Express.js i własne API - pomoc do lab02

Tworzenie serwisów Web 2.0

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

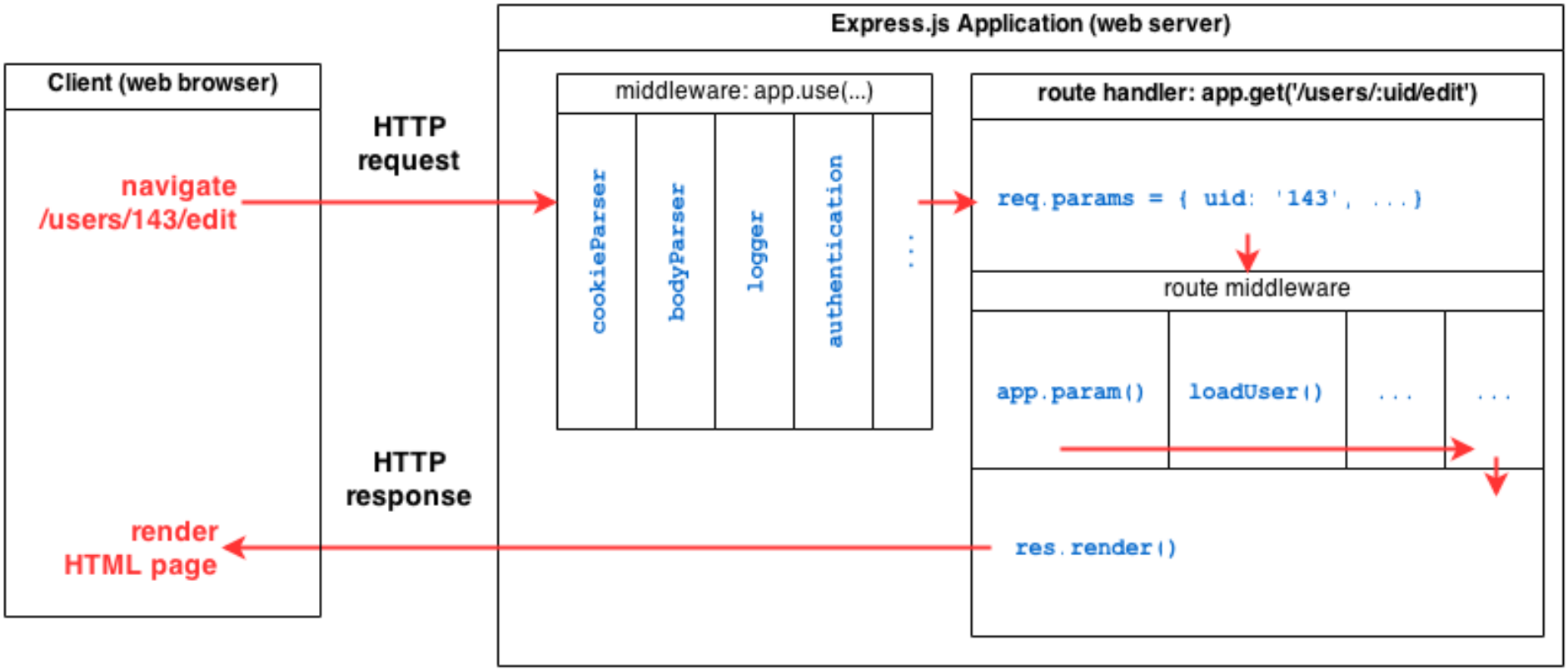
Politechnika Częstochowska
Instytut Informatyki Teoretycznej i Stosowanej

25 lutego 2018

MEAN Stack

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)
})
```

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 ...
```

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 z parametrami

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

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');
```

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*
```

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);
  });
});
```

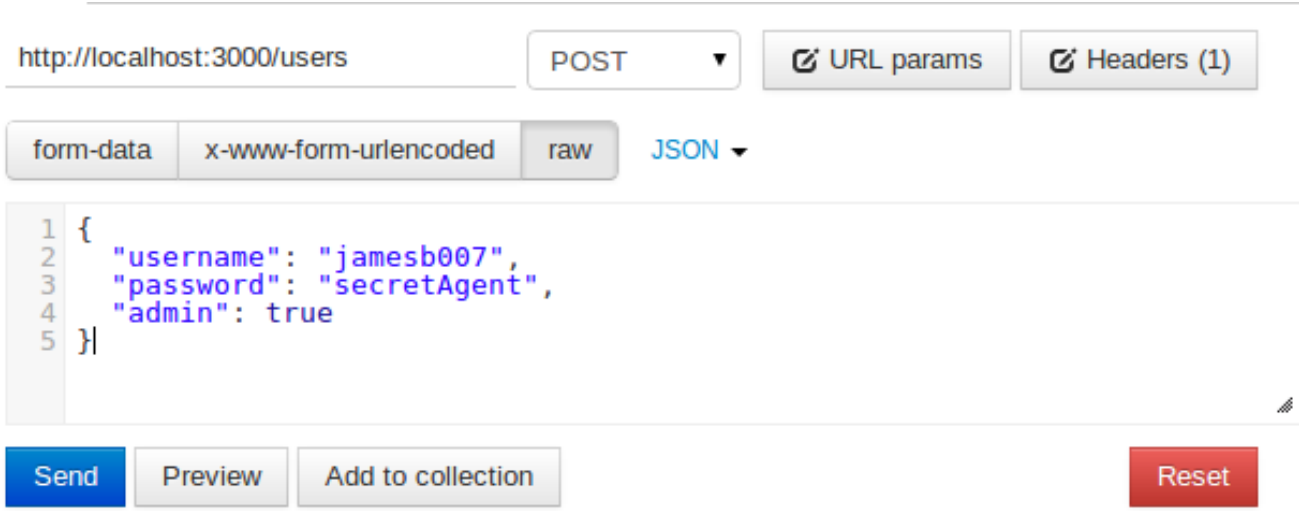
Aktualizacja wybranego użytkownika:

routes/users.js

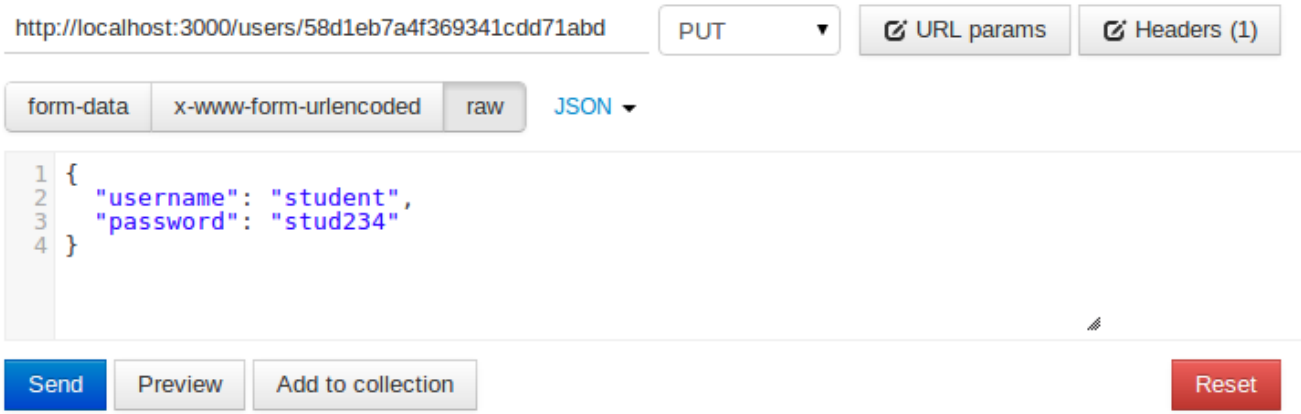
```
/* PUT /users/:id */
router.put('/:id', function(req, res, next) {
  Users.findByIdAndUpdate(req.params.id, req.body, function (err, doc) {
    if (err) return next(err);
    res.json(doc);
  });
});
```

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:

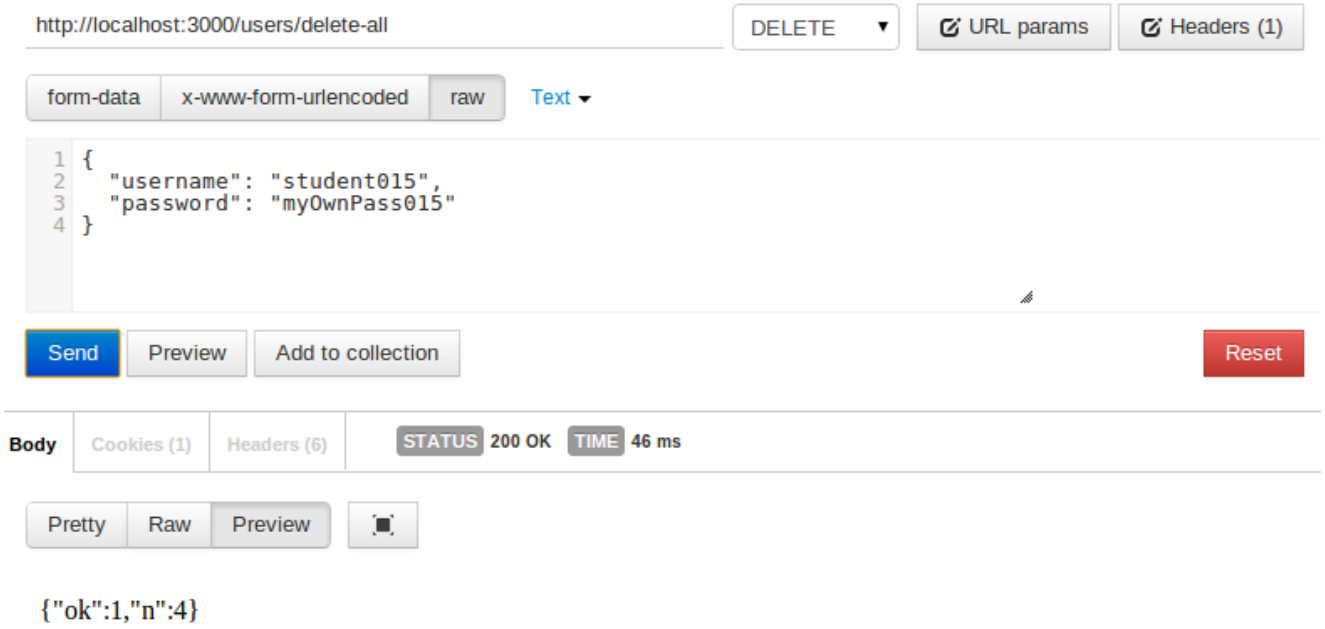
routes/users.js

```
/* 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);
    });
  });
});
```