownika:



Aktualizacja użytkownika o podanym ID:



Usuwanie danych z kolekcji, metoda DELETE

Usuwanie wszystkich dokumentów z kolekcji:

routes/users.js

```
/* DELETE /users/delete-all */
router.delete('/delete-all', function(req, res, next) {
  Users.remove({}, function (err, writeRes) {
    if (err) return next(err);
    // console.log(writeRes);
    res.send(writeRes);
  });
});
```

Usuwanie wybranego użytkownika z kolekcji:

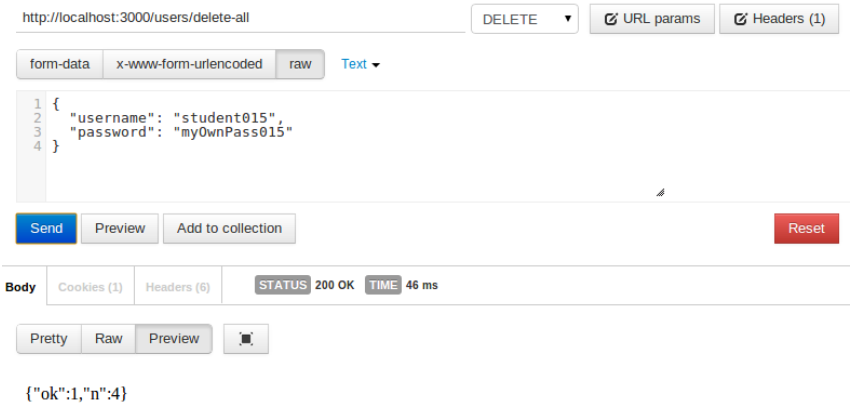
routes/users.js

```
/* DELETE /users/:id */
router.delete('/:id', function(req, res, next) {
  Users.findByIdAndRemove(req.params.id, function (err, doc) {
    if (err) return next(err);
    res.json(doc);
  });
});
```

Usuwanie danych z kolekcji, metoda DELETE

Usuwanie użytkownika o podanym ID:

Usuwanie wszystkich dokumentów z kolekcji:

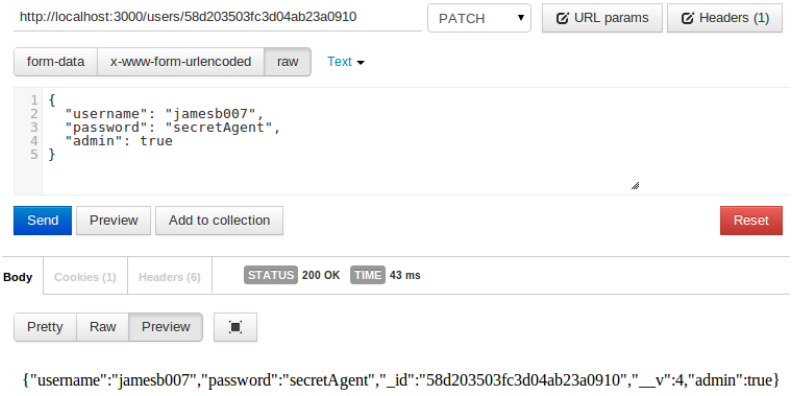


Zwiększanie wersji dokumentu, metody PATCH

Zwiększanie wersji dokumentu użytkownika o podanym ID:

routes/users.js

```
/* PATCH /users/:id */
router.patch('/:id', function(req, res, next) {
  Users.findById(req.params.id, function (err, doc) {
    if (err) return next(err);
    doc.increment();
    doc.save(function (err, savedDoc) {
      if (err) return next(err);
      res.json(savedDoc);
    });
  });
});
```



4 Źródła

Źródła

- https://nodejs.org/en/
- https://en.wikipedia.org/wiki/Node.js
- http://expressjs.com/
- https://en.wikipedia.org/wiki/Express.js
- http://www.tutorialspoint.com/nodejs/index.htm
- https://www.npmjs.com/
- https://github.com/libuv/libuv
- https://en.wikipedia.org/wiki/Callback_(computer_programming)
- http://jade-lang.com/
- https://pugjs.org
- http://mongoosejs.com/