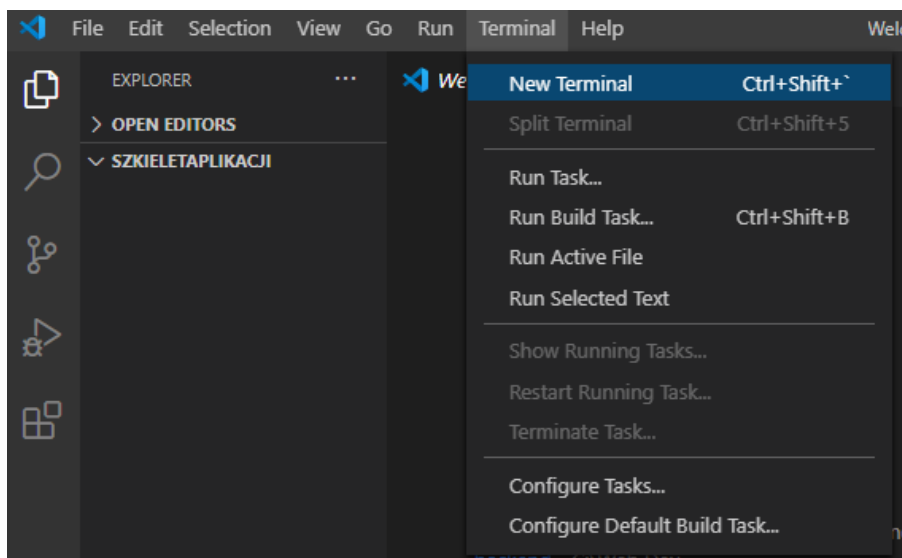
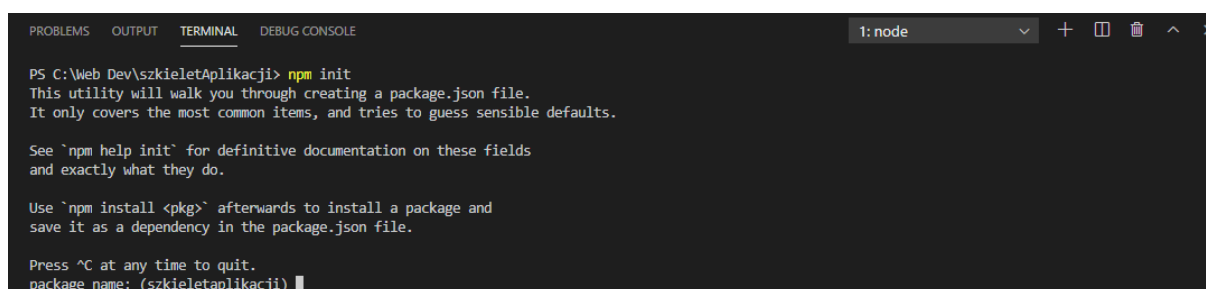


# NodeJs + Express – szkielet aplikacji

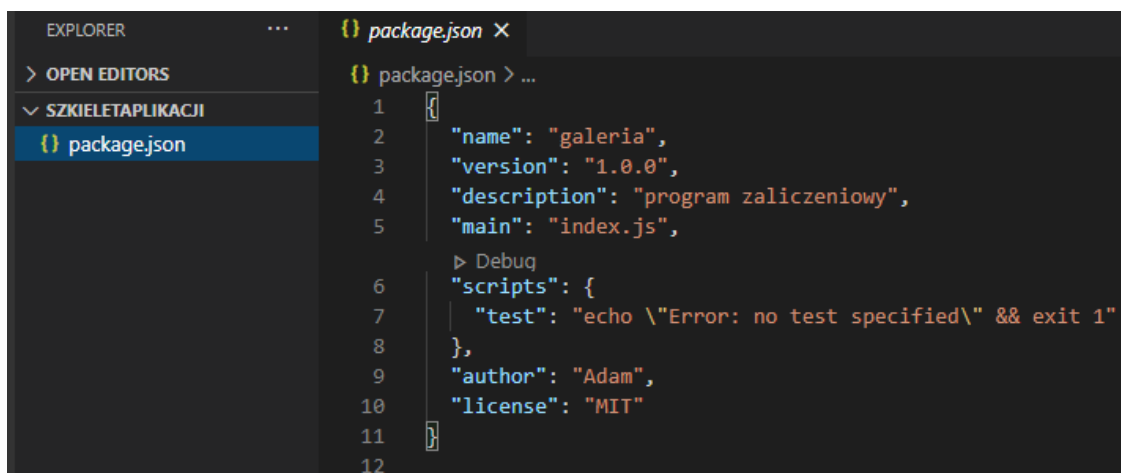
Przed rozpoczęciem pracy warto uruchomić Visual Studio Code i uruchomić terminal.



Aby zacząć projekt należy skorzystać z polecenia `npm init`



Po uruchomieniu polecenia należy ustalić nazwę aplikacji, opis, licencję, resztę poleceń można pominąć. Po wykonaniu powinien utworzyć się plik `package.json`



Plik ten nie musi być generowany za pomocą polecenia `npm init`, można równie dobrze stworzyć go samemu.

Kolejnym krokiem jest zainstalowanie pakietu Express. Do instalacji standardowo używamy npm.

```
PS C:\Web Dev\szkieletAplikacji> npm i express
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN galeria@1.0.0 No repository field.

+ express@4.17.1
added 50 packages from 37 contributors and audited 50 packages in 3.566s
found 0 vulnerabilities

PS C:\Web Dev\szkieletAplikacji>
```

W tym przypadku polecenie zostało uruchomione poleceniem skróconym, można stosować „i” zamiast „install”. W tym momencie zostały utworzony plik package-lock.json dla nas nie istotny, oraz folder node\_modules zawierający potrzebne moduły. W pliku package.json został dodany parametr dependencies.

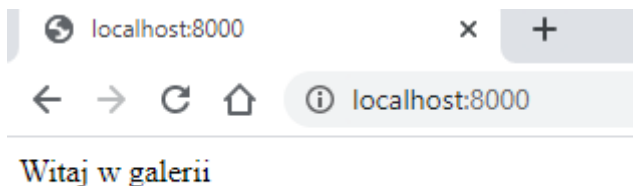
```
{
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Teraz kolejnym krokiem będzie utworzenie pliku index.js. Oczywiście można nazwać go inaczej jednak trzeba wówczas zmienić nazwę w pliku package.json.

```
1  const express = require('express') //zamontowanie paczki express
2  const port = 8000 //port na którym będzie działać aplikacja
3  const app = express() //utworzenie obiektu express
4  app.set('view engine', 'html');
5
6  app.get('/', (req,res) => {      //instrukcja get
7    res.send('Witaj w galerii')    //wysłanie napisu na stronę
8  })
9
10 app.listen(port) //nasłuchiwanie portu, jest konieczne do działania aplikacji
```

W powyższym kodzie app.get jest typową funkcją która przyjmuje req i res, nazwy skrócone od request i response. W tym przykładzie widać również nową formę zapisu funkcji przy użyciu strzałki. Linijka 6 oznacza że dla aplikacji Express dla metody Get, dla ścieżki głównej „/” czyli „localhost:8000/” i argumentami req i res wyślij napis „Witaj w galerii”. Taka instrukcja powinna wyświetlić napis „Witaj w galerii” po uruchomieniu poleceniem node index.js.

```
PS C:\Web Dev\szkieletAplikacji> node index.js
```

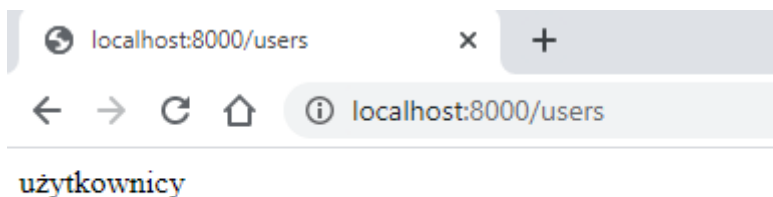


The screenshot shows a web browser window with the address bar set to localhost:8000. The page content displays the text "Witaj w galerii" in a simple font.

W tym miejscu mamy obsługę części CRUD a dokładnie Get, nie jest to jednak domyślne miejsce tego, ale przetestujmy działanie innych ścieżek.

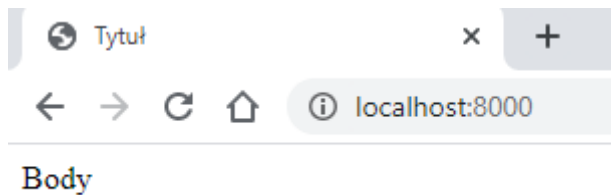
```
1  const express = require('express') //zamontowanie paczki express
2  const port = 8000 //port na którym będzie działać aplikacja
3  const app = express() //utworzenie obiektu express
4  app.set('view engine', 'html');
5
6  app.get('/', (req,res) => {      //instrukcja get
7    res.send('Witaj w galerii')    //wysłanie napisu na stronę
8  })
9
10 app.get('/users', (req,res) => {
11   res.send('użytkownicy')
12 })
13
14 app.get('/comments', (req,res) => {
15   res.send('komentarze')
16 })
17
18
19 app.listen(port) //nasłuchiwanie portu, jest konieczne do działania aplikacji
```

Aby te ścieżki zadziałały należy przerwać działanie serwera (Ctrl+c) i uruchomić na nowo komendą node index.js.



Generowanie napisu nie jest naszym celem, powinniśmy mieć możliwość wczytania pliku html. Dlatego res.send z linijki 7 możemy zmienić na sendFile, co pozwoli nam wczytać plik, który należy uprzednio stworzyć.

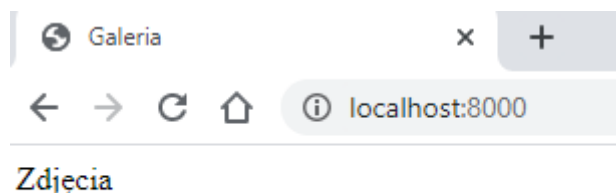
```
1  const express = require('express') //zamontowanie paczki express
2  const port = 8000 //port na którym będzie działać aplikacja
3  const app = express() //utworzenie obiektu express
4
5
6  app.get('/', (req,res) => {      //instrukcja get
7    res.sendFile(__dirname+'/index.html')
8  })
9
10 app.get('/users', (req,res) => {
11   res.send('użytkownicy')
12 })
13
14 app.get('/comments', (req,res) => {
15   res.send('komentarze')
16 })
17
18
19 app.listen(port) //nasłuchiwanie portu, jest konieczne do działania aplikacji
```



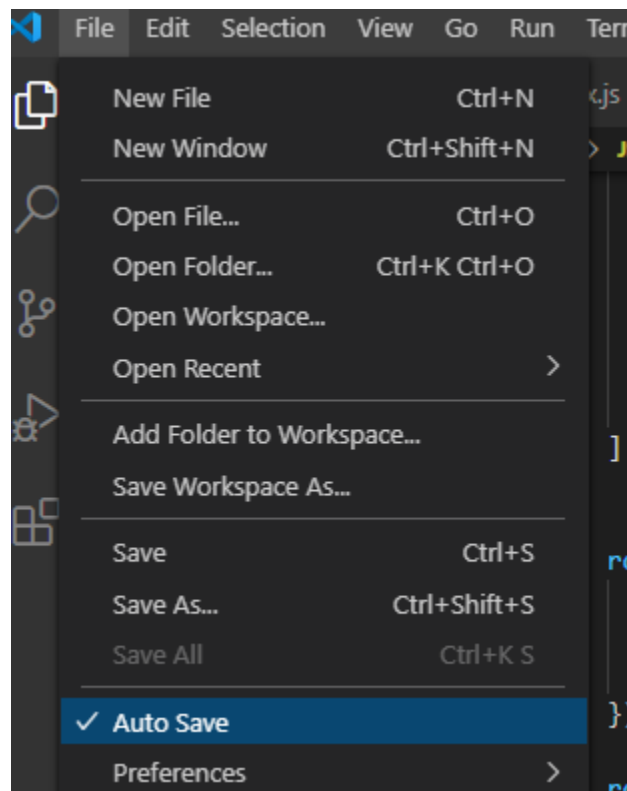
W tym momencie potrzebujemy przekazać coś do widoku, dlatego możemy skorzystać z paczki hbs, oczywiście jest do wyboru dużo innych paczek generowania widoku, ale w tym momencie możemy skorzystać z hbs (<https://www.npmjs.com/package/hbs>). Aby zainstalować paczkę wystarczy uruchomić instrukcję `npm install hbs`. Aby paczka zadziałała prawidłowo należy postąpić według instrukcji, a następnie utworzyć plik w folderze views.

```
EXPLORER
  OPEN EDITORS
  SZKIELETAPLIKACJI
    node_modules
    views
      index.hbs
    index.html
    index.js
    package-lock.json
    package.json
  package.json JS index.js index.hbs X
  views > index.hbs > html
  1 <!doctype html>
  2
  3 <html lang="en">
  4 <head>
  5   <meta charset="utf-8">
  6   <title>{{Title}}</title>
  7   <meta name="description">
  8   <meta name="author">
  9 </head>
  10
  11 <body>
  12   {{Body}}
  13 </body>
  14 </html>
```

```
JS index.js > ...
1 const express = require('express') //zamontowanie paczki express
2 const port = 8000 //port na którym będzie działać aplikacja
3 const app = express() //utworzenie obiektu express
4 app.set('view engine', 'hbs')
5
6
7 app.get('/', (req, res) => { //instrukcja get
8   res.render('index', {
9     Title: "Galeria",
10    Body: "Zdjęcia"
11  })
12 })
```



Możemy nieco pozmienić aplikację i dodać więcej udogodnień. Program Visual Studio Code umożliwia automatyczny zapis. Dzięki temu nie trzeba zapisywać po każdej zmianie kodu, natomiast nasz serwer dalej trzeba odświeżać aby zaktualizować dane.



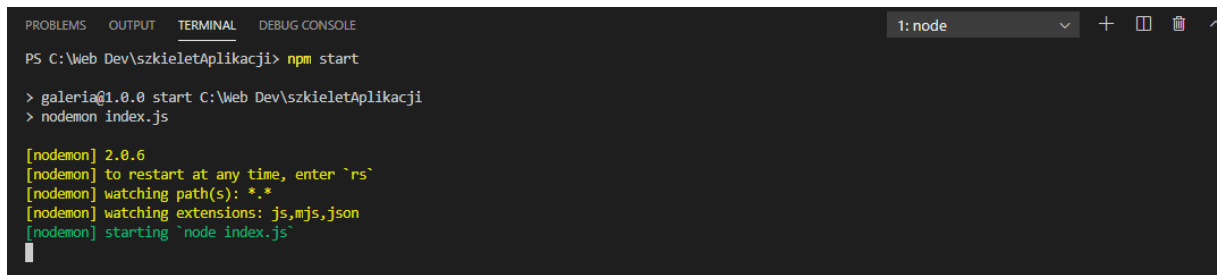
Aby nie trzeba było od nowa uruchamiać serwera po każdej zmianie, możemy wykorzystać moduł nodemon instalowany poleceniem: `npm install --save-dev nodemon`

Aby uruchomić serwer należy dodać skrypt do pliku `package.json`

```
{ package.json > {} scripts > start
1  {
2    "name": "backend",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "start": "nodemon index.js"
8    },
9    "author": "",
10   "license": "ISC",
11   "dependencies": {
12     "body-parser": "^1.19.0",
13     "express": "^4.17.1",
14     "hbs": "^4.1.1"
15   },
16   "devDependencies": {
17     "nodemon": "^2.0.6"
18   }
19 }
20
```

Kolejnym krokiem przydatnym do testowania komunikacji klient-serwer będzie dodanie modułu body-parser zgodnie z instrukcją: <https://www.npmjs.com/package/body-parser>. Po wykonaniu instalacji, `package.json` również uległ zmianie i do dependencies został dodany body-parser.


Po instalacji tych modułów, aby uruchomić aplikację wystarczy skorzystać z polecenia `npm start`, które uruchomi `index.js` za pomocą `nodemon`:



```
PS C:\Web Dev\szkieletAplikacji> npm start
> galeria@1.0.0 start C:\Web Dev\szkieletAplikacji
> nodemon index.js

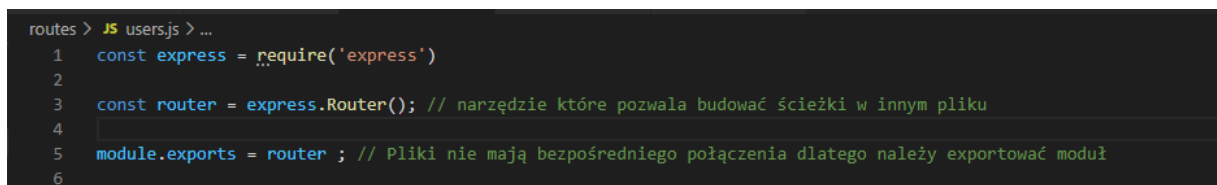
[nodemon] 2.0.6
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
```

Aby skorzystać z modułu `body-parser` należy dodać moduł oraz ustawić używanie modułu dla plików `json` (dwie linijki z komentarzami)



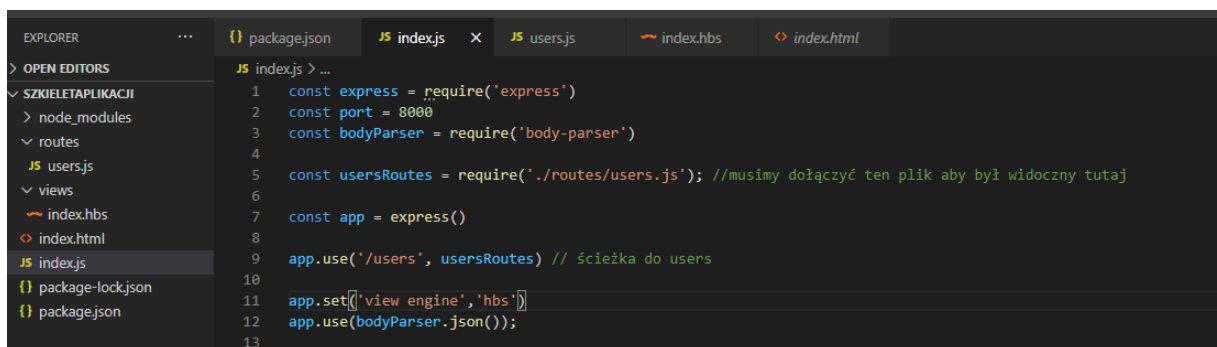
```
JS index.js > app.get('/') callback
1  const express = require('express')
2  const port = 8000
3  const bodyParser = require('body-parser') //zamontowanie paczki body-parser
4  const app = express()
5
6  app.set('view engine', 'hbs')
7  app.use(bodyParser.json()); //ustawienie aplikacji aby korzystała z json
8
9  app.get('/', (req, res) => {
10     res.render('index', {
11         Title: "Galeria",
12         Body: "Zdjęcia"
13     })
14 })
```

Teraz możemy stworzyć kolejne metody np. `Post`, ale w całej aplikacji będzie ich dużo dlatego warto posprzątać. Stworzymy folder „`routes`” a wewnątrz plik `users.js`



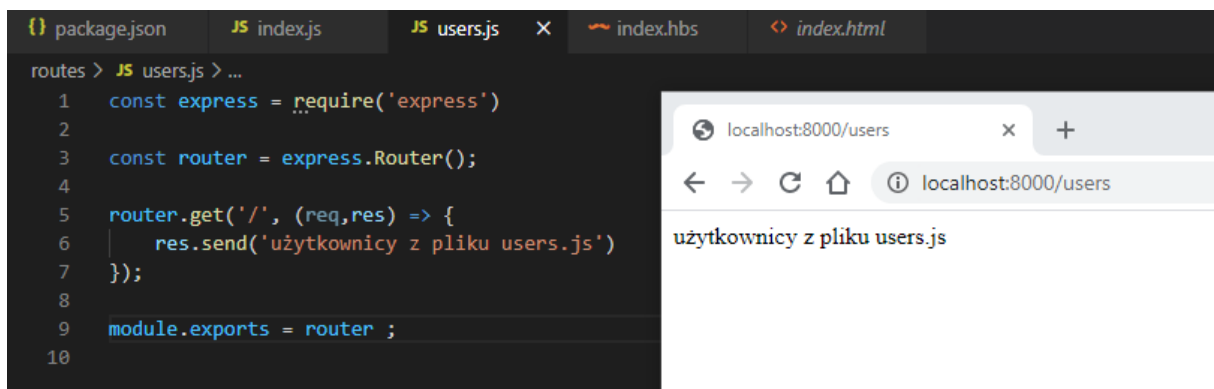
```
routes > JS users.js > ...
1  const express = require('express')
2
3  const router = express.Router(); // narzędzie które pozwala budować ścieżki w innym pliku
4
5  module.exports = router; // Pliki nie mają bezpośredniego połączenia dlatego należy exportować moduł
6
```

Dzięki temu będzie można uporządkować poszczególne ścieżki tak aby po „`/users`” można było bazować wewnątrz pliku `users.js` już bez używania nazwy „`users`”. Musimy zatem dodać pewnie instrukcje do pliku `index.js` aby ta ścieżka była widoczna.



```
JS index.js > ...
1  const express = require('express')
2  const port = 8000
3  const bodyParser = require('body-parser')
4
5  const usersRoutes = require('./routes/users.js'); //musimy dołączyć ten plik aby był widoczny tutaj
6
7  const app = express()
8
9  app.use('/users', usersRoutes) // ścieżka do users
10
11 app.set('view engine', 'hbs')
12 app.use(bodyParser.json());
13
```

Aby przetestować ścieżkę wystarczy przesłać cokolwiek, np. napis za pomocą routera i funkcji get:



```
routes > JS users.js > ...
1  const express = require('express')
2
3  const router = express.Router();
4
5  router.get('/', (req,res) => {
6    res.send('użytkownicy z pliku users.js')
7  });
8
9  module.exports = router ;
10
```

localhost:8000/users

użytkownicy z pliku users.js

Możemy również stworzyć tymczasowe dane do testów, ponieważ nie posiadamy jeszcze bazy danych, ale na podstawie naszego API wiemy jakie metody powinniśmy utworzyć. Przykład wzięty z PetStore:

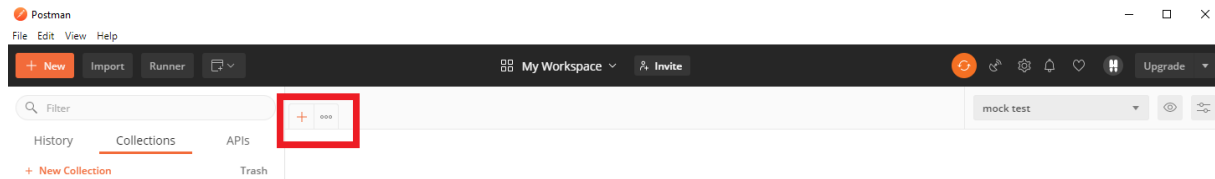
user		Operations about user	Find out more about our store
POST	/user	Create user	
POST	/user/createWithArray	Creates list of users with given input array	
POST	/user/createWithList	Creates list of users with given input array	
GET	/user/login	Logs user into the system	
GET	/user/logout	Logs out current logged in user session	
GET	/user/{username}	Get user by user name	
PUT	/user/{username}	Updated user	
DELETE	/user/{username}	Delete user	

Zatem korzystając z API utworzonego na poprzednich zajęciach możemy wstępnie zaimplementować metody, pamiętając że ścieżki już teraz zaczynają się od „/user” dlatego nie należy dopisywać tego wewnątrz pliku router/users.js. Do testu przygotujmy podstawową funkcję Get i Post która będzie przyjmowała i wyświetlała obiekty z tablicy sformatowanej podobnie do Json. Aby czytelnie widzieć dane wejściowe i wyjściowe można skorzystać z dodatków do przeglądarki. W przypadku chrome należy wybrać jeden z wielu dostępnych formaterów.

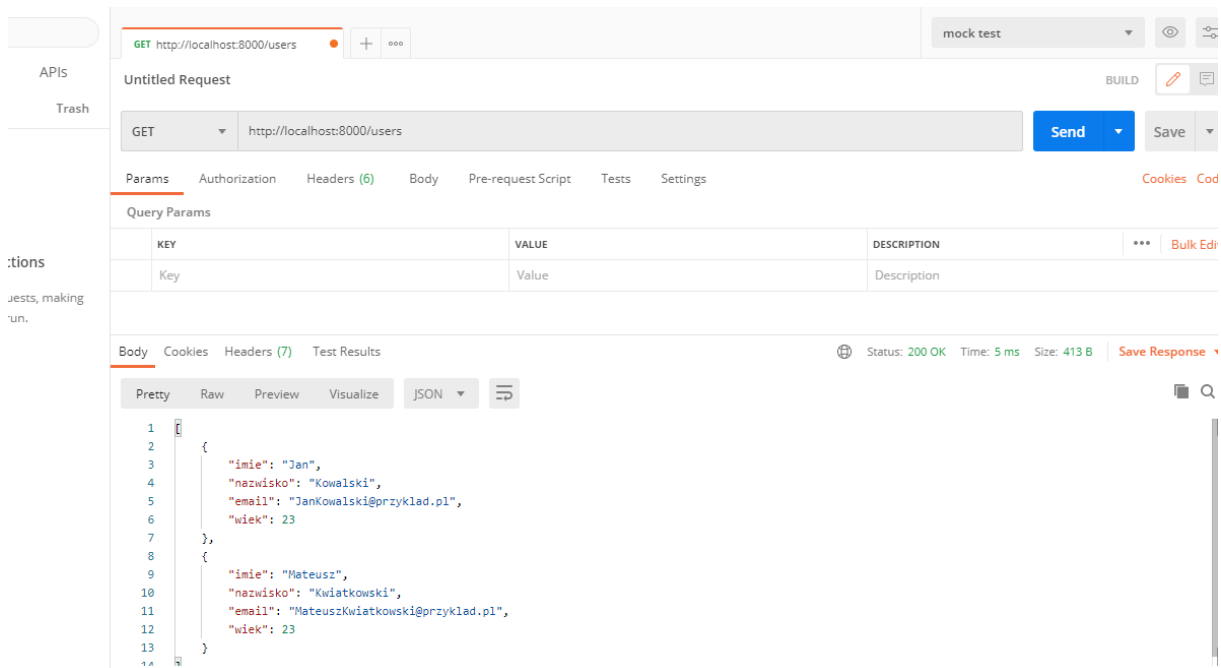




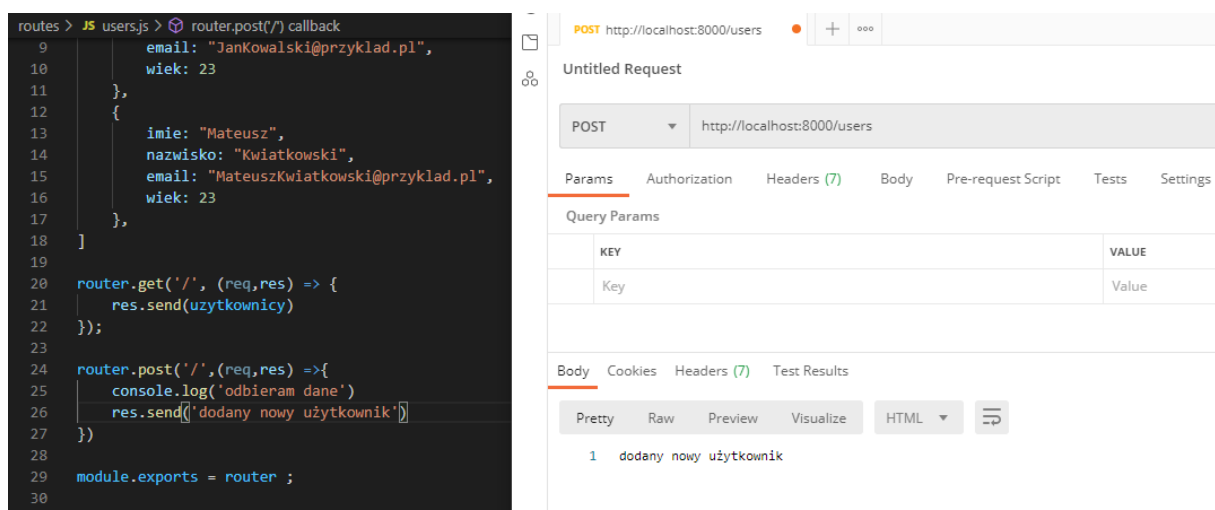
Po uruchomieniu aplikacji pokaże się interfejs w którym należy dodać nowe okno:



Po wybraniu powinien pojawić się pasek z wyborem metody i URL. Przetestujmy funkcję Get. Wystarczy wkleić adres naszego serwera, zaznaczyć metodę Get i wysłać, a poniżej otrzymujemy odpowiedź.



Status serwera 200, oraz wypisani użytkownicy sugerują że polecenie zadziałało poprawnie, dlatego przygotujmy funkcję Post.



Teraz spróbujmy faktycznie dodać użytkownika do naszej tablicy i wyświetlić.

```

23
24 router.post('/', (req, res) => {
25     console.log('odbieram dane')
26     const uzytkownik = req.body;
27     uzytkownicy.push(uzytkownik);
28     res.send(uzytkownicy)
29 })
30
31 module.exports = router ;
32

```

Aby skorzystać z Postmana musimy ustawić funkcję na POST, oznaczyć że dane będą wysyłane z Body, a następnie po prawej „text” zmienić na „JSON”

A screenshot of the REST client interface. At the top, it says 'Untitled Request' and 'BUILD' on the right. Below this is a bar with 'POST' on the left and 'http://localhost:8000/users' in the center. To the right of this bar is a blue 'Send' button. Below the bar is a row of tabs: 'Params', 'Authorization', 'Headers (7)', 'Body' (which is underlined in red), 'Pre-request Script', 'Tests', and 'Settings'. Below the tabs is a row of radio buttons: 'none', 'form-data', 'x-www-form-urlencoded', 'raw' (which is selected with a red dot), 'binary', 'GraphQL', and 'JSON' (which has a dropdown arrow).

Teraz wprowadźmy użytkownika i wyślijmy:

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL **JSON** ▼

```
1 {  
2   "imie": "Ala",  
3   "nazwisko": "Kot",  
4   "email": "AlaKot@przyklad.pl",  
5   "wiek": 25  
6 }
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize **JSON** ▼ ↺

```
1 {  
2   {  
3     "imie": "Jan",  
4     "nazwisko": "Kowalski",  
5     "email": "JanKowalski@przyklad.pl",  
6     "wiek": 23  
7   },  
8   {  
9     "imie": "Mateusz",  
10    "nazwisko": "Kwiatkowski",  
11    "email": "MateuszKwiatkowski@przyklad.pl",  
12    "wiek": 23  
13  }  
14 }
```

Po wykonaniu operacji został dodany nowy element:

```
1  [  
2    {  
3      "imie": "Jan",  
4      "nazwisko": "Kowalski",  
5      "email": "JanKowalski@przyklad.pl",  
6      "wiek": 23  
7    },  
8    {  
9      "imie": "Mateusz",  
10     "nazwisko": "Kwiatkowski",  
11     "email": "MateuszKwiatkowski@przyklad.pl",  
12     "wiek": 23  
13   },  
14   null  
15 ]
```

Jak widać nie jest on poprawny, dlatego jest błąd w kodzie, a dokładniej w body-parser.


```
JS index.js > ...  
1  const express = require('express')  
2  const port = 8000  
3  const bodyParser = require('body-parser')  
4  const app = express()  
5  const usersRoutes = require('./routes/users.js');  
6  
7  
8  app.use(bodyParser.json()); //musi być zdefiniowany przed ścieżką do użytkowników  
9  
10 app.use('/users', usersRoutes)  
11  
12 //app.use(bodyParser.json()); // tutaj był za późno zdefiniowany  
13  
14 app.set('view engine', 'hbs')  
15  
16  
17 app.get('/', (req, res) => {
```

Po poprawieniu tego błędu widzimy że element został dodany:



```
1  [
2    {
3      "imie": "Jan",
4      "nazwisko": "Kowalski",
5      "email": "JanKowalski@przyklad.pl",
6      "wiek": 23
7    },
8    {
9      "imie": "Mateusz",
10     "nazwisko": "Kwiatkowski",
11     "email": "MateuszKwiatkowski@przyklad.pl",
12     "wiek": 23
13   },
14   {
15     "imie": "Ala",
16     "nazwisko": "Kot",
17     "email": "AlaKot@przyklad.pl",
18     "wiek": 25
19   }
20 ]
```

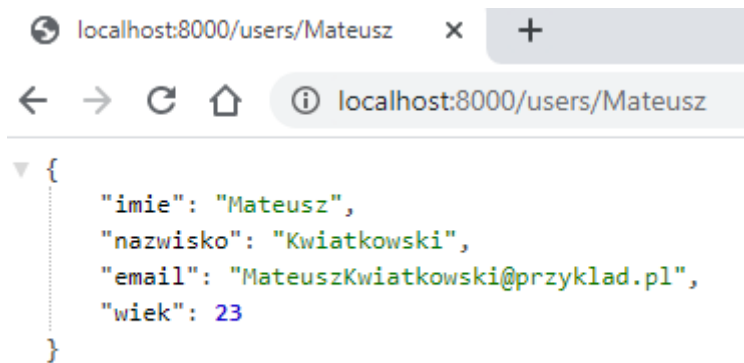
W kolejnym kroku możemy potrzebować odnieść się do konkretnego użytkownika czyli `Get:id`. Jako że w przykładzie nie ma ID możemy odnosić się np. do imienia, oczywiście nie jest to dobre rozwiązanie bo imion jest wiele, ale jako przykład przedstawię poniżej:



```
routes > JS users.js > ...
22   res.send(uzytkownicy)
23 });
24
25 router.post('/', (req, res) => {
26   console.log(req.body)
27   const uzytkownik = req.body;
28   uzytkownicy.push(uzytkownik);
29   res.send(uzytkownicy)
30 })
31
32 router.get('/:id', (req, res) => {
33   const {id} = req.params;
34   const uzytkownikPoImieniu = uzytkownicy.find((user) => user.imie === id)
35   res.send(uzytkownikPoImieniu)
36 });
37
38 module.exports = router ;
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

Req.params przyjmuje parametry które możemy później stosować w kodzie, polecam tutaj wypisać jak wygląda to w konsoli. W tym przypadku znajdujemy użytkownika w tablicy po id, którym jest imię. Aby odnieść się w przeglądarce wystarczy po ukośniku dopisać id.



## Zadanie.

Zaimplementuj CRUD zgodnie z utworzonym własnym API dla diagramu User, a następnie przetestuj w Postmanie z danymi utworzonymi na sztywno. Dodatkowo stwórz pole id dla którego generowany będzie niepowtarzalny ciąg znaków. Wykorzystaj do tego dostępne moduły npm.