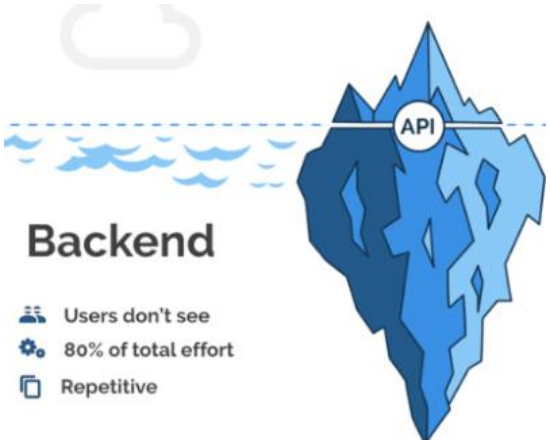


Programowanie aplikacji internetowych
2025/26

Instrukcja laboratoryjna
Backend w .NET albo Express

Prowadzący: Tomasz Goluch
Wersja: 1.0

I. Wprowadzenie

Cel: Przekazanie podstawowych informacji o laboratorium.

Laboratorium odbywa się na maszynach fizycznych (komputery laboratoryjne albo własne laptopy), z wykorzystaniem preferowanego edytora lokalnego (np. *Visual Studio*, *Visual Studio Code, Rider*, *WebStorm*. Nie ma żadnych przeciwskazań aby laboratorium uruchomić w oparciu o kontenery Docker, szczególnie z wykorzystaniem Docker Compose.

II. WEB API (Express, Cors)

Cel: Zapoznanie z frameworkiem Express w wersji językowej TypeScript oraz mechanizmem Cors pozwalającym na łatwe tworzenie interfejsów API.

Express to framework webowy o minimalnej funkcjonalności w kład której wchodzi: routing i wywołania oprogramowania pośredniczącego (middleware. Aplikacja Express jest w zasadzie serią wywołań funkcji oprogramowania pośredniczącego.

Tworzymy nowy projekt (należy uważać aby nie utworzyć odwołania cyklicznego nazywając projekt identycznie jak jedno z jego zależności np. express albo cors):

`npm init` jeśli chcemy pominąć kwestionariusz można dodać parametr `-y`. Otrzymujemy plik projektu `package.json`, który zawiera ,podstawowe informacje takie jak: nazwa, wersja listę zależności wymaganych przez aplikację czy opis projektu. Jego głównym zadaniem jest umożliwienie menedżerowi npm uruchamianie projektu, skryptów oraz instalowanie zależności[1].

```
{
  "name": "backend",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Instalacja frameworka Express przy pomocy npm:

```
npm install express
```

Plik `package.json` powinien teraz zawierać sekcje zależności – `dependencies`:

```
  "dependencies": {
    "express": "^4.21.1"
  }
```

[1] https://mwebs.pl/czym-jest-package-json

Moduły powinny zostać zainstalowane w folderze `node_modules`. Przykladowo w folderze `node_modules\express` powinien znajdować się moduł express zawierający między innymi własny plik projektu `package.json`:

```
{
  "name": "express",
        ...
}
```

W kolejnym kroku instalujemy typescript'a:

```
npm install --save-dev typescript @types/node @types/express ts-node-dev
```

Flaga `--save-dev` aktualizuje `devDependencies` w pakiecie. Są one używane tylko do lokalnego testowania i rozwoju aplikacji.

```
  "devDependencies": {
    "@types/express": "^5.0.0",
    "@types/node": "^22.9.0",
    "ts-node-dev": "^2.0.0",
    "typescript": "^5.6.3"
  }
```

Dodajemy plik `tsconfig.json` konfigurowania opcji kompilatora dla projektu:

```
npx tsc –init
```

Otrzymany plik konfiguracyjny należy dostosować do własnych potrzeb, np. ustawiając:

- wersję językową emitowanego JavaScript'u aby był zgodny z dołączonymi deklaracjami bibliotek
- folder dla emitowanych plików
- pliki które będą należały do projektu (`include`[2]) oraz ewentualne włączenia (`exclude`). Exclude wyłącza tylko pliki, które są uwzględniane w ramach `include`.
- referencje (`references`) do projektu[3].

```
{
  "compilerOptions": {
    "target": "es2023",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
    "outDir": "./dist",
  },
```

[2] https://www.typescriptlang.org/tsconfig/#include
[3] https://www.typescriptlang.org/docs/handbook/project-references.html

```
  "include": ["src/**/*.ts", "tests/**/ts"],
  "references": [{ "path": "./infrastructure.json" }],
  "exclude": ["node_modules"]
 }
```

Po takich zmianach należy odpowiednio przystosować strukturę katalogów, dodając `src`, `tests` oraz ewentualnie podfoldery i przenieść do nich odpowiednie pliki. Docelowa struktura folderu `src` dla aplikacji Clear Architecture może wyglądać następująco:

```
src/
├── domain/
│   ├── entities/
│   └── interfaces/
├── use-cases/
├── infrastructure/
│   ├── database/
│   └── repositories/
└── interface/
    ├── controllers/
    └── routes/
```

Od TypeScript 3.0 dostępne są odwołania (`references`) do projektu, które umożliwiają strukturowanie programów TypeScript w mniejsze części. Dzięki temu możesz znacznie skrócić czas kompilacji, wymusić logiczny podział między komponentami i zorganizować kod w nowy i lepszy sposób. Domyślnie tsc nie będzie automatycznie kompilować zależności, chyba że zostanie wywołany z flagą `-build`. Ma to umożliwić szybsze kompilacje TypeScript.

III. Zarządzanie konfiguracją i sekretami (.env)

Cel: Zapoznanie ze sposobem bezpiecznego zarządzania konfiguracją i sekretami aplikacji (np. hasła, klucze API) przy użyciu zmiennych środowiskowych i pakietu dotenv.

Zapisywanie wrażliwych danych, takich jak hasła do bazy danych czy sekretne klucze do tokenów, bezpośrednio w kodzie źródłowym jest **bardzo złą i niebezpieczną praktyką**. Dane te mogą łatwo wyciec, np. poprzez publiczne repozytorium kodu (Git). Aby temu zapobiec, używa się **zmiennych środowiskowych**. Są to zmienne dostarczane do aplikacji z zewnątrz (przez system operacyjny lub narzędzia uruchomieniowe). Pakiet `dotenv` ułatwia zarządzanie tymi zmiennymi podczas lokalnego rozwoju. Wczytuje on zmienne z pliku o nazwie `.env` do globalnego obiektu `process.env` w Node.js.

W pierwszym kroku instalujemy pakiet `dotenv`:

```
npm install dotenv
```

Następnie w **głównym folderze projektu** (obok `package.json`) tworzymy plik o nazwie `.env`. Plik ten **nigdy** nie powinien być wysyłany do repozytorium kodu. Umieszczamy w nim naszą konfigurację w formacie `KLUCZ=WARTOŚĆ`:

```
# Port na którym uruchomimy serwer
PORT=3000
```

```
# Dane logowania do MongoDB (używane w sekcji Mongoose)
MONGO_LOGIN="twoj_login_do_bazy"
MONGO_PASSWORD="twoje_haslo_do_bazy"

# Sekretny klucz do podpisywania tokenów JWT
TOKEN_KEY="bardzo_dlugi_i_tajny_klucz_jwt"
```

Aby aplikacja mogła wczytać te zmienne, musimy zaimportować i skonfigurować `dotenv` **na samym początku** głównego pliku aplikacji (np. `index.ts` lub `app.ts`).

```
// Wczytaj zmienne z pliku .env na sam początek!
require("dotenv").config(); [cite: 380]

// Reszta importów
var express = require('express')
// ...
```

Od teraz wszystkie zmienne z pliku `.env` są dostępne w obiekcie `process.env`. Możemy od razu wykorzystać zmienną `PORT` przy uruchamianiu serwera:

```
// Zamiast: const port = 3000;
const port = process.env.PORT || 3000;

app.listen(port, () => {
 console.log("Server running on port " + port);
});
```

Użycie `|| 3000` zapewnia wartość domyślną, gdyby zmienna `port` nie została zdefiniowana w pliku `.env`.

Aby mieć pewność, że plik `.env` z naszymi hasłami nie zostanie przypadkowo wysłany na GitHuba, **musimy** dodać go do pliku `.gitignore` (w głównym folderze projektu):

```
# Zależności
node_modules

# Pliki konfiguracyjne ze zmiennymi środowiskowymi
.env

# Skompilowane pliki TypeScript
dist
```

IV. Użycie Express.js

Cel: Stworzenie podstawowego serwera API w Express, obsługa żądań HTTP (GET/POST), konfiguracja kluczowych middleware (CORS, parsowanie body) oraz wprowadzenie do modularnego routingu za pomocą express.Router.

Tworzenie aplikacji Express i obsługa żądań:

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

Funkcja `require` nie jest częścią standardowego API JS lecz jest udostępniana przez Node.js. Służy do importowania modułów JSON oraz plików lokalnych. Moduły importowane są z folderu node_modules Funkcja `express` jest funkcją najwyższego poziomu eksportowaną przez moduł `express` i tworzy aplikację Express. Ostatnia funkcja `express.json` umieszcza przeanalizowane dane z żądania JSON w `req.body`.

Uruchomienie aplikacji (plik `index.js`):

```
const port = 3000;
app.listen(port, () => {
    console.log("Server running on port " + 3000);
});

app.get('/', (req, res) => {
    res.json({ app: 'Run app' });
});
```

Funkcja `listen` służy do wiązania i nasłuchiwania połączeń na określonym hoście i porcie. Jeśli numer portu zostanie pominięty lub będzie miał wartość 0, system operacyjny przypisze dowolny nieużywany port, może to być przydatne w przypadku automatyzacji zadań, np. testy jednostkowe. Funkcje `get`, `post`, `put`, i `delete` realizują odpowiadające im żądania http.

Komenda uruchamiająca `npm run start` (wymaga wpisu `"start": "node index.js"` w sekcji `scripts` w pliku `package.json`) albo bezpośrednio `node index.js`:

```
D:\_VUE_Backend\Express>node app.js
Server running on port 3000
```

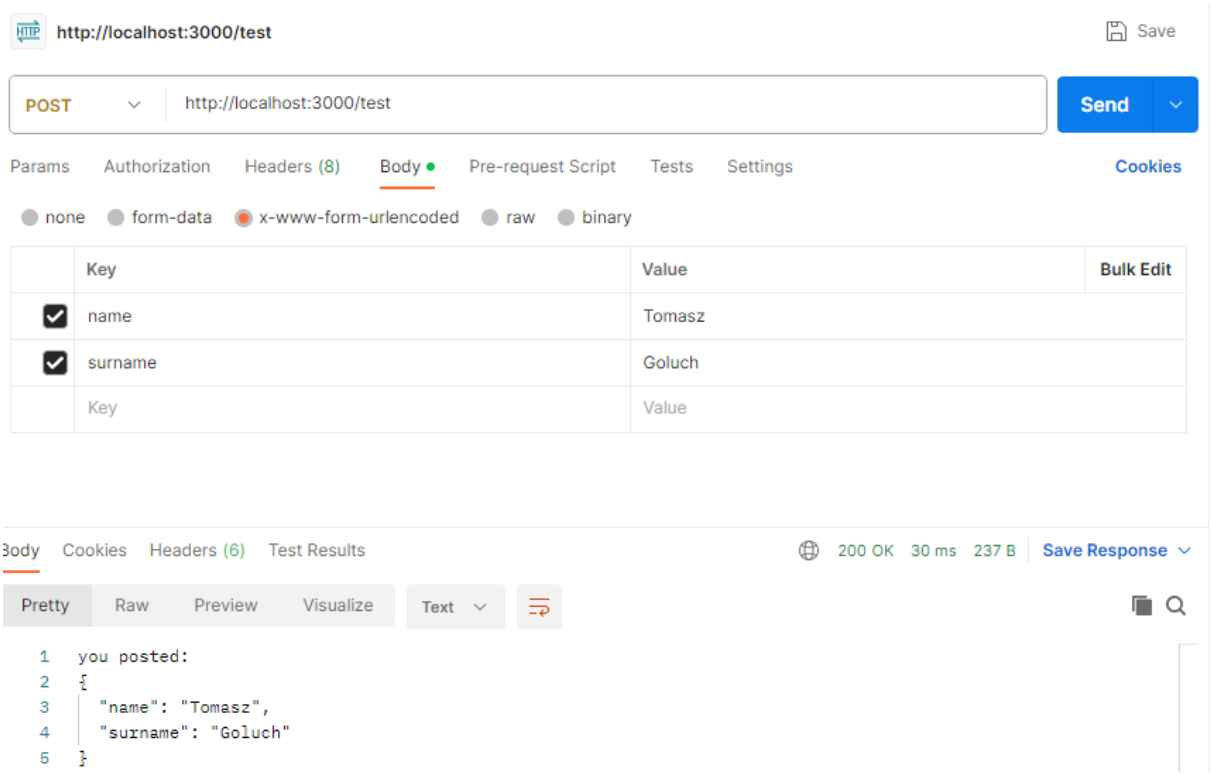
Serwer dostępny jest pod adresem: http://localhost:3000/:


```
{"app":"Run app"}
```

W celu przetworzenia żądania post wykorzystamy middleware do parsowania zawartości requestów post.

```
app.use(express.json())
app.use(express.urlencoded({ extended: false }))
app.post("/test", (req, res) => {
    res.setHeader('Content-Type', 'text/plain')
    res.write('you posted:\n')
    res.end(JSON.stringify(req.body, null, 2))
 });
```

W celu wysłania żądania najlepiej posłużyć się aplikacją Postman:



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`. Jeśli chcemy możemy zainstalować express'a z wszystkimi zależnościami jednym poleceniem: `npm install express cors`. Użycie:

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

Wykorzystanie klasy `express.Router` podział pozwala na tworzenie modularnych, montowalnych hander'ów do obsługi routing'u. W powyższym przypadku moduł jest zaimplementowany w pliku `birds.js`:

```
var api = require('./api')
app.use('/api', api);


const express = require('express')
const router = express.Router()

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', (req, res) => {
  res.send('Birds home page')
```

```
})
// define the about route
router.get('/about', (req, res) => {
  res.send('About birds')
})

module.exports = router
```

Poniższe dwie linijki montują moduł routera (`express.Router`[4]) w pliku `index.js` pod URI `http://<adres>:<port>/api`, czyli `http://localhost:3000/birds`

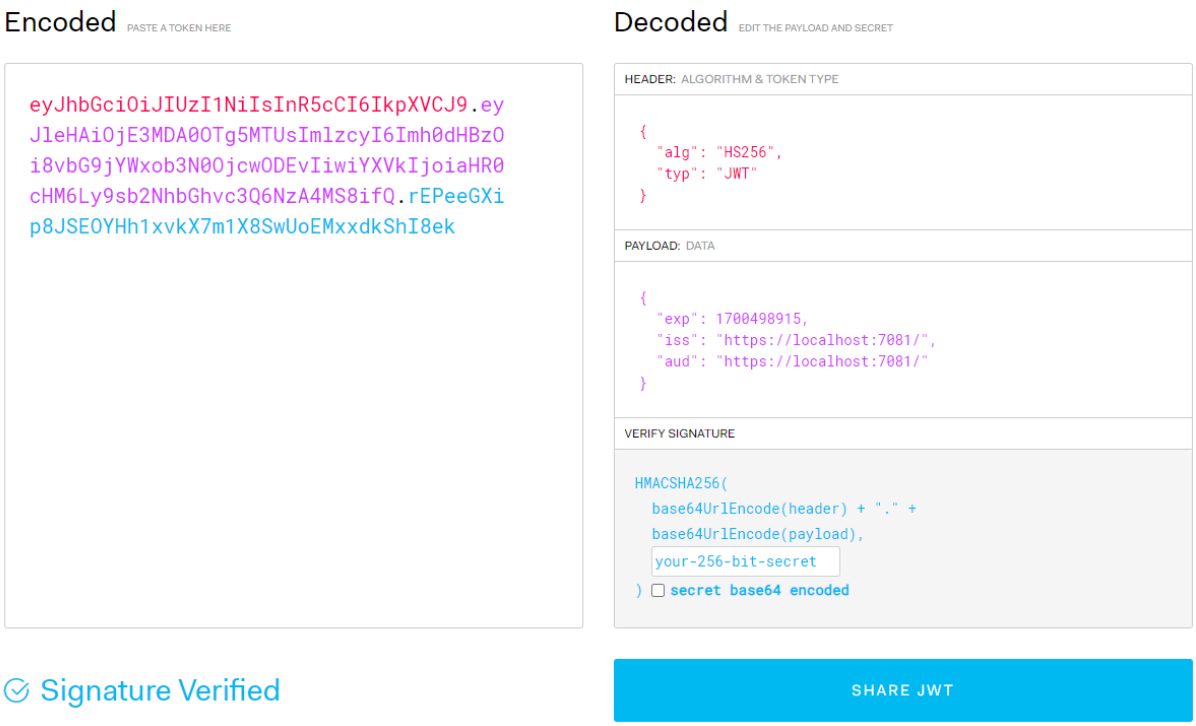
```
const birds = require('./birds')
app.use('/birds', birds)
```

V. JWT (.NET)

Cel: Zapoznanie z JSON Web Tokens w .NET.

Uwierzytelnianie na okaziciela (Bearer authentication) to schemat uwierzytelniania HTTP obejmujący tokeny zabezpieczające (bearer tokens). Są to tokeny JWT (JSON WEB Token) a dokładniej JSON Web Signature (JWS) zapewniające bezpieczny sposób wymiany danych między stronami przy wykorzystaniu obiektów JSON. Podczas logowania klient otrzymuje dwa tokeny: AccessToken zwany dalej po prostu tokenem i RefreshToken. Klient musi wysłać token w nagłówku Authorization podczas wysyłania żądań do chronionych zasobów. Token nie jest szyfrowany. Nie powinien zatem zawierać wrażliwych danych, jak hasła, czy klucze szyfrujące a komunikacja klient-serwer powinna odbywać się po HTTPS. W przypadku przejęcia tokena przez osobę trzecia będzie ona mogła korzystać z API w naszym imieniu. W celu dodatkowego zabezpieczenia token posiada pewien czas życia po którym musi zostać odnowiony. Dla standardu OAuth 2.0 jest to to godzina, ale można zmieniać tę wartość od 5 min do jednego dnia[5]. Odświeżenie tokena odbywa się w oparciu o tzw. Refresh Token, który również posiada zwój czas życia i musi on być dłuższy. Proces odświeżania odbywa się, bez potrzeby ingerencji użytkownika. Token może zawierać np. nazwę użytkownika oraz jego rolę ponieważ w oparciu o sekrety aplikacji jest podpisywany poprzez dołączenie sumy kontrolnej. W jego skład wchodzą 3 części: nagłówka, payloudu, czyli listy twierdzeń (claims'ów) i podpisu, oddzielone kropkami. Mając taki token możemy go odszyfrować na stronie: https://jwt.io/.

[4] https://expressjs.com/en/guide/routing.html
[5] https://learn.microsoft.com/en-us/azure/active-directory-b2c/configure-tokens?pivots=b2c-user-flow#token-lifetime-behavior



Token zawiera wiele claims'ów, oto siedem rekomendowanych aczkolwiek niewymaganych przez specyfikację:

- `iss (issuer)` – emitent JWT;
- `sub (subject)` – podmiot JWT (użytkownik, organizacja lub usługa);
- `aud (audience)` – odbiorca/y, dla którego przeznaczony jest JWT;
- `exp (expiration time)` – czas, po którym wygaśnie JWT;
- `nbf (not before time)` – czas, przed którym JWT nie może zostać przyjęty do przetwarzania;
- `iat (issued at time)` – czas wydania JWT; można wykorzystać do określenia jego wieku;
- `jti (JWT ID)` – unikalny identyfikator; można użyć, aby zapobiec ponownemu odtworzeniu JWT (pozwala na użycie tokena tylko raz).

Lista wszystkich zarejestrowanych claims'ów znajduje się tutaj: https://www.iana.org/assignments/jwt/jwt.xhtml#claims.

W backendzie .NET obsługę uwierzytelniania na okaziciela udostępnia biblioteka Microsoft.AspNetCore.Authentication.JwtBearer. Instalacja za pomocą npm:

```
NuGet\Install-Package Microsoft.AspNetCore.Authentication.JwtBearer
```

Przykładowa rejestracja i konfiguracja walidacji tokenów:

```
builder.Services.AddSwaggerGen();
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
```

```
    ValidAudience = configuration["JwtAuthentication:Audience"]
    ValidIssuer = configuration["JwtAuthentication:Issuer"]
    ValidateAudience = true,
    ValidateIssuer = true,
    ValidateLifetime = true,
    ValidateIssuerSigningKey = true,
    IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Config.Key))
    };
});
```

Klasa TokenValidationParameters służy do konfiguracja walidacji tokenów i zawiera wiele parametrów. Oto najbardziej nas interesujące:

- `ValidAudience` – nazwa odbiorcy/ów (np. klienta api) dla którego został wydany;
- `ValidateAudience` – czy podczas walidacji tokena mamy uwzględnić – czy wartości w `"aud"` i `ValidAudience` są identyczne?;
- `RequireAudience` – czy wartość `"aud"` jest wymagana?;
- `ValidIssuer` / `ValidateIssuer` – to samo co w przypadku odbiorców tylko odnośnie emitenta `"iss"`.
- `IgnoreTrailingSlashWhenValidatingAudience` – czy walidacja ma ignorować ostatniego slash'a w nazwie emitenta tokena;
- `IssuerSigningKey` – klucz służący do podpisania tokena – jest to sekret i na potrzeby rozwijania aplikacji wystarczy trzymać go w osobnym pliku: `secrets.json`, jednak podczas wdrożenia należy wykorzystać dedykowane do tego rozwiązanie, np. `Azure KeyVault`.
- `ValidateIssuerSigningKey` – czy podczas walidacji tokena mamy uwzględnić walidację klucza `IssuerSigningKey`;
- `RequireSignedTokens` – wymaganie podpisania tokena.

W kolejnym kroku musimy dodać do pipeline'u aplikacji warstwy pośrednie odpowiedzialne za uwierzytelnianie i autoryzację:

```
app.UseAuthentication();
app.UseAuthorization();
```

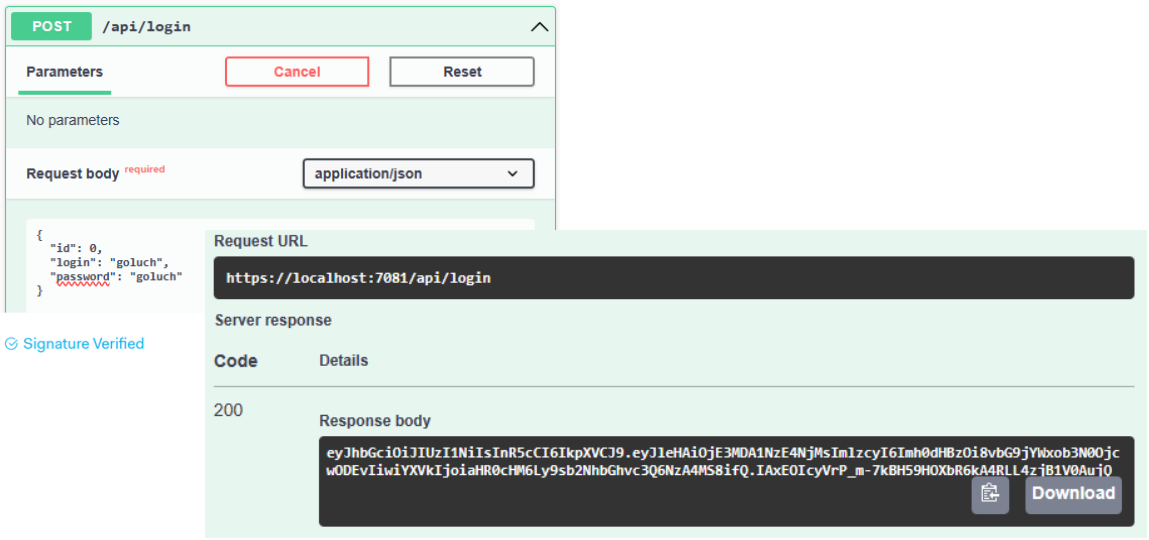
Do utworzenia tokena będą pomocne następujące klasy:

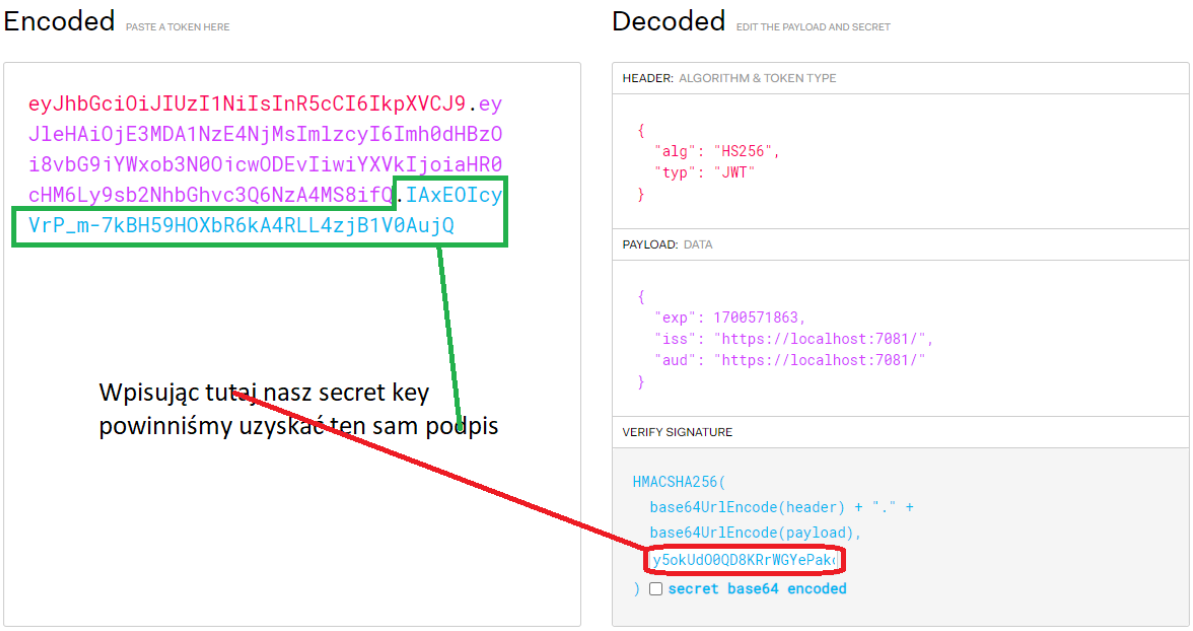
- `SymmetricSecurityKey` – reprezentuje symetryczny klucz kryptograficzny;
- `SigningCredentials` – reprezentuje klucz kryptograficzny (np. `SymmetricSecurityKey`) i algorytmy zabezpieczeń używane do generowania podpisu cyfrowego;
- `JwtSecurityToken` – reprezentuje token JWT, jako parametry można podać: `issuer`, `audience`, `claims`, `expires`, `notBefore`, `signingCredentials` (czyli obiekt klasy `SigningCredentials`).
- `JwtSecurityTokenHandler`().WriteToken – metoda serializująca JWT do JWE lub JWS zwracanego w postaci stringa.

Przykładowy kod:

```
var securityKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(Config.Key));
var credentials = new SigningCredentials(securityKey,
SecurityAlgorithms.HmacSha256);
var token = new JwtSecurityToken(Config.Issuer,
    Config.Issuer,
    null,
    expires: DateTime.Now.AddMinutes(120),
    signingCredentials: credentials);

return new JwtSecurityTokenHandler().WriteToken(token);
```

JWS po zdekodowaniu, `iss` i `aud` są identyczni ponieważ uruchamiamy ich na tej samej maszynie:

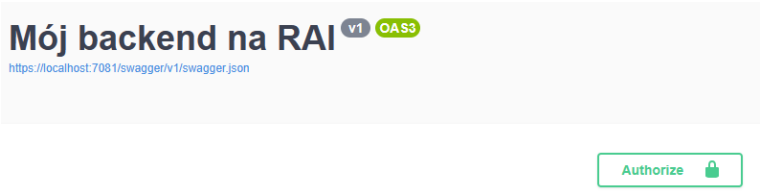




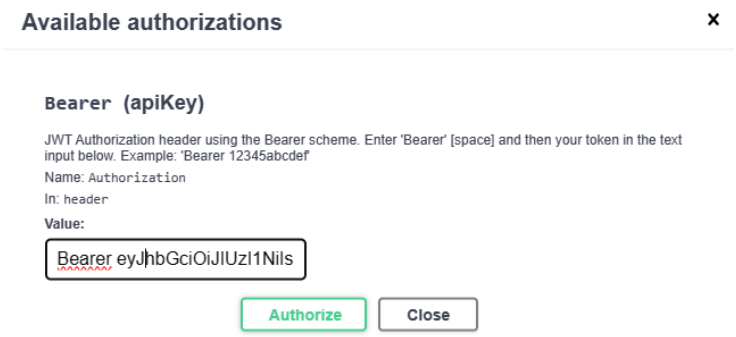
Jeśli chcemy dokonać uwierzytelnienia z poziomu UI Swaggera to musimy odpowiednio skonfigurować obiekt `SwaggerGenOptions`:

```
builder.Services.AddSwaggerGen((c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "M�j backend na RAI", Version = "v1" });
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Description = @"JWT Authorization header using the Bearer scheme.
                      Enter 'Bearer' [space] and then your token in the text input below.
                      Example: 'Bearer 12345abcdef'",
        Name = "Authorization",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });

    c.AddSecurityRequirement(new OpenApiSecurityRequirement()
      {
        {
          new OpenApiSecurityScheme
          {
            Reference = new OpenApiReference
              {
                Type = ReferenceType.SecurityScheme,
                Id = "Bearer"
              },
              Scheme = "oauth2",
              Name = "Bearer",
              In = ParameterLocation.Header,

            },
            new List<string>()
          }
        });
}));
```

Powinna pojawić się opcja uwierzytelniania;



Po wprowadzeniu tokena poprzedzonego ciągiem Bearer i spacją powinniśmy zostać uwierzytelnieni:



Po uwierzytelnieniu możemy już korzystać z API, oznacza to, że metody z naszego API oznaczone atrybutem `[Authorize]` będą dostępne. Jeśli chcemy udostępnić metody dla nieautoryzowanego użytkownika, np. możliwość rejestracji, wykorzystujemy atrybut `[AllowAnonymous]`.

Więcej o komunikacji w backendzie .NET w oparciu o bibliotekę Microsoft.AspNetCore.Authentication.JwtBearer tutaj: https://masterbranch.pl/uwierzytelnianie-w-api-czyli-bearer-token/.

W przypadku API opartego o javascript, `express.js` logika jest podobna, oczywiście należy wykorzystać dedykowane biblioteki np.: `express-bearer-token` i `swagger-ui-express`.

VI. JWT (Express)

Cel: Zapoznanie z implementacją uwierzytelniania JWT w aplikacji Express z użyciem biblioteki jsonwebtoken.

W przypadku API opartego o JavaScript i Express, logika jest bardzo podobna do tej przedstawionej w .NET . Również opieramy się na tokenie składającym się z nagłówka, payloadu i podpisu, który jest weryfikowany przy każdym zabezpieczonym żądaniu. Do implementacji wykorzystamy najpopularniejszą bibliotekę: `jsonwebtoken`. Na początku musimy ją zainstalować wraz z wymaganymi definicjami typów dla TypeScriptu:

```
npm install jsonwebtoken
npm install --save-dev @types/jsonwebtoken
```

Proces implementacji składa się z trzech głównych kroków:

Pierwszym krokiem jest utworzenie tokena dla uwierzytelnionego użytkownika, np. po poprawnym logowaniu. Endpoint ten (np. `/api/login`) jest publicznie dostępny. Do podpisania tokena wykorzystamy klucz `TOKEN_KEY` zdefiniowany wcześniej w pliku `.env`.

```
import { Router } from "express";
import jwt from "jsonwebtoken";

const router = Router();
const config = process.env;

router.post("/login", (req, res) => {
    // Krok 1: Weryfikacja użytkownika (w prawdziwej aplikacji: w bazie
danych)
```

```
    const { login, password } = req.body;

    // Uproszczona weryfikacja na potrzeby przykładu
    if (login === "goluch" && password === "goluch") {

        // Krok 2: Pobranie sekretnego klucza z .env
        const tokenKey = config.TOKEN_KEY;
        if (!tokenKey) {
            // Błąd serwera, jeśli klucz nie jest skonfigurowany
            return res.status(500).send("Brak klucza TOKEN_KEY w
konfiguracji!");
        }

        // Krok 3: Tworzenie tokena (Payload)
        const payload = {
            userId: "unikalne_id_uzytkownika_z_bazy",
            login: login
        };

        const token = jwt.sign(
            payload,
            tokenKey,
            { expiresIn: "1h" } // Opcje, np. czas wygaśnięcia
        );

        // Krok 4: Wysłanie tokena do klienta
        res.status(200).json({ token: token });

    } else {
        res.status(401).send("Niepoprawne dane logowania");
    }
});

export { router as authRoutes };
```

Następnie montujemy nowy router w głównym pliku aplikacji (np. `index.ts`), pamiętając, aby był on dostępny **przed** ewentualnymi zabezpieczeniami:

```
// w pliku index.ts
import { authRoutes } from "./API/routes/AuthRoutes"; // (dostosuj ścieżkę)
import { userRoutes } from "./API/routes/UserRoutes";

// ...
app.use(express.json()); // Middleware do parsowania JSON

// Publiczne trasy logowania i rejestracji
app.use("/api", authRoutes);

// Pozostałe trasy aplikacji
```

```
app.use("/api", userRoutes);
// ...
```

Kolejny krok to Weryfikacja tokena (Middleware). Musimy stworzyć "strażnika" (middleware), który będzie przechwytywał żądania do chronionych zasobów. Jego zadaniem jest sprawdzenie, czy do żądania dołączony jest poprawny token w nagłówku Authorization.

```
import { Request, Response, NextFunction } from "express";
import jwt from "jsonwebtoken";

const config = process.env;

// Rozszerzamy domyślny interfejs Request, aby móc dodać do niego dane
użytkownika
export interface RequestWithUser extends Request {
    user?: string | jwt.JwtPayload;
}

export const verifyToken = (req: RequestWithUser, res: Response, next:
NextFunction) => {

    // Oczekujemy tokena w nagłówku "Authorization: Bearer TOKEN"
    const authHeader = req.headers["authorization"];
    const token = authHeader && authHeader.split(' ')[1]; // Wyciągnięcie
samego tokena

    if (!token) {
        return res.status(403).send("Brak tokena uwierzytelniającego.");
    }

    const tokenKey = config.TOKEN_KEY;
    if (!tokenKey) {
        return res.status(500).send("Brak klucza TOKEN_KEY w konfiguracji!");
    }

    try {
        // Weryfikacja tokena przy użyciu sekretnego klucza
        const decoded = jwt.verify(token, tokenKey);

        // Dodajemy zdekodowane dane (payload) do obiektu żądania
        req.user = decoded;

    } catch (err) {
        return res.status(401).send("Nieprawidłowy lub nieważny token.");
    }

    // Jeśli wszystko jest OK, przechodzimy do następnego middleware lub
docelowej funkcji
    return next();
```

```
};
```

W trzecim kroku użyjemy middleware do ochrony tras. Teraz możemy użyć naszego middleware verifyToken, aby zabezpieczyć wybrane trasy:

```
import { Router } from "express";
import { InMemoryUserRepository } from
"../../infrastructure/InMemoryUserRepository"
import { GetAllUsers } from "../../use-cases/GetAllUsers";
import { UserController } from "../controllers/UserController";

// Import naszego middleware
import { verifyToken, RequestWithUser } from "../middleware/auth"; //
(dostosuj ścieżkę)

const router = Router();
const userRepository = new InMemoryUserRepository();
const getAllUsers = new GetAllUsers(userRepository);
const userController = new UserController(getAllUsers);

// Trasa /api/users jest teraz chroniona
// Każde żądanie musi mieć poprawny token Bearer
router.get("/users", verifyToken, (req, res) => {

    // Wewnątrz chronionej trasy mamy dostęp do danych użytkownika z tokena
    const requestWithUser = req as RequestWithUser;
    console.log("Dostęp do chronionej trasy przez:", requestWithUser.user);

    // Wywołanie kontrolera
    userController.getAll(req, res);
});

export { router as userRoutes };
```

W ten sposób zaimplementowaliśmy pełny cykl uwierzytelniania JWT w Express: publiczną trasę logowania generującą token oraz chronione trasy, które weryfikują ten token za pomocą dedykowanego middleware.

VII. MongoDB

Cel: Instalacja dokumentowej bazy danych MongoDB oraz podstawowe operacje jak: utworzenie i połączenie z nową instancją bazy oraz dodanie prostej kolekcji.

Dokumentowe bazy danych (NoSQL) uwalniają programistę od sztywnego schematu tabel relacyjnych baz danych[6]. Dane przechowywane są dokumentach (odpowiadającym rzędom w tablicy bazy danych SQL) CML, YAML, JSON lub binarnych, jak BSON. Dokumenty przechowywane są w kolekcjach (odpowiadających tablicom w bazach danych SQL) przypominających słowniki, ponieważ do każdego z nich przypisany jest określony

[6] https://docs.aws.amazon.com/documentdb/latest/developerguide/document-database-documents-understanding.html

klucz/identyfikator. Identyfikatory są zwykle indeksowane w bazie danych, aby przyspieszyć wyszukiwanie danych. Można dodawać lub usuwać pola (odpowiadające kolumnom w tablicy bazy danych SQL), zagnieżdżać dane w wielu warstwach otrzymując elastyczny model danych, który może dostosowywać się do ciągle zmieniających się potrzeb. W przypadku brakujących informacji – nie mamy pustych miejsc w kolumnach, jak ma to miejsce w przypadku baz SQL – są one po prostu pomijane. Zawartość dokumentów w kolekcjach jest klasyfikowana za pomocą metadanych pozwala to bazie na „rozumienie" jaki typ informacji przechowuje (adresy, numer telefonu). Język zapytań, umożliwia wysyłanie zapytań do dokumentów na podstawie metadanych lub jego treści. W ten sposób można np. pobrać wszystkie dokumenty, które zawierają tę samą wartość w określonym polu.

Z drugiej strony jeśli nie ma zgody co do tego, jak powinien wyglądać model danych, a każdy dokument w kolekcji zawiera duże różnice w układzie pól, może to doprowadzić do wielu problemów. Dotyczy to szczególnie baz danych o dużej objętości z siecią wzajemnie powiązanych danych.

Instalator bazy danych można pobrać z: https://www.mongodb.com/try/download/community. Możemy również uruchomić bazę danych w kontenerze Docker. W tym przypadku zaleca się przechowywać dane w zewnętrznym folderze:

```
docker run --name=my_mongo --
volume=/dysk/sciezka/do/pliku/:/data/db/ -p 27017:27017 -d mongo
/mongodb-community-server:latest
```

lub w wolumenie:

```
docker volume create mongodbdata
```

```
docker run --name=my_mongo -p 27017:27017 -v mongodbdata:/data/db/ -
d mongo/mongodb-community-server:latest
```

Do zarządzania bazą z wiersza poleceń będziemy jeszcze potrzebowali MongoDB Shell'a: https://www.mongodb.com/try/download/shell, a dokładnie plik: mongosh.exe z folderu bin, który po prostu uruchamiamy.

Tworzymy nową bazę danych:

```
use <nazwa_bazy_danych>
```

Dodajemy kolekcję:

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

Sprawdźmy czy baza oraz kolekcja została dodana:

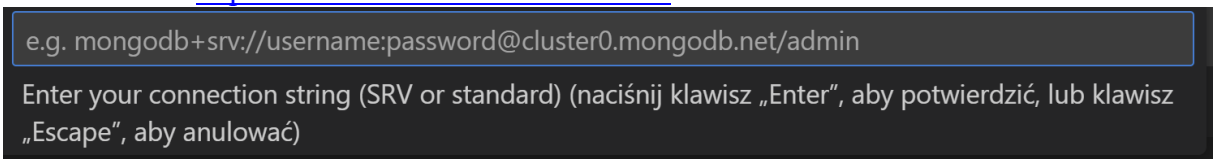
```
show dbs
show collections
```

oraz czy element kolekcji został poprawnie dodany do bazy:

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

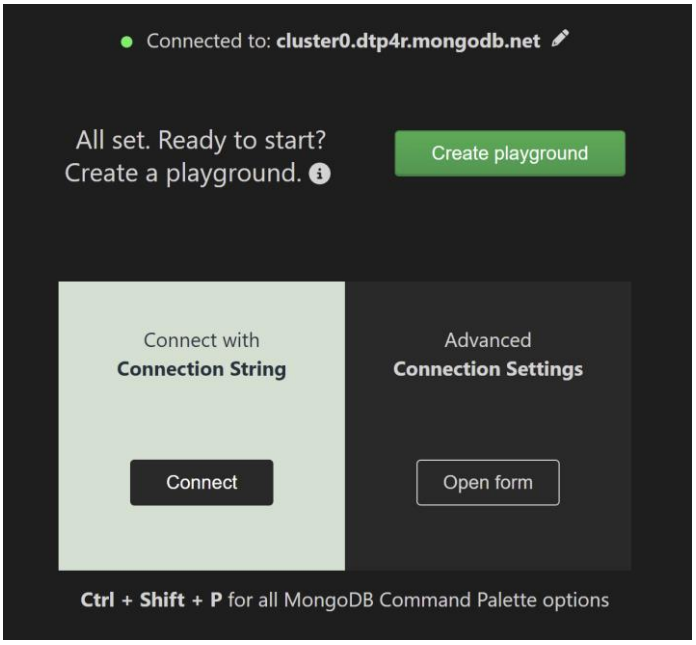
```
goluch> show collections
users
goluch> db.users.find()
[
  {
    _id: ObjectId("65524ec22a6d9f1a24c77b56"),
    name: 'student',
    password: 'student'
  }
]
```

Jeśli korzystamy z VS Code przydatna może być rozszerzenie: MongoDB for VS Code, która ułatwia pracę z MongoDB. W Command Palette (Ctrl+Shift+P) wyszukujemy "MongoDB: Connect" i wybieramy "Connect with Connection String.". Podajemy connection string w sugerowanym formacie, pamiętając, że następujące znaki : / ? # [ ] @ użyte w haśle muszą być przekonwertowane do formatu percent encoding. W tym celu można użyć dowolnego kodera online: https://www.url-encode-decode.com/.



Domyślny ConnectionString bazy uruchomionej lokalnie (zainstalowanej lokalnie lub uruchomionej w kontenerze) będzie wyglądał tak: mongodb://127.0.0.1:27017.

W przypadku niepowodzenia należy sprawdzić czy nasz adres ip jest na białej liście. Po poprawnym połączeniu jesteśmy gotowi do utworzenia pierwszej piaskownicy (playground).



https://www.mongodb.com/docs/mongodb-vscode/playgrounds/

```
// MongoDB Playground
// To disable this template go to Settings | MongoDB | Use Default Template
For Playground.
// Make sure you are connected to enable completions and to be able to run a
playground.
// Use Ctrl+Space inside a snippet or a string literal to trigger completions.

// Select the database to use.
use('mongodbVSCodePlaygroundDB');

// The drop() command destroys all data from a collection.
// Make sure you run it against the correct database and collection.
db.sales.drop();

// Insert a few documents into the sales collection.
db.sales.insertMany([
  { '_id': 1, 'item': 'abc', 'price': 10, 'quantity': 2, 'date': new
Date('2014-03-01T08:00:00Z') },
  { '_id': 2, 'item': 'jkl', 'price': 20, 'quantity': 1, 'date': new
Date('2014-03-01T09:00:00Z') },
  { '_id': 3, 'item': 'xyz', 'price': 5, 'quantity': 10, 'date': new
Date('2014-03-15T09:00:00Z') },
  { '_id': 4, 'item': 'xyz', 'price': 5, 'quantity':  20, 'date': new
Date('2014-04-04T11:21:39.736Z') },
  { '_id': 5, 'item': 'abc', 'price': 10, 'quantity': 10, 'date': new
Date('2014-04-04T21:23:13.331Z') },
  { '_id': 6, 'item': 'def', 'price': 7.5, 'quantity': 5, 'date': new
Date('2015-06-04T05:08:13Z') },
  { '_id': 7, 'item': 'def', 'price': 7.5, 'quantity': 10, 'date': new
Date('2015-09-10T08:43:00Z') },
```

```
  { '_id': 8, 'item': 'abc', 'price': 10, 'quantity': 5, 'date': new
Date('2016-02-06T20:20:13Z') },
]);

// Run a find command to view items sold on April 4th, 2014.
db.sales.find({ date: { $gte: new Date('2014-04-04'), $lt: new Date('2014-04-
05') } });

// Build an aggregation to view total sales for each product in 2014.
const aggregation = [
  { $match: { date: { $gte: new Date('2014-01-01'), $lt: new Date('2015-01-
01') } } },
  { $group: { _id: '$item', totalSaleAmount: { $sum: { $multiply: [ '$price',
'$quantity' ] } } } }
];

// Run the aggregation and open a cursor to the results.
// Use toArray() to exhaust the cursor to return the whole result set.
// You can use hasNext()/next() to iterate through the cursor page by page.
db.sales.aggregate(aggregation);
```

VIII. Mongoose[7] (Express)

Cel: Zapoznanie z biblioteką modelowania danych Mongoose.

Mongoose to nakładka komunikująca się z bazą danych MongoDB. Zapewnia proste, oparte na schematach modelowanie danych aplikacji umożliwiając programistom wymuszenie określonego schematu w warstwie aplikacji. Obejmuje wbudowane rzutowanie typów, haki do logiki biznesowej, walidację modelu, i inne funkcje mające na celu ułatwienie pracy z MongoDB. Instalacja Mongoose za pomocą npm:

```
npm install mongoose
```

Mongoose pozwala od razu zacząć korzystać z modeli danych, bez czekania, na ustanowienie połączenia z MongoDB[8]. Połaczenie z lokalną bazą danych (przykłady):

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

Przykład połączenia z bazą w chmurze (np. MongoDB Atlas) z wykorzystaniem zmiennych środowiskowych:

```
// Upewnij się, że require("dotenv").config() jest na górze pliku const config
= process.env;
```

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

```
const mongoConnectionString =
mongodb+srv://${config.MONGO_LOGIN}:${config.MONGO_PASSWORD}@<clusterX>.<instan
cja>.mongodb.net/test?retryWrites=true&w=majority;
// Użyj nowego stringa do połączenia
mongoose.connect(mongoConnectionString);
```

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:

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const userSchema = new Schema({
  login: { type: String,unique: true, lowercase: true, required: true  },
  password: { type: String, required: true },
                    ...
});
module.exports = mongoose.model('user', userSchema);
```

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:

```
const UserModel = require('<sciezka/do/models/Kolekcja>');

const newUser = new UserModel({
  login: 'admin',
  password: 'admin',
                    ...
});
newUser.save((err) => {
  mongoose.disconnect();
  if (err) throw err;
  console.info('User saved successfully!');
});
```

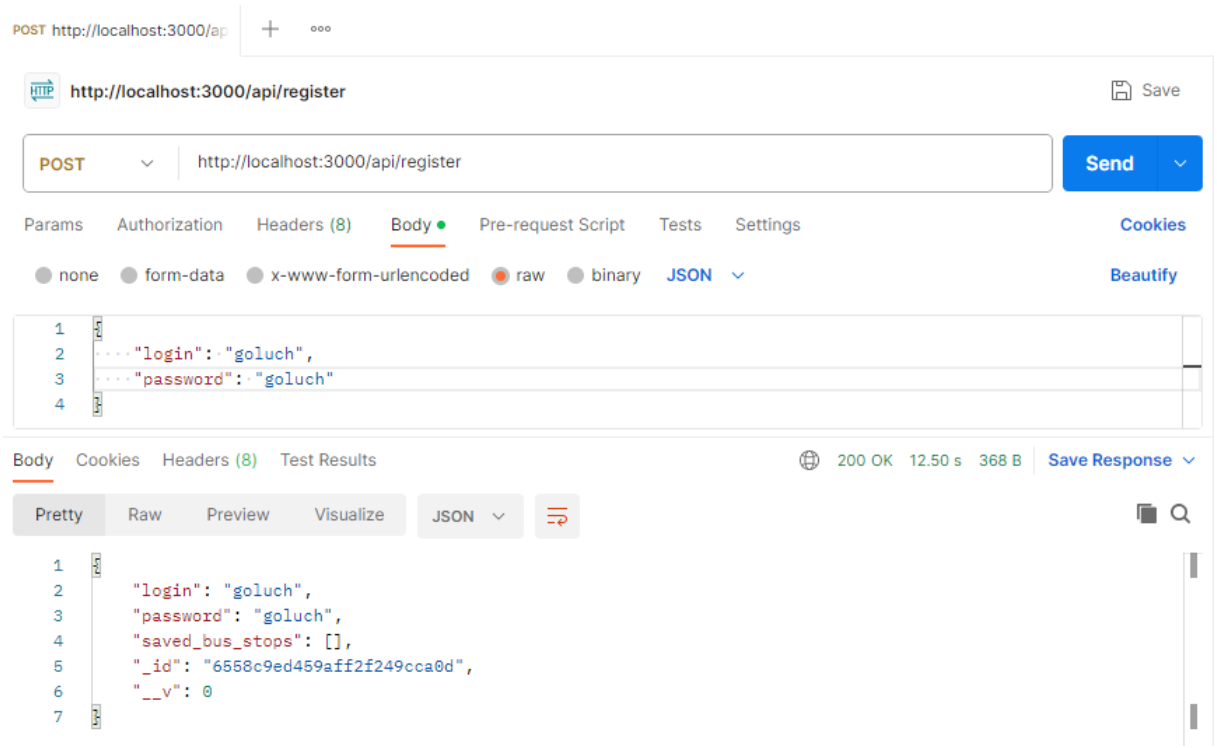
Wyszukanie kolekcji z bazy danych:

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

Użycie w mongo.js:

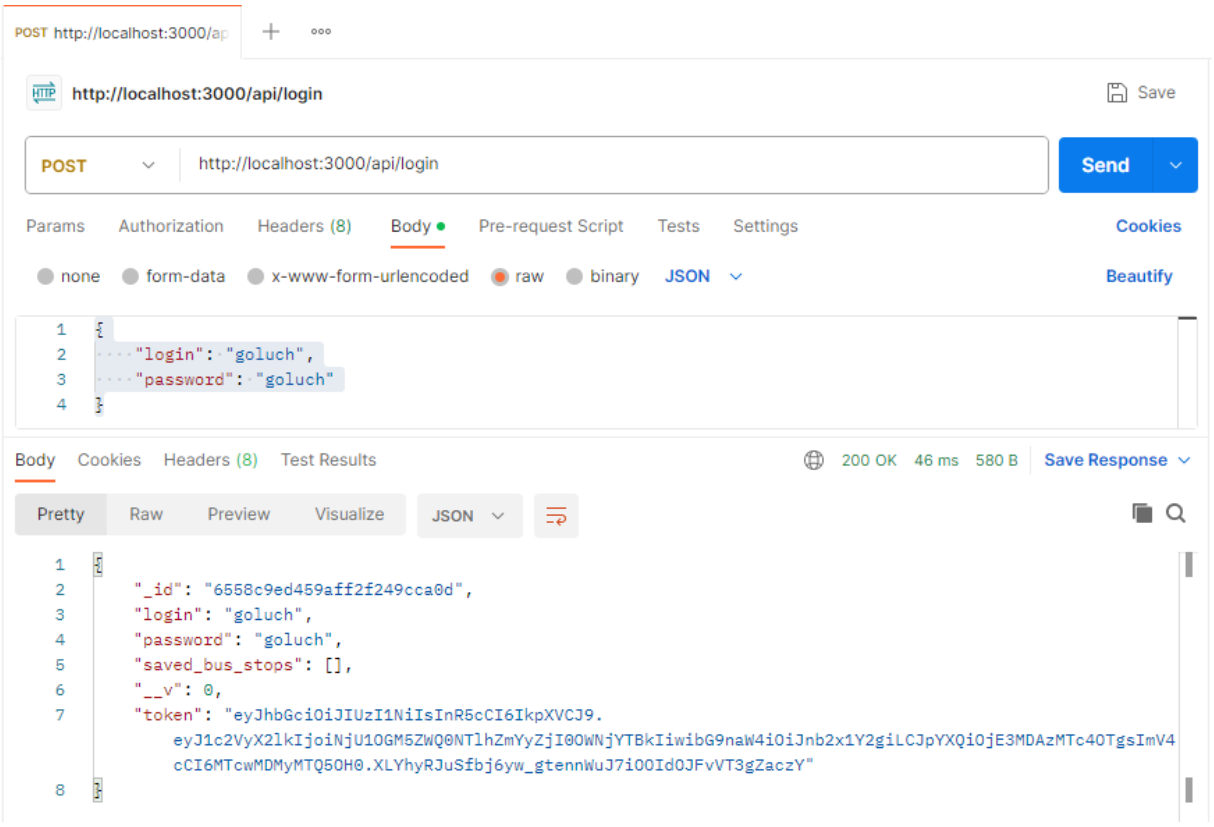
```
require("dotenv").config();
```

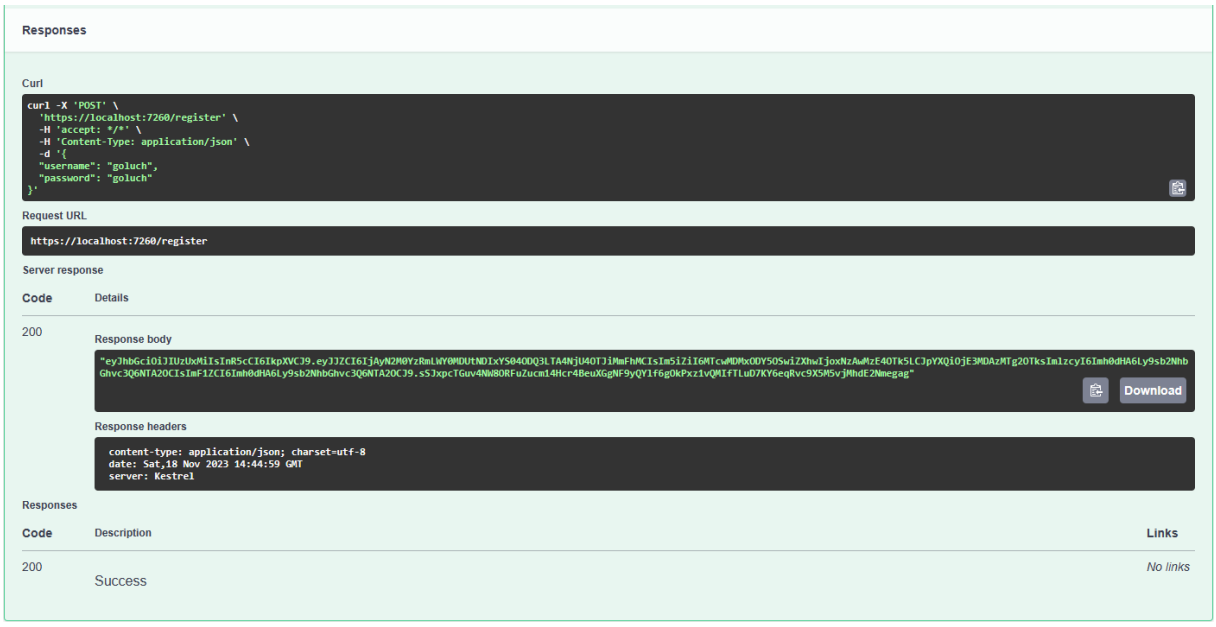
```
const config = process.env;
          ...
var mongoConnectionString = "mongodb+srv://" + config.MONGO_LOGIN + ":" +
config.MONGO_PASSWORD +
"@<clusterX>.<instancja>.mongodb.net/?retryWrites=true&w=majority";
```




```
{
    "login": "goluch",
    "password": "goluch"
}
```

```
{
    "login": "goluch",
    "password": "goluch",
    "saved_bus_stops": [],
    "_id": "6558c9ed459aff2f249cca0d",
    "__v": 0
}
```





IX. Czysta Architektura (Express)

Cel: Zapoznanie z zasadami uniezależniania logiki biznesowej zgodnie z zasadami czystej architektury wujka Bob'a (Robert C. Martin).

Zadaniem czystej architektury jest zapewnienie łatwej skalowalności, utrzymania i testowania aplikacji. Kluczowe zasady czystej architektury:

- Niezależność – podstawowa logika biznesowa nie powinna zależeć od zewnętrznych bibliotek, interfejsu użytkownika, baz danych oraz frameworków.

- Testowalność – aplikacja powinna być łatwa do przetestowania bez polegania na zewnętrznych systemach.
- Elastyczność – powinna być łatwa do zmiany lub zastąpienia części aplikacji bez wpływu na inne jej części.

Typowy podział aplikacji implementującej czystą architekturę na warstwy (kolejność według zależności):

1) Warstwa domeny – zawiera logikę biznesową, encje/obiekty wartości i interfejsy. Ta warstwa jest niezależna od innych warstw.

Dla obiektów wartości warto stworzyć klasę abstrakcyjną aby nie powtarzać logiki powiązanej z ich porównywaniem, może być przydatne wykorzystanie modułu `shallow-equal-object`:

```
import { shallowEqual } from "shallow-equal-object";

interface ValueObjectProps {
    [index: string]: any;
}

export abstract class ValueObject<T extends ValueObjectProps> {
    public readonly props: T;

    constructor (props: T) {
        this.props = Object.freeze(props);
    }

    public equals (vo?: ValueObject<T>) : boolean {
    if (vo === null || vo === undefined) {
      return false;
    }
    if (vo.props === undefined) {
      return false;
    }
    return shallowEqual(this.props, vo.props)
  }
}
```

Implementacja klasy obiektu wartości:

```
import { ValueObject } from "../abstractions/ValueObject";

interface UserNameProps {
    username: string
}

export class UserName extends ValueObject<UserNameProps> {
    constructor(props: UserNameProps) {
        super(props)
```

```
    }

    get Value() {
        return this.props
    }

    get username() {
        return this.props.username
    }
}
```

Użycie obiektu wartości:

```
import { UserName } from "./userName";

let vo = new UserName({ username: 'admin' })
let value = vo.Value
let username = vo.username

console.log(vo)
//  UserName { props: { username: 'admin' } }
console.log(value)
// { username: 'admin' }
console.log(username)
// admin
```

Definiujemy interfejs w którym deklarujemy obiekty wartości jako tablice dowolnych typów indeksowane stringami. W konstruktorze zostają one zamrożonych za pomocą `Object.freeze()` i stają się niezmienne, a ich wartości nie mogą być ponownie przypisywane. Metoda `shallowEqual` porównuje wartości tablic props. Podklasy Value Object można rozszerzyć, aby zawierały metody ułatwiające, takie jak `greaterThan(vo?: ValueObject<T>)` lub `lessThan(vo?: ValueObject<T>)`. Ma to sens w przypadku takich obiektów wartości jak: `LoggingLevels` lub `BusinessRatings`.

Definicja klasy encji z użyciem obiektu wartości:

```
import { UserName } from "../value-objects/UserName";

export class User {
    constructor(
      public readonly id: string,
      public author: UserName,
      public lastLogonDate: Date
    ) {}
}
```

Definicja interfejsu repozytorium:

```
import { User } from "../entities/User"
```

```
export interface UserRepository {
  findAll(): Promise<User[]>;
  findById(id: string): Promise<User | null>;
  create(User: User): Promise<User>;
  update(User: User): Promise<void>;
  delete(id: string): Promise<void>;
}
```

2) Warstwa aplikacji/przypadków użycia – zawiera przypadki użycia aplikacji lub reguły biznesowe, które można wykonać w aplikacji. Współdziałają one z warstwą domeny i są niezależne od używanego frameworka lub bazy danych.

```
import { UserRepository } from "../domain/interfaces/UserRepository";

export class GetAllUsers {
  constructor(private userRepository: UserRepository) {}

  async execute() {
    return await this.userRepository.findAll();
  }
}
```

3) Warstwa infrastruktury – zawiera implementacje interfejsów zdefiniowanych w warstwie Domeny, takie jak połączenia z bazą danych.

Implementacja operacji CRUD repozytorium klasy User. Repozytorium dla uproszczenia przykładu przechowuje użytkowników w prywatnym polu klasy:

```
import { User } from "../domain/entities/User";
import { UserRepository } from "../domain/interfaces/UserRepository";

export class InMemoryUserRepository implements UserRepository {
  private users: User[] = [];

  async findAll(): Promise<User[]> {
    return this.users;
  }

  async findById(id: string): Promise<User | null> {
    return this.users.find(user => user.id === id) || null;
  }

  async create(user: User): Promise<User> {
    this.users.push(user);
    return user;
  }

  async update(user: User): Promise<void> {
    const index = this.users.findIndex(b => b.id === user.id);
```

```
    if (index !== -1) {
      this.users[index] = user;
    }
  }

  async delete(id: string): Promise<void> {
    this.users = this.users.filter(user => user.id !== id);
  }
}
```

4) Warstwa interfejsu – zawiera kontrolery, trasy i dowolny inny kod zależny od frameworka internetowego.

Implementacja kontrolera `UserController`, udostępniającego metodę `getAll` zwracającą wszystkich użytkowników:

```
import { Request, Response } from "express";
import { GetAllUsers } from "../../use-cases/GetAllUsers";

export class UsesController {
  constructor(private getAllUsers: GetAllUsers) {}

  async getAll(req: Request, res: Response) {
    const users = await this.getAllUsers.execute();
    res.json(users);
  }
}
```

Implementacja routera obsługującego żądanie GET wykorzystując kontroler `UserController`:

```
import { Router } from "express";
import { InMemoryUserRepository } from
"../../infrastructure/InMemoryUserRepository"
import { GetAllUsers } from "../../use-cases/GetAllUsers";
import { UserController } from "../controllers/UserController";

const router = Router();

const userRepository = new InMemoryUserRepository();
const getAllUsers = new GetAllUsers(userRepository);
const userController = new UserController(getAllUsers);

router.get("/users", (req, res) => userController.getAll(req, res));

export { router as userRoutes };
```

Wywołanie funkcji middleware dodającej routing `userRoutes`:

```
import express from "express";
import { userRoutes } from "./API/routes/UserRoutes";

const app = express();

app.use(express.json());
app.use("/api", userRoutes);

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Konwencja nazewnictwa dla TypeScriptu jest dość dowolna, ważne aby była spójna. Jedną z przykładowych może być stosowanie zasady aby komponenty, klasy JS i główne funkcje przechowywać w oddzielnych plikach o nazwie pliku takiej same nazwie. Możemy wtedy zastosować różne case:

- TitleCase/PascalCase – dla komponentów React/Vue i klas JS;
- camelCase – dla normalnych funkcji JS;
- kebab-case – dla katalogów, bibliotek lub plików z wieloma funkcjami użytkowymi.

X. Artykuły i przykładowe kody

Cel: Dodatkowa literatura oraz kody na których można się wzorować podczas realizacji laboratorium.

Artykuły (Express):

1) https://paulallies.medium.com/clean-architecture-typescript-express-api-b90846794998
2) https://dev.to/dipakahirav/modern-api-development-with-nodejs-express-and-typescript-using-clean-architecture-1m77
3) https://khalilstemmler.com/articles/typescript-value-object/
4) https://henrikgronvall.com/articles/typescript-value-object-explained/

Kody (Express):

1) https://github.com/jameswillis99/clean_architecture_typescript_express