



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH***

Szkielety programistyczne w aplikacjach internetowych

dr Mariusz Dzieńkowski

Lublin 2022

LABORATORIUM 1B. ORGANIZACJA ŚRODOWISKA PROGRAMISTYCZNEGO DO BUDOWY APLIKACJI INTERNETOWYCH WYKORZYSTUJĄCYCH SZKIELETY PROGRAMISTYCZNE

Cel laboratorium:

Opanowanie podstawowych umiejętności pracy ze szkieletem Express. Implementacja prostych przykładów zawierających istotne elementy aplikacji działających po stronie serwera.

Zakres tematyczny zajęć:

- Czynności przygotowawcze: inicjowanie projektu, instalacja głównych pakietów składających się na aplikację opartą na szkielecie Express.
- Struktura aplikacji opartych na szkielecie Express, import modułów, wyznaczanie tras, nasłuchiwanie serwera na danym porcie.
- Implementacja i uruchamianie prostych aplikacji opartych na szkielecie Express.

Pytania kontrolne:

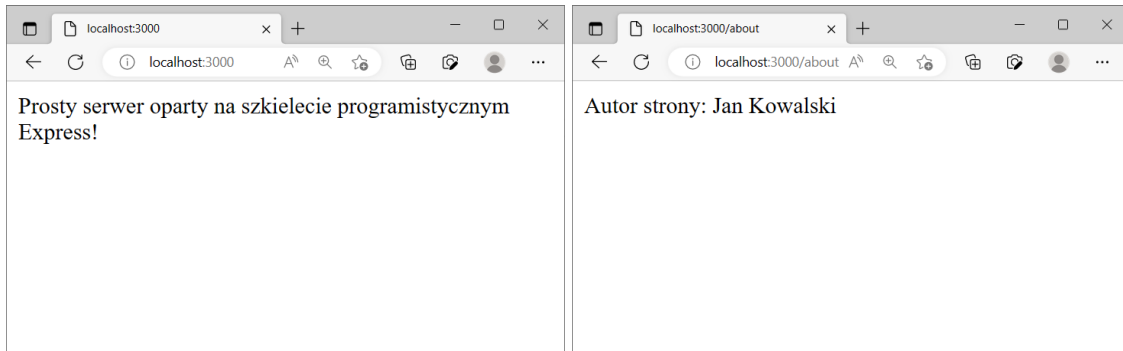
- a) Wymień podstawowe cechy szkieletu programistycznego Express.
- b) Do czego służy i jak jest zbudowany plik manifestu *package.json*?
- c) Na czym polega semantyczne wersjonowanie bibliotek?
- d) Czym jest, do czego służy, jakie są typy oraz jak się uruchamia oprogramowanie pośredniczące?
- e) Do czego służą szablony i z jakich silników szablonów umożliwia szkielet Express?

Zadanie 1.8. Implementacja podstawowego serwera przy pomocy szkieletu Express

Aplikacja, która po wpisaniu w pasku adresu przeglądarki adresu <http://localhost:3000> wyświetli w jej oknie przykładowy tekst.

1. We wcześniej przygotowanym katalogu aplikacji utworzyć plik *package.json* – stanowiący integralną część każdego projektu Node. Plik *package.json* powinien zawierać następujące metadane:
 - name: "express-serwer"
 - description: "Prosta aplikacja serwerowa oparta na szkielecie Express"
 - main: "zad_01.js"
 - keywords: ["express", "serwer"]
 - author: "Jan"
2. Zainstalować szkielet Express, w taki sposób, aby jego zależność znalazła się w części *dependencies* pliku *package.json*.
3. Utworzyć plik *zad_01.js* i zaimplementować kod aplikacji, na który złożą się następujące czynności:
 - dołączenie kodu szkieletu express do aplikacji oraz utworzenie instancji aplikacji internetowej o nazwie *app*,

- utworzenie stałej `PORT` zawierającej numer portu, na którym będzie nasłuchiwał tworzony serwer,
- utworzenie routingu poprzez wywołanie metody `get` dla obiektu `app` i ustawienie trasy `'/'` oraz funkcji ją obsługującej (ang. `callback`), która następnie uruchamia metodę `send` na obiekcie odpowiedzi (ang. `response`) i która wyświetla w oknie przeglądarki tekst: *Prosty serwer oparty na szkielecie programistycznym Express!*,
- wywołanie metody `listen` na instancji obiektu `app` zawierającej jako pierwszy parametr numer portu, a jako drugi funkcję zwrrotną, która w terminalu wyświetli informację postaci: *Serwer działa na porcie: 3000*.



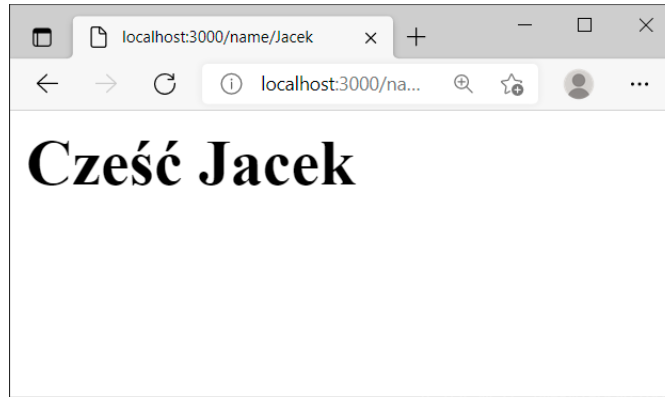
Rys. 1.4 Aplikacja oparta na szkielecie Express

4. Dodać dodatkową trasę `/about`, po użyciu której w oknie przeglądarki wyświetli się informacja o autorze strony: *Autor strony: Jan Kowalski*.
5. Zainstalować pakiet `nodemon`, który spowoduje, że po każdorazowym wprowadzeniu zmian w kodzie, nie będzie potrzeby ponownego zatrzymywania i uruchamiania aplikacji. Zależność tego pakietu powinna się znaleźć w części `devDependencies` pliku `package.json`.
6. Zmodyfikować plik `package.json`, tak aby w części `"scripts"` zawierał dwie definicje start i dev, realizujące różne operacje:

```
"start": "node zad_01.js",  
"dev": "nodemon zad_01.js"
```
7. Uruchomić i sprawdzić działanie serwer wykorzystując obie powyższe definicje.

Zadanie 1.9. Aplikacja wykorzystująca ścieżkę z parametrem

Aplikacja, która po wprowadzeniu w przeglądarce adresu w postaci: `http://localhost:3000/name/Jacek` wyświetli komunikat witający Jacka. Wynik działania aplikacji przedstawia Rys. 1.5.



Rys. 1.5 Aplikacja Express z trasą zawierającą parametr

1. Aplikacja powinna zawierać definicję routingu składającą się z trasy: '/name:imie', odpowiedzi, której status wynosi 200, typu zwracanej treści: 'text/html' oraz samej treści strony www zbudowanej ze znaczników html.
2. Zmodyfikować aplikację w ten sposób, aby przyjmowała dwa parametry i wyświetlała komunikat w postaci *Cześć Jacek i Placek*.

Zadanie 1.10. Aplikacja z obsługą formularza

Aplikacja, która dane wprowadzone do formularza wysyła do serwera, a następnie zwraca do klienta i wyświetla w oknie przeglądarki. Składa się ona z dwóch plików: pierwszy zawiera kod HTML i zawiera formularz (Listing 1.11), drugi to skrypt JavaScript (Listing 1.12), który przetwarza żądanie i odsyła dane do klienta.

Listing 1.11 Kod pliku form.html

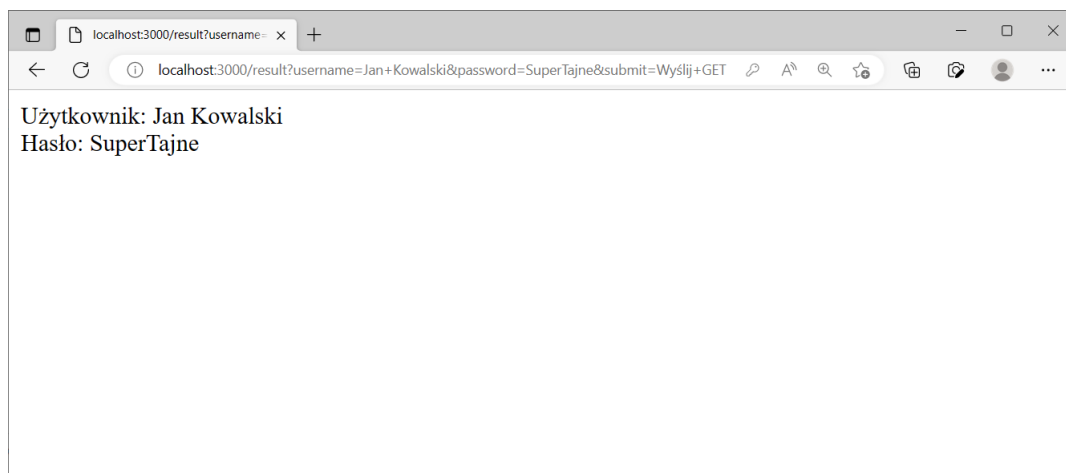
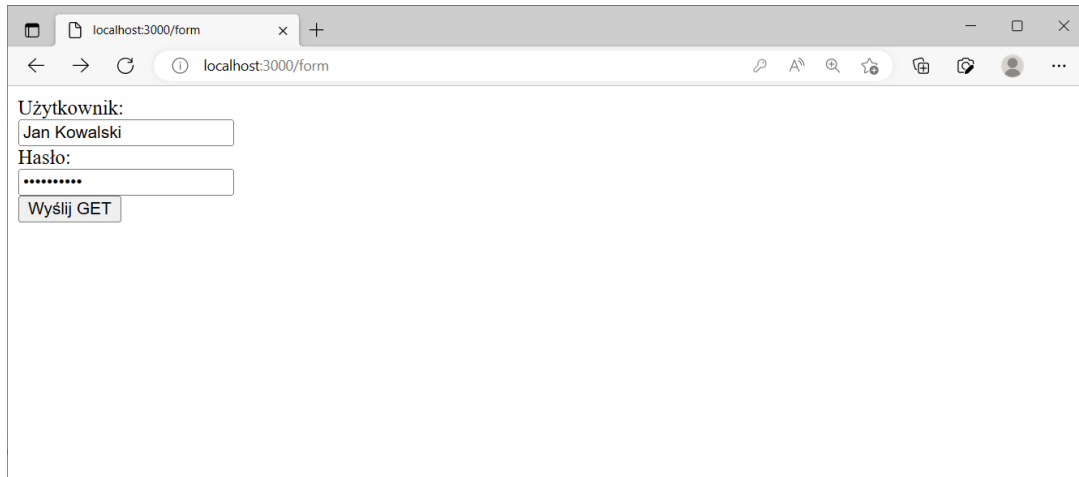
```
<html>
  <body>
    <form action = "/result" method = "GET">
      <label>Użytkownik: </label><br/>
      <input type = "text" name = "username" /> <br/>
      <label>Hasło: </label><br/>
      <input type = "password" name = "password" /> <br/>
      <input type = "submit" name = "submit" value = "Wyślij GET">
    </form>
  </body>
</html>
```

Listing 1.12 Kod aplikacji serwerowej index.js

```
const express = require('express')
const path = require('path')
const app = express()
const PORT = 3000

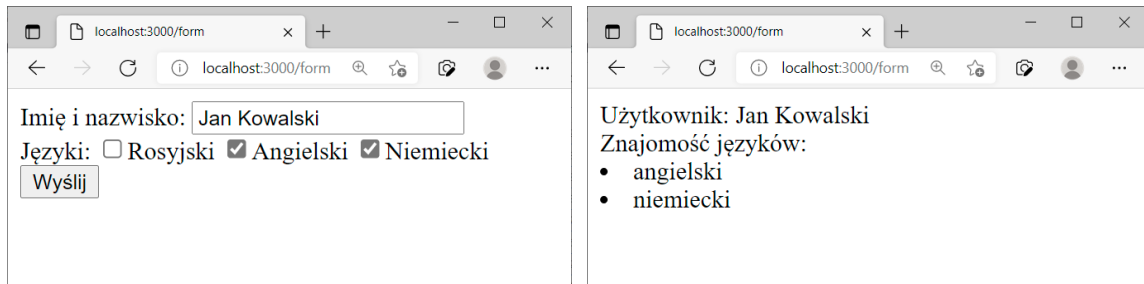
app.get("/form", (req, res) => {
  res.sendFile(path.join(__dirname, "form.html"))
})
```

```
app.get("/result", (req, res) => {  
  let username = req.query.username  
  let password = req.query.password  
  res.send("Użytkownik: " + username + "<br>Hasło: " + password)  
})  
  
app.listen(PORT, ()=> console.log(`Serwer działa na porcie ${PORT}`))
```



Rys. 1.6 Aplikacja przetwarzająca dane z formularza

- Spraw by w przypadku niewprowadzenia danych: użytkownika lub hasła ukazał się komunikat: *Uzupełnij dane!*.
- Zmodyfikuj kod aplikacji tak, aby wykorzystywał do wysyłania danych metodę POST.
- Utwórz nowy formularz zawierający pole tekstowe: imię i nazwisko oraz 3 komponenty checkbox weryfikujących znajomość języków obcych. Do wysyłania danych na serwer niech posłuży komponent typu *submit*. Wygląd formularza oraz okno z wynikami przedstawia Rys. 1.7.



The image shows two browser window screenshots side-by-side, both displaying the URL 'localhost:3000/form'. The left window shows a form with the following fields: 'Imię i nazwisko:' with the value 'Jan Kowalski', 'Języki:' with radio buttons for 'Rosyjski' (unchecked), 'Angielski' (checked), and 'Niemiecki' (checked), and a 'Wyślij' button. The right window shows the result of the submission: 'Użytkownik: Jan Kowalski', 'Znajomość języków:', and a list of selected languages: 'angielski' and 'niemiecki'.

Rys. 1.7 Obsługa formularza z komponentami checkbox

Zadanie 1.11. Walidacja formularzy

Istnieje wiele metod przeznaczonych do sprawdzania danych przychodzących z formularza do serwera. Są one zdefiniowane w module *validator.js*. Oto lista wybranych metod:

- `contains()` – sprawdza, czy zawiera określoną wartość
- `equals()` – sprawdza, czy wprowadzona wartość równa się innej wartości
- `isAlpha()`
- `isAlphanumeric()`
- `isAscii()`
- `isBase64()`
- `isBoolean()`
- `isCurrency()`
- `isDecimal()`
- `isEmpty()`
- `isFQDN()` – sprawdza, czy nazwa domeny jest w pełni kwalifikowana
- `isFloat()`
- `isHash()`
- `isHexColor()`
- `isIP()`
- `isIn()`, sprawdza, czy wartość znajduje się w tablicy dozwolonych wartości
- `isInt()`
- `isJSON()`
- `isLatLong()`
- `isLength()`
- `isLowercase()`
- `isMobilePhone()`
- `isNumeric()`
- `isPostalCode()`
- `isURL()`
- `isUppercase()`
- `isWhitelisted()` - sprawdza poprawność wprowadzanych danych na podstawie listy białych, dozwolonych znaków

Więcej informacji dotyczących biblioteki *validator.js* można znaleźć w repozytorium GitHub pod adresem: <https://github.com/validatorjs/validator.js#validators>

Listing 1.13 przedstawia fragment kodu aplikacji obsługującej formularz przedstawiony na Rys. 1.8 i 1.9. Dane wprowadzane do pól formularza są walidowane. W przypadku napotkania błędu w danych wejściowych generowana będzie informacja o błędach w formacie JSON (Rys. 1.9).

Listing 1.13 Kod aplikacji z walidacją danych z formularza

```
//...
const { check, validationResult } = require('express-validator')
//...
app.post("/form", [
  check('nazwisko').isLength({ min: 3 }),
  check('email').isEmail(),
  check('wiek').isNumeric()
], (req, res) => {
  const errors = validationResult(req)
  if (!errors.isEmpty()) {
    return res.status(422).json({ errors: errors.array() })
  }
  const nazwisko = req.body.nazwisko
  const email = req.body.email
  const wiek = req.body.wiek
  res.send("Użytkownik: " + nazwisko + "<br>Email: " + email + "<br>Wiek: " + wiek)
})
```

The image shows a web form on the left with three input fields: 'Użytkownik:' containing 'Kowalski', 'Email:' containing 'kowalski@gmail.com', and 'Wiek:' containing '24'. Below these is a 'Wyślij' button. To the right, a separate box displays the successful submission result: 'Użytkownik: Kowalski', 'Email: kowalski@gmail.com', and 'Wiek: 24'.

Rys. 1.8 Formularz z walidacją po stronie serwera oraz zwracany wynik

The image shows a web form on the left with three input fields: 'Użytkownik:' containing 'Ko', 'Email:' containing 'kowalski@gmail.com', and 'Wiek:' containing '24'. Below these is a 'Wyślij' button. To the right, a box displays a JSON error response:

```
{
  "errors": [
    {
      "value": "Ko",
      "msg": "Invalid value",
      "param": "nazwisko",
      "location": "body"
    }
  ]
}
```

Rys. 1.9 Obsługa formularza zawierającego błąd w nazwisku oraz wynik w formacie JSON

- Zmodyfikuj walidację nazwiska, tak, aby dodatkowo możliwe było sprawdzenie, czy jego długość nie przekracza 25 znaków. Zabezpiecz, aby wprowadzane dane były wyłącznie literami alfabetu.
- Dla pola *wiek* zastosuj walidację, która będzie sprawdzała jego zakres (0 - 110 lat).
- Zastąp własnym, domyślny tekst generowany na wskutek wystąpienia błędu we wprowadzonych danych. Użyj do tego celu metody: `withMessage('komunikat o błędzie')`

Istnieje możliwość utworzenia własnego, niestandardowego walidatora przy użyciu metody `custom()`. W funkcji wywołania zwrotnego można zaniechać walidacji, albo przez wygenerowanie wyjątku (Listing 1.14), albo przez zwrócenie odrzuconej obietnicy (Listing 1.15).

Listing 1.14 Fragment kodu z własną walidacją polegającą na generowaniu wyjątku

```
check('email').custom(email => {  
  if(alreadyHaveEmail(email)){  
    throw new Error('Email już istnieje!')  
  }  
}),
```

Listing 1.15 Fragment kodu z własną walidacją polegającą na odrzuceniu obietnicy

```
check('email').custom(email => {  
  if(alreadyHaveEmail(email)){  
    return Promise.reject('Email już istnieje!')  
  }  
}),
```

Z walidacją często wykonuje się oczyszczanie danych przychodzących z formularza. Z języka angielskiego operacje tą nazywa się sanityzacją. Za pomocą dostarczonych odpowiednich metod, można na przykład usunąć białe znaki z początku i końca wartości oraz znormalizować adres email do spójnego wzorca. Zabieg ten może pomóc w usunięciu duplikatów kontaktów utworzonych przez nieco inne dane wejściowe. Na przykład, 'Marek@gmail.com' oraz 'marek@gmail.com' zostaną zamienione na 'marek@gmail.com'. Sanityzery mogą być w prosty sposób dołączane na końcu walidatorów.

Listing 1.16 Fragment kodu, w którym zastosowano oczyszczanie danych

```
check('message')  
  .isLength({ min: 1 })  
  .withMessage('Message is required')  
  .trim(),
```

W Tabeli 1.1 znajduje się kilka wybranych metod służących do oczyszczania danych wejściowych.

Tabela 1.1 Lista popularnych metod do oczyszczania danych

<code>trim()</code>	Przycina znaki na początku i końcu łańcucha
<code>escape()</code>	Zastępuje znaki <, >, &, ' , " za pomocą ukośnika (/)
<code>normalizeEmail()</code>	Normalizuje adresy email, przekształcając je na małe litery
<code>blacklist()</code>	Usuwa znaki pojawiające się na czarnej liście
<code>whitelist()</code>	Usuwa znaki pojawiające się na białej liście
<code>stripLow()</code>	Usuwa niewidoczne, kontrolne znaki ASCII
<code>bail()</code>	Zatrzymuje walidację, jeśli którakolwiek z poprzednich zakończyła



	się niepowodzeniem
--	--------------------

Istnieje możliwość opracowania własnej metody oczyszczającej (Listing 1.17).

Listing 1.17 Implementacja własnej metody oczyszczającej dane

```
const sanitizeValue = value => {
  //sanitize...
}

app.post('/form', [
  check('value').customSanitizer(value => {
    return sanitizeValue(value)
  }),
], (req, res) => {
  const value = req.body.value
})
```

- d) W kodzie skryptu zastosuj metody oczyszczające: *trim()*, *normalizeEmail()*, *stripLow()* oraz *bail()*.
- e) Zdefiniuj własną metodę oczyszczającą, która z imienia i nazwiska usuwa wszystkie litery oprócz pierwszych, tworząc inicjały.

Zadanie 1.12. Implementacja prostego API

Kod z Listingu 1.18 zawiera tablicę literałów obiektowych przechowujących informacje o trzech użytkownikach. Zdefiniowana tu została również trasa */api/users*, po wywołaniu której w przeglądarce internetowej zostaną wyświetlone dane w formacie JSON (Rys. 1.10).

Listing 1.18 Tablica literałów obiektowych

```
const users = [
  {
    id: 1,
    name: "Jan Kowalski",
    email: "jan.kowalski@gmail.com",
    status: "aktywny"
  },
  {
    id: 2,
    name: "Adam Nowak",
    email: "adam.nowak@gmail.com",
    status: "nieaktywny"
  },
  {
    id: 3,
    name: "Andrzej Stach",
    email: "andrzej.stach@gmail.com",
    status: "aktywny"
  },
]
```



```
app.get('/api/users', (req,res) => {
  res.json(users)
})
```



Rys. 1.10 Wynik w formacie JSON wyświetlony w przeglądarce

- a) Utwórz moduł *users.js* i przenieś do niego tablicę *users*, a następnie dokonaj eksportu tej tablicy. Jednocześnie w pliku serwera zaimportuj dane użytkowników.

Kod z Listingu 1.19 wyświetla w oknie przeglądarki wszystkich użytkowników znajdujących się w tablicy. Poszerzenie funkcjonalności aplikacji o możliwość wyświetlania pojedynczego, wybranego użytkownika, na podstawie jego identyfikatora, wiąże się z dodaniem do kodu programu definicji trasy z Listingu 1.21.

Listing 1.19 Implementacja trasy (z parametrem) wyświetlająca dane wybranego użytkownika

```
app.get('/api/users/:id', (req, res) => {
  res.json(users.filter(user => user.id === parseInt(req.params.id)))
})
```

Efekt działania powyższego kodu nastąpi po wpisaniu w polu adresu przeglądarki: <http://localhost:3000/api/users/2>

Istotne jest zabezpieczenie sytuacji, kiedy to podany zostanie identyfikator użytkownika spoza zakresu 1 - 3. Sytuację przekroczenia tego zakresu zabezpiecza kod z Listingu 1.20.

Listing 1.20 Zabezpieczenie przed wprowadzeniem id użytkownika spoza zakresu

```
app.get('/api/users/:id', (req, res) => {
  const found = users.some(user => user.id === parseInt(req.params.id))
  if(found){
```

```
res.json(users.filter(user => user.id === parseInt(req.params.id)))
} else {
  res.status(400).json({msg: `Użytkownik o id ${req.params.id} nie został odnaleziony`})
}
```

Rozbudowa aplikacji polegająca na dodawaniu nowego użytkownika sprowadza się do użycia metody POST na obiekcie *app* (Listing 1.21). W tym kodzie musi zostać dodane oprogramowanie pośredniczące odpowiedzialne za rozpoznawanie przychodzących żądań w postaci obiektów JSON.

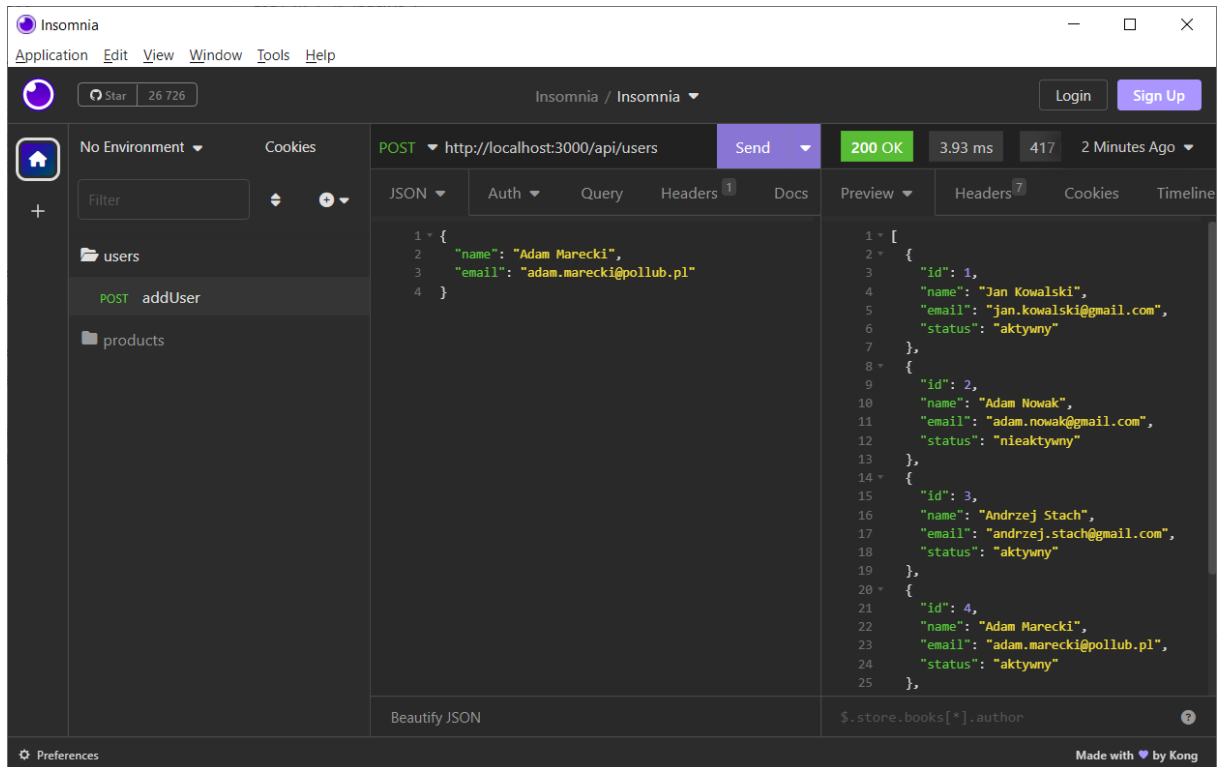
Listing 1.21 Definicja metody dodającej nowego użytkownika do tablicy

```
app.post('/api/users', (req, res) => {
  const newUser = {
    id: users.length + 1,
    name: req.body.name,
    email: req.body.email,
    status: "aktywny"
  }

  if(!newUser.name || !newUser.email){
    return res.status(400).json({msg: 'Wprowadź poprawne imię i nazwisko oraz email!'})
  }
  users.push(newUser)
  res.json(users)
})
```

Do przetestowania żądań typu POST można użyć narzędzi Insomnia lub Postman. Posługując się środowiskiem Insomnia (Rys. 1.11), należy dodać nowe żądanie, nadać mu nazwę i ustawić typ – w tym przypadku POST. Następnie w polu adresu, należy wprowadzić adres: <http://localhost:3000/api/users>, skonfigurować żądanie wybierając z listy Body opcję JSON oraz wprowadzić treść ciała żądania:

```
{
  "name": "Adam Marecki",
  "email": "adam.marecki@example.pl"
}
```



Rys. 1.11 Narzędzie do generowania żądań - Insomnia

Kolejną funkcjonalność tworzonej aplikacji to aktualizowanie danych wybranego użytkownika identyfikowanego na podstawie klucza - id. Do tego celu można wykorzystać żądanie typu PATCH.

Listing 1.22 Definicja metody PATCH modyfikującej dane wybranego użytkownika

```
app.patch('/api/users/:id', (req, res) => {
  const found = users.some(user => user.id === parseInt(req.params.id))
  if(found){
    const updUser = req.body
    users.forEach(user => {
      if(user.id === parseInt(req.params.id)) {
        user.name = updUser.name ? updUser.name : user.name
        user.email = updUser.email ? updUser.email : user.email

        res.json({msg: 'Dane użytkownika zaktualizowane', user})
      }
    })
  } else {
    res.status(400).json({msg: `Użytkownik o id ${req.params.id} nie istnieje!`})
  }
})
```

- b) Za pomocą programu Insomnia przetestuj działanie modyfikacji adresu email użytkownika o id = 2. W tym celu utwórz żądanie PATCH i w jego ciele umieść treść odpowiedzialną za aktualizację emaila:

```
{  
  "email":"adam.nowak123@gmail.com"  
}
```

- c) Zaimplementuj żądanie usunięcia użytkownika o wybranym identyfikatorze.

Zadanie 1.13. Implementacja warstwy pośredniczącej (middleware)

Kod na Listingu 1.23 to implementacja funkcji pośredniczącej z trzema parametrami, która w terminalu wyświetla informacje o rodzaju użytej metody żądania oraz trasy.

Listing 1.23 Definicja funkcji pośredniczącej

```
let metoda = (req, res, next) => {  
  console.log("Metoda: ", req.method)  
  let sciezka = "Ścieżka: " + req.protocol + "://" + req.get('host') + req.originalUrl  
  console.log(sciezka)  
  next()  
}
```

- a) Zmodyfikuj funkcję tak, aby wyświetlić informacje w oknie przeglądarki. Użyj do tego celu metody *res.send()*.
- b) Wykorzystując oprogramowanie Insomnia, przetestuj działanie zdefiniowanej funkcji pośredniczącej dla żądań POST, PUT i DELETE.

Zadanie 1.14. Middleware i modularyzacja aplikacji – kontynuacja zadania 1.10

- a) Zaimplementuj funkcję pośredniczącą *isAuthorized*, która sprawdza czy użytkownik jest zalogowany i ma dostęp do zasobów. Jeżeli ustawionym hasłem jest *secretpaswd* - użytkownik uzyska dostęp do danych, w przeciwnym wypadku wyświetli się informacja *Dostęp zabroniony* oraz zostanie zwrócony status odpowiedzi 401. Przetestuj utworzoną funkcję za pomocą narzędzia Insomnia.
- b) Utwórz katalog *api* i w nim plik *routes.js* oraz przenieś do niego wszystkie metody definiujące trasy.

Na początku tego pliku należy dołączyć szkielet express oraz moduł users. Dodatkowo należy użyć klasy *express.Router*, aby utworzyć modułowe, montowalne manipulatory tras. Instancja Routera jest kompletnym systemem middleware i routingu.

```
const router = express.Router()
```

W pliku, w którym znajduje się część główna aplikacji, powinien odbyć się import zawartości pliku *routes.js*, a następnie dołączyć routera jako oprogramowania pośredniczącego.

W wywołaniach metod obsługujących żądania obiekty *app* należy zastąpić obiektem *router*. Ponadto na końcu tego modułu powinien zostać zainicjowany export obiektu *router*.

Kolejną czynnością będzie modyfikacja tras w pliku *routes.js* we wszystkich zaimplementowanych tam metodach odpowiedzialnych za routing, np:

```
router.get('/', (req, res) => {...}  
router.get('/:id', (req, res) => {...}
```

- c) Na koniec należy utworzyć katalog *middleware*, umieścić w nim pliki *metoda.js* i *autoryzacja.js* oraz przenieść do nich kod metod pośredniczących o nazwie *metoda*.

Zadanie 1.15. Stosowanie szablonów

Express jest w stanie obsługiwać silniki szablonów po stronie serwera. Silniki szablonów pozwalają na dodawanie danych do widoku i dynamiczne generowanie kodu HTML. Istnieje możliwość stosowania wielu różnych silników szablonów, np. Pug, Handlebars, Mustache, EJS.

Instalacja silnika Handlebars wygląda następująco:

```
npm install hbs
```

Oto prosty szablon pliku *about.hbs*.

Cześć {{name}}!

Szablony najczęściej umieszcza się w folderze *views*.

Listing 1.24 Kod aplikacji obsługującej endpoint *about*

```
const express = require('express')
const path = require('path')
const app = express()
const hbs = require('hbs')

app.set('view engine', 'hbs')
app.set('views', path.join(__dirname, 'views'))

app.get('/about', (req, res) => {
  res.render('about', {name: 'Jan'})
})
app.listen(3000, () => console.log('Serwer działa!'))
```

- a) Wykorzystując przykład z zadania 1.9, utwórz szablon *info.hbs* wyświetlający dane przechwycone z formularza, wg poniższego wzoru.

Nazwisko: Nowak

Email: nowak@gmail.com

Wiek: 24

Po stronie serwera można również renderować aplikację React, wykorzystując do tego pakiet *express-react-views*. Oprócz niego należy również zainstalować moduły *react* oraz *react-dom*:
`npm install express-react-views react react-dom`

W tym przypadku zamiast silnika *hbs* używa się silnika *express-react-views*. Tym razem pliki szablonu będą miały strukturę w formacie JSX.

Listing 1.25 Kod szablonu w formacie JSX – plik *about.jsx* przechowywany w folderze *views*

```
const React = require('react')

const HelloMessage = (props) => {
  return <div>Nazwisko: {props.nazwisko}</div>
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



```
}  
  
module.exports = HelloMessage
```

Listing 1.26 Implementacja aplikacji korzystająca z szablonu React

```
const express = require('express')  
const path = require('path')  
const app = express()  
const reactEngine = require('express-react-views')  
  
app.set('view engine', 'jsx')  
app.engine('jsx', reactEngine.createEngine())  
  
app.get('/about', (req, res) => {  
  res.render('about', {nazwisko: 'Kowalski'})  
})  
app.listen(3000, () => console.log('Serwer działa!'))
```

- b) Uzupełnij kod w ten sposób, aby aplikacja dodatkowo przekazywała, a następnie wyświetlała email i wiek.

Zadanie 1.16. Aplikacja generująca komunikaty w konsoli

Utwórz aplikację, która będzie podawała następujące komunikaty w konsoli serwera:

```
PS C:\Users\user\Desktop\app> node server.js  
01.03.2020 19:20:14.080 === Serwer zostaje uruchomiony na porcie 3000.  
01.03.2020 19:20:18.905--- Klient wysłał zapytanie o plik scripts/script.js.  
01.03.2020 19:20:23.359--- Klient wysłał zapytanie o plik styles/css/style.css.  
01.03.2020 19:20:33.124--- Klient wysłał zapytanie o plik index.html.
```

Do wypisywania daty stwórz oddzielną funkcję `getDate()` w pliku serwera. Następnie umieść funkcję `getDate()` w oddzielnym pliku w folderze *server-files* pod nazwą *getDate.js*. Do eksportu funkcji użyj polecenia `module.exports = getDate`.

Zadanie 1.17. Aplikacja z trasami do pięciu plików

Utwórz aplikację, której zadaniem będzie routowanie pięciu różnych plików .html (strona1, strona2, ...) za pomocą Expressa.

Zadanie 1.18. Aplikacja konwertująca stopnie na radiany

Utwórz aplikację, która według parametrów *value* oraz *toRad* (false lub true) będzie zamieniała wartości ze stopni na radiany oraz z radianów na stopnie (zależnie od przekazanej wartości *toRad*). Wartość podana w stopniach/radianach przekazywana jako parametr *value*.

`localhost:3000/?toRad=true&value=160`

`160 stopni to 2.792526803190927 radianow.`

Zadanie 1.19. Aplikacja wykorzystująca parametry

Korzystając parametru *bg* przesłanego w adresie przez klienta, ustaw kolor tła strony html.



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny

