Uniwersytet Pedagogiczny im. Komisji Edukacji Narodowej
Instytut Bezpieczeństwa i Informatyki

PROJEKT INŻYNIERSKI
DOKUMENTACJA PROJEKTOWA

World of Notes

wykonany przez:

Michał Topa
Nr albumu: 148806
&
Artur Węgrzyn
Nr albumu: 148780
&
Damian Wilk
Nr albumu: 148785

pod opieką:
dr Wojciech Gwizdała
mgr Wojciech Baran

Kraków 2023

Spis treści

1 Serwer

1.1 Architektura i technologie

Język programowania

Aplikacja została napisana w języku **Typescript**. Jest to utrzymywany przez Microsoft open-sourcowy nadzbiór JavaScriptu. Język ten zaliczany jest to wysokopoziomowych, jednak posiada statyczne typowanie. Ta kombinacja okazała się być idealna do naszej aplikacji, ponieważ pozwala w szybkim tempie pisać bezpieczny kod.

- Typescript - strona projektu
- Typescript - dokumentacja

Środowisko uruchomieniowe

Node.js – jest to narzędzie do uruchamiania JavaScript w środowisku innym niż przeglądarka internetowa.

- Node.js - strona projektu
- Node.js - dokumentacja

Framework NestJs

Nest. JS jest frameworkiem do budowania aplikacji po stronie serwerowej na platformie Node.js. Napisany jest w języku TypeScript. Stworzył go Kamil Myśliwiec w 2017 roku. NestJS zapewnia nie tylko zestaw narzędzi potrzebny dla budowania projektów, ale także wprowadza pewne zasady, które gwarantują stosowanie najlepszych praktyk.

- NestJs - strona projektu
- NestJs - dokumentacja

Swagger

Swagger to framework, który pozwala wizualizować i korzystać z aplikacji API, przy okazji tworząc dokumentację.

- Swagger - strona projektu
- Swagger - dokumentacja

Serverless, Amazon Web Services

Aplikacja została napisana w modelu **Serverless**. W podejściu tym integrujemy się z dostawcą chmury(u nas AWS), który odpowiedzialny jest za wykonanie fragmentu kodu poprzez dynamiczną alokację zasobów. W modelu tym naliczanie opłat odbywa się jedynie za zasoby faktycznie wykorzystane do uruchomienia kodu.

- AWS - strona
- AWS - dokumentacja

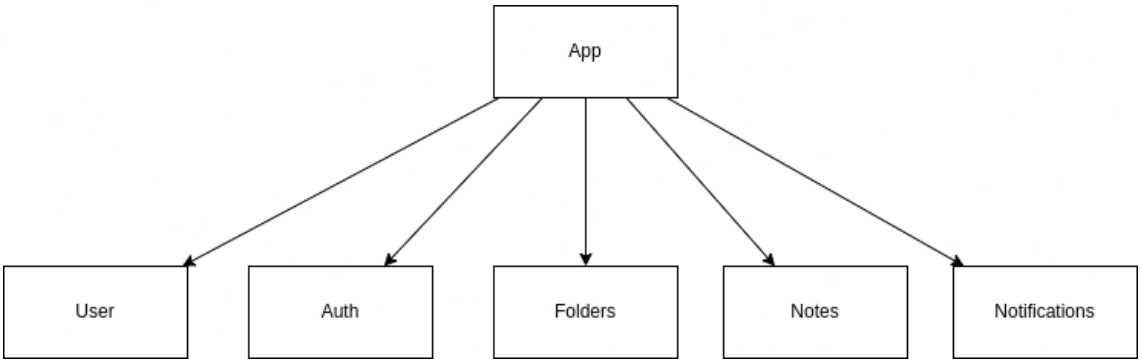
1.2 Szczegóły implementacji

1.2.1 Struktura, podział projektu

Aplikacja składa się z **modułów**, które służą do podziału aplikacji ze względu na domenę danych oraz zakres funkcjonalności. Moduły natomiast zawierają:

- **kontrolery**, czyli klasy odpowiedzialne za przechwytywanie przychodzących zapytań, walidację danych oraz zwracanie odpowiedzi do klienta
- **serwisy**, w których zawiera się główna część logiki - zapytania do bazy, integracje z API AWS.

Poniższy diagram ilustruje organizację najważniejszych modułów w aplikacji



Organizacja plików i katalogów została przygotowana z myślą, aby imitować modułowy podział aplikacji.
Oto drzewo plików w przypadku kilku przykładowych klas:

```
src
└── app.module
    ├── auth
    │   ├── auth.module.ts
    │   ├── auth.service.ts
    │   └── auth.user.strategy.ts
    ├── emails
    │   ├── emails.module.ts
    │   ├── emails.service.ts
    │   ├── emails.service.spec.ts
    │   └── emails.templates.ts
    └── notes
        ├── notes.module.ts
        ├── notes.dto.ts
        ├── notes.types.ts
        ├── notes.service.ts
        ├── notes.controller.ts
        └── notes.controller.spec.ts
```

1.2.2 Zakres funkcjonalności

Zakres funkcjonalności przy poszczególnych modułach:

- User
  - logowanie
  - rejestracja
  - pobieranie i zarządzanie danymi użytkowników
- Auth
  - uwierzytelnianie
- Folders
  - zarządzanie danymi kategorii notatek użytkowników
    - pobieranie
    - dodawanie
    - edycja
    - usuwanie
- Notes
  - zarządzanie danymi notatek użytkowników
    - pobieranie
    - dodawanie
    - edycja
    - usuwanie
- Notifications
  - zarządzanie danymi powiadomień użytkowników
    - pobieranie
    - dodawanie
    - usuwanie
  - planowanie powiadomień
  - wysyłanie maili

1.2.3 Walidacja danych

Do walidacji danych użytkownika korzystamy z biblioteki **class-validator**. Tworzymy specjalne klasy nazywane **DTO** (Data Transfer Object), następnie za pomocą wbudowanych w bibliotekę adnotacji określamy wymagany przez nas kształt danych. Wysłanie requestu z błędnymi parametrami skutkuje odpowiedzią z kodem 400(Bad Request) i opisem niezgodności.

Listing 1: Przykładowa klasa DTO

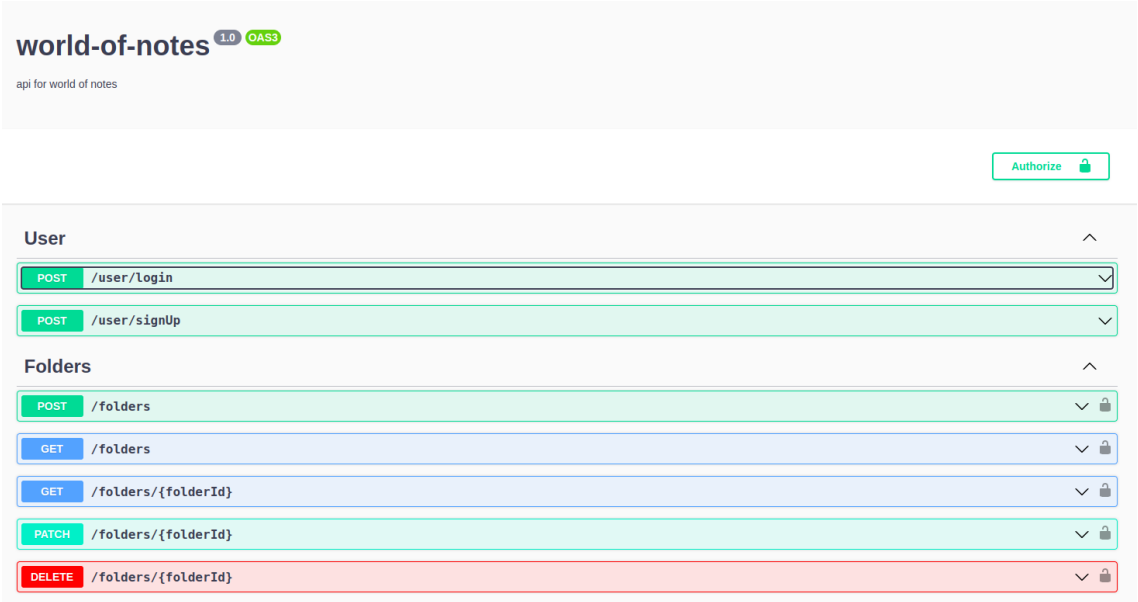
```
export class CreateReoccurringNotificationDto {
  @IsEnum(Days)
  @IsNotEmpty()
  dayOfWeek: Days;

  @IsInt()
  @IsNotEmpty()
  @Min(0)
  @Max(23)
  hours: number;

  @IsInt()
  @IsNotEmpty()
  @Min(0)
  @Max(59)
  minutes: number;
}
```

Listing 2: Wykorzystanie w kontrolerze

```
export class NotificationsController {
  constructor(private readonly notificationsService: NotificationsService) {}

  @Post('reoccurring')
  createReoccurring(
    @Body() createNotificationDto: CreateReoccurringNotificationDto,
    @UserId() userId: string,
  ): Promise<NotificationModel> {
    return this.notificationsService.createReoccurring(
      createNotificationDto,
      userId,
    );
  }
}
```

1.2.4 Integracja Swaggera

W projekcie spory nacisk położyliśmy na wygodne i pewne połączenie między serwerem i interfejsem użytkownika. W tym celu zdecydowaliśmy się na użycie Swaggera. Narzędzie to pozwoliło nam bez większych problemów utworzyć dokumentację pozwalającą przeglądać oraz testować endpointy. Na jej podstawie tworzymy później kod do komunikacji wraz z potrzebnymi typami.



Link do wygenerowanej dokumentacji: Dokumentacja Swagger

1.2.5 Uwierzytelnianie i autoryzacja

Logowanie odbywa się za pomocą standardowej procedury z loginem i hasłem. Poprawne zapytanie o login oraz utworzenie konta zwraca specjalny token **JWT**. Token ten jest weryfikowany przy każdym kolejnym endpoincie, aby ograniczyć dostęp do prywatnych danych tylko zalogowanym użytkownikom. Odbywa się to za pomocą specjalnej adnotacji zwanej **guardem**.

Listing 3: Guard weryfikujący token JWT

```
export const USER_JWT_STRATEGY = 'user-jwt';

@Injectable()
export class UserJwtStrategy extends PassportStrategy(
  Strategy,
  USER_JWT_STRATEGY,
) {
  constructor(private readonly config: ConfigService<Config, true>) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: config.get('jwt', { infer: true }).secret,
    });
  }

  async validate(payload: JwtTokenPayload) {
    return payload;
  }
}

export class UserGuard extends AuthGuard(USER_JWT_STRATEGY) {}
```

Listing 4: Wykorzystanie w kontrolerze

```
@Controller('folders')
@ApiTags('Folders')
@UseGuards(UserGuard) // Podpiecie guarda pod wszystkie endpointy kontrolera
@ApiBearerAuth()
export class FoldersController {
  constructor(private readonly foldersService: FoldersService) {}
...
```

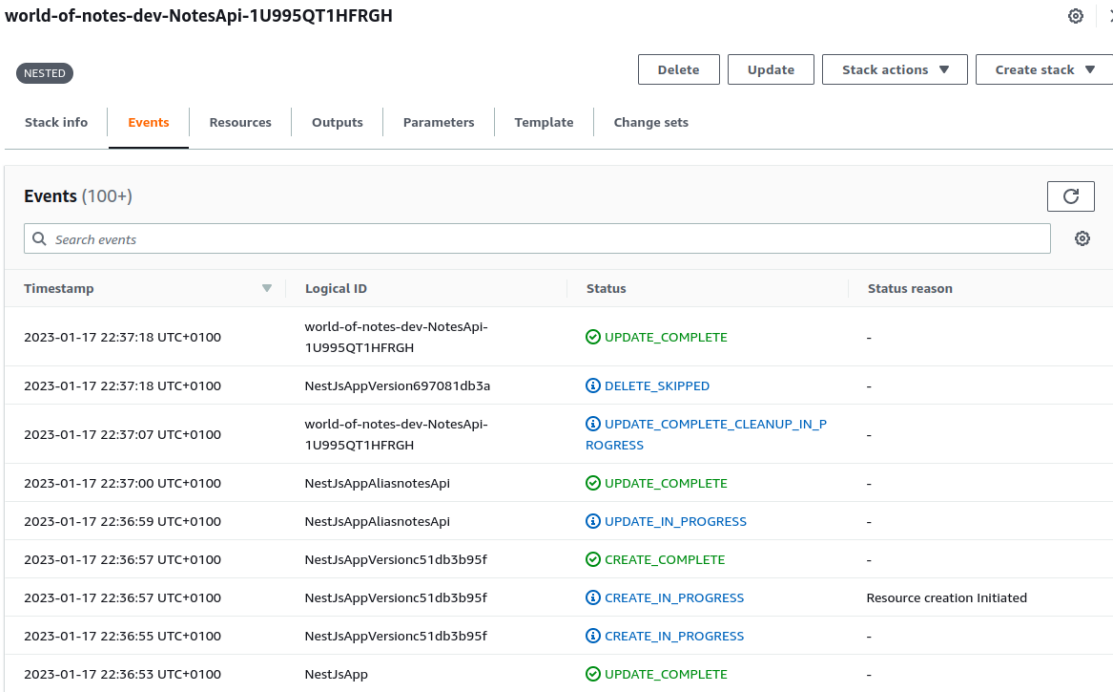
1.2.6 Budowanie i wdrażanie

Aplikacja budowana jest na **Lambdę**, czyli usługę obliczeniową, która uruchamia kod w odpowiedzi na zdarzenia i automatycznie zarządza zasobami obliczeniowymi wymaganymi przez ten kod. W tym celu posiłkujemy się biblioteką **@vendia/serverless-express**.

Konfigurację lambdy, zarówno z innymi serwisami AWS tworzymy w specjalnych plikach .yml (osobne repozytorium **notes-infrastructure**), gdzie określamy ich

parametry oraz dependencje. Wdrożenie nowej wersji aplikacji odbywa się poprzez zbudowanie, utworzenie zipa z kodem, następnie wrzucenie na AWS nowej wersji templatki za pomocą usługi **CloudFormation**.

Logi przy wdrażaniu z usługą CloudFormation:



1.2.7 Integracja z serwisami AWS

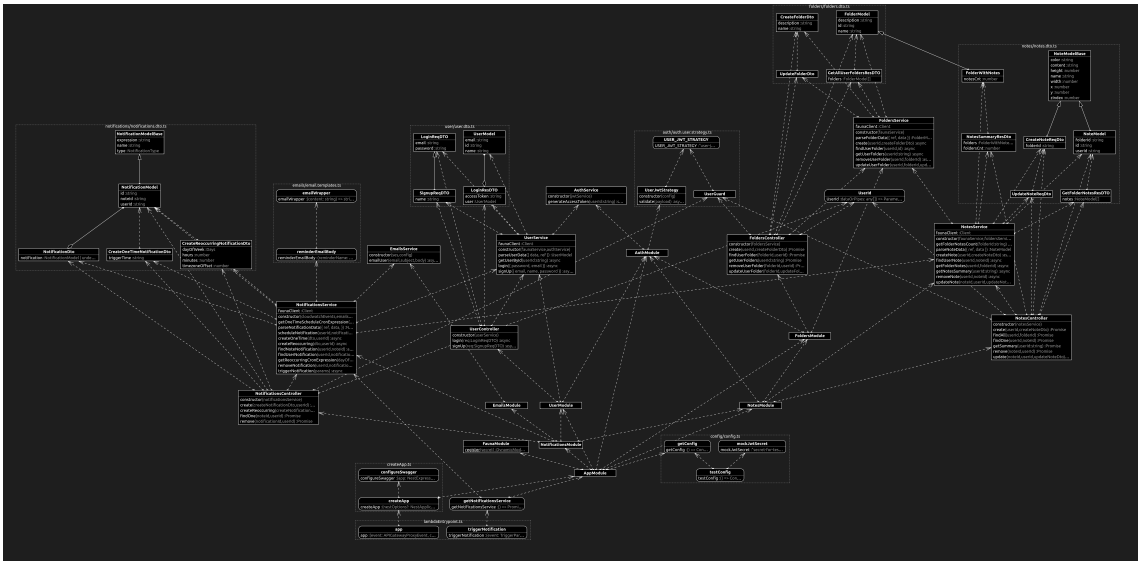
Lista wykorzystywanych serwisów AWS w aplikacji:

- Lambda
  - kod aplikacji zawierający endpointy reagujące na zapytania http
  - Handler do wysyłania zaplanowanych notyfikacji emailowych
- SES(Simple Email Service) - usługa do wysyłania emaili
- CloudFormation - automatyzacja infrastruktury dla AWS, która wdraża zasoby w sposób powtarzalny, testowalny i podlegający audytowi.
- EventBridge - usługa obsługująca zdarzenia. W przypadku naszej aplikacji służy to planowania wywołań lambdy wysyłającej notyfikacje dla użytkownika Integracja z tymi usługami odbywa się poprzez AWS SDK dla Javascriptu.

Listing 5: Wykorzystanie SDK do wysyłania emaili usługą SES

```
@Injectable()
export class EmailsService {
  constructor(
    private readonly ses: SES,
    private readonly config: ConfigService<Config, true>,
  ) {}

  public async emailUser(
    email: string,
    subject: string,
    body: string,
  ): Promise<void> {
    const { sender } = this.config.get('emails', {
      infer: true,
    });

    await this.ses
      .sendEmail({
        Destination: { ToAddresses: [email] },
        Message: {
          Body: {
            Html: { Data: body },
          },
          Subject: { Data: subject },
        },
        Source: `<${sender}>`,
      })
      .promise();
  }
}
```

1.3 Diagram klas UML - w projekcie



1.4 Testowanie

Testy w aplikacji są pisane w oparciu o bibliotekę "jest". Przed każdym testem stawiana jest na nowo aplikacja przy użyciu specjalnego modułu do testowania wraz z pustą bazą danych.

Listing 6: Setup testów

```
1  beforeEach(async () => {
2    // tworzenie nowej, pustej bazy danych, wykonanie na niej migracji
3    const { secret, childFauna } = await setupTestDatabase();
4    // helper do wypelniania bazy danymi
5    factory = new FaunaFactory(childFauna);
6
7    // tworzenie testowego modulu
8    const module: TestingModule = await Test.createTestingModule({
9      imports: [
10       ConfigModule.forRoot({ load: [testConfig], isGlobal: true }),
11       FaunaModule.register(secret),
12       NotesModule,
13     ],
14   }).compile();
15
16   app = module.createNestApplication(undefined as any, { bodyParser: true });
17   await app.init();
```

W aplikacji posiadamy dwa rodzaje testów:

- testy kontrolerów - testy e2e = imitują zapytania http w rzeczywistych warunkach. Weryfikowane są odpowiedzi serwisów, zmiany w bazie.
- testy serwisów - testy jednostkowe, sprawdzające działanie konkretnych metod w klasach.

Poniżej przedstawiam output z komendy puszczającej testy w aplikacji. Można przy nim wyraźnie zobaczyć nazwy testów napisanych dla konkretnych plików i endpointów.

Listing 7: Output z testów

```
1  -> % yarn test
2  yarn run v1.22.19
3  $ jest --runInBand --verbose
4   PASS  src/notes/notes.controller.spec.ts (66.349 s)
5    notes controller test
6      POST /notes
7            should add note (3354 ms)
8            fails when folder does not belong to user (2551 ms)
9            fails when required data is missing (2419 ms)
10           fails without authentication token (2608 ms)
11     GET /notes?folderId
12           returns all notes from specified folder (2362 ms)
```

```
          fails when folder does not belong to user (2306 ms)
          fails without authentication token (2794 ms)
    GET /notes/:noteId
          returns selected note (2434 ms)
          fails when note does not exist (2370 ms)
          fails when note does not belong to user (2638 ms)
          fails without authentication token (2363 ms)
    GET /notes/summary
          returns user notes summary (2466 ms)
          works fine when user does not have any notes yet (2682 ms)
          fails without authentication token (2332 ms)
    PATCH /notes/:noteId
          correctly updates note data (2348 ms)
          moves note to another folder (2737 ms)
          fails when folder does not belong to user (2285 ms)
          fails when note does not exist (2177 ms)
          fails without authentication token (2481 ms)
    DELETE /notes/:noteId
          removes user note (2263 ms)
          fails when note does not exist (2254 ms)
          fails when note does not belong to user (2459 ms)
          fails without authentication token (2104 ms)

PASS  src/notifications/notifications.controller.spec.ts (40.417 s)
 notes controller test
    POST /notifications/one-time
          should add notification (2573 ms)
          fails when note does not belong to user (2353 ms)
          fails when trigger date is not valid (2405 ms)
          fails without authentication token (2282 ms)
    POST /notifications/reoccurring
          should add notification (2246 ms)
          fails when note does not belong to user (2267 ms)
          fails when params are not valid (2177 ms)
          fails without authentication token (2488 ms)
    GET /notifications/:noteId
          returns notification associated with selected note (2214 ms)
          works fine in case there is no notifications (2185 ms)
          fails when note does not exist (2486 ms)
          fails when note does not belong to user (2083 ms)
          fails without authentication token (2291 ms)
    DELETE /notifications/:notificationId
          removes notification from database, removes aws event rule (2404 ms)
          fails when notification does not exist (2130 ms)
          fails when notification does not belong to user (2319 ms)
          fails without authentication token (2295 ms)

PASS  src/folders/folders.controller.spec.ts (37.335 s)
 folders controller test
    GET /folders
          should return all user folders (2204 ms)
          fails without authentication token (2215 ms)
    GET /folders/:folderId
          should return specific user folder (2318 ms)
          fails when folder does not belong to user (2187 ms)
```

```
          fails without authentication token (2394 ms)
    POST /folders
          should add folder (2186 ms)
          fails when user already has a folder with the same name (2129 ms)
          fails when required data is missing (2506 ms)
          fails without authentication token (2111 ms)
    PATCH /folders/:folderId
          should update folder (2382 ms)
          allows to update partial data (2289 ms)
          fails when folder does not belong to user (2179 ms)
          fails without authentication token (2580 ms)
    DELETE /folders/:folderId
          should delete folder (2398 ms)
          fails when folder does not belong to user (2271 ms)
          fails without authentication token (2445 ms)

 PASS  src/user/user.controller.spec.ts (27.497 s)
  use controller test
    POST /login
          should return access token (2498 ms)
          fails when user does not exist (2049 ms)
          when password is invalid (2339 ms)
          email is invalid (2316 ms)
          password is missing (2361 ms)
          password is too short (2415 ms)
    POST /signUp
          creates user account (2132 ms)
          fails when user already exists (2221 ms)
          email is invalid (2179 ms)
          password is missing (2058 ms)
          name is missing (2321 ms)
          password is too short (2156 ms)

 PASS  src/notifications/notifications.service.spec.ts
  NotificationsService
    getReoccurringCronExpression
          adds offset properly (2 ms)
          with negative offset
          adds hours when needed
          subtract hours when needed
          adds days when needed
          subtract days when needed
          takes days from another week if needed (1 ms)
          takes days from previous week if needed

 PASS  src/emails/emails.service.spec.ts
  Emails service test
        should send email to user (2 ms)

Test Suites: 6 passed, 6 total
Tests:       77 passed, 77 total
Snapshots:   0 total
Time:        174.868 s
Ran all test suites.
Done in 175.97s.
```

2 Baza danych

2.1 Technologie

FaunaDb

W przypadku bazy danych również postawiliśmy na rozwiązanie typu serverless - **Fauna**. Jest to baza **NoSQL** posiadająca swój własny język do tworzenia query - **FQL**

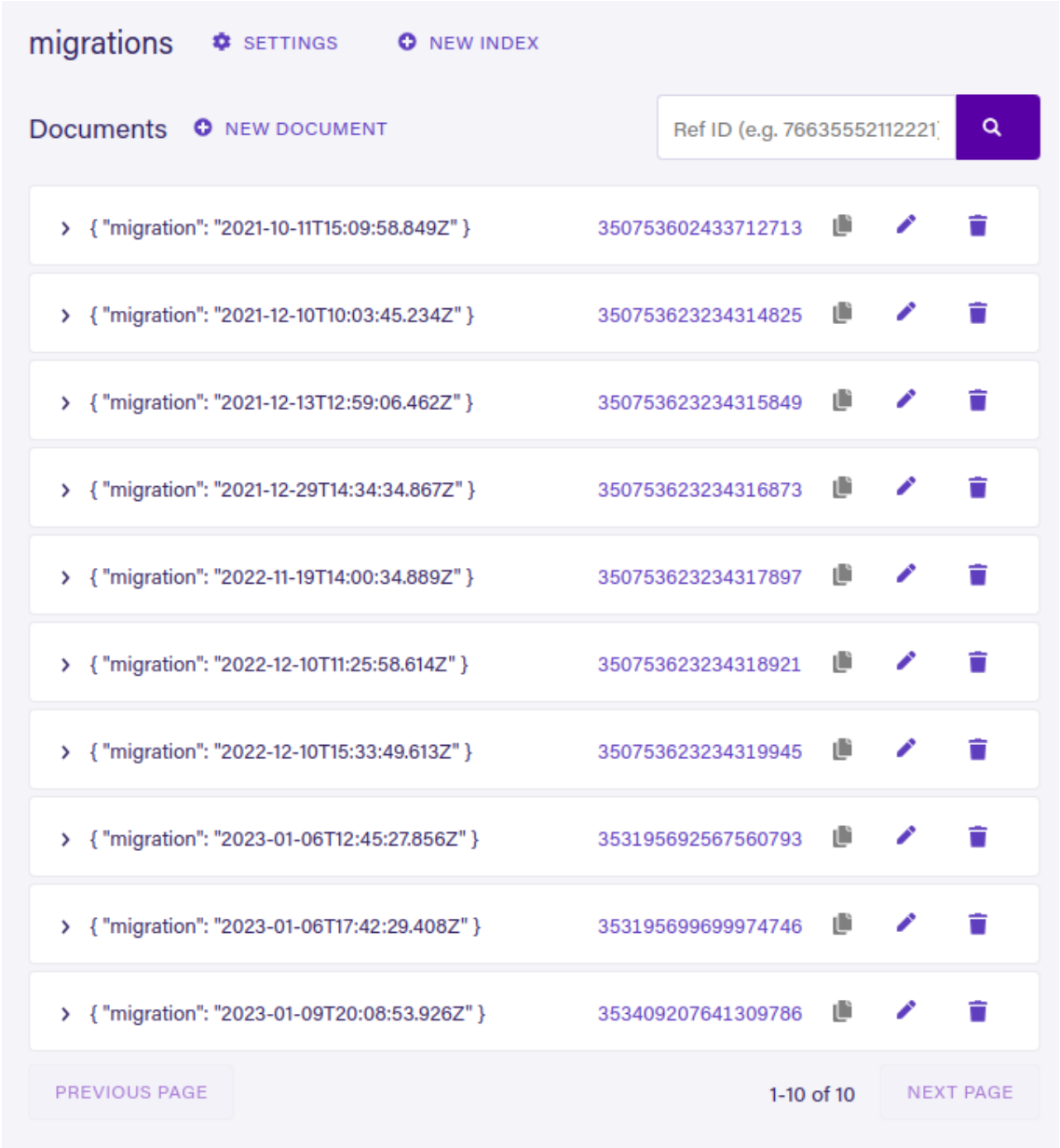
- Fauna - strona projektu
- Fauna - dokumentacja

2.1.1 Docker

2.2 Migracje, obsługa w projekcie

Migracje

Przy projektowaniu baz danych sporym wyzwaniem jest zachowanie spójności między różnymi środowiskami. Z pomocą przychodzi nam biblioteka @fauna-labs/fauna-schema-migrate. Za pomocą systemu tworzenia **migracji** pomaga nam utrzymać jednakowe tabele, indexy w lokalnym, testowym oraz produkcyjnym środowisku.
Migracje w inżynierii oprogramowania odnoszą się do zarządzania kontrolowanymi wersjami, przyrostowymi i odwracalnymi zmianami w schematach relacyjnych baz danych. Innymi słowy są to generowane w odpowiedniej kolejności instrukcje do tworzenia oraz modyfikowania struktury bazy.
W bazie tworzy się specjalna tabela **migrations**, dzięki czemu narzędzie wie, które instrukcje zostały już wdrożone, a które nie.



Rysunek 1: Tabela migracji nas stronie internetowej fauny

Obsługa zapytań w NestJs

Do połączenia z bazą w aplikacji napisaliśmy specjalny moduł, który korzystając ze zmiennych środowiskowych inicjuje połączenie. Moduł ten jest globalny, przez co jest dostępny we wszystkich serwisach.

Listing 8: Moduł do połączenia z bazą

```
@Module({})
export class FaunaModule {
  static register(secret?: string): DynamicModule {
    return FaunadbModule.forRootAsync({
      imports: [ConfigModule],
      useFactory: (config: ConfigService<Config, true>) => {
        const fdbConfig: FaunadbModuleOptions = {
          ...config.get('db', {
            infer: true,
          }),
        };

        if (secret) {
          fdbConfig.secret = secret;
        }

        return fdbConfig;
      },
      inject: [ConfigService],
    });
  }
}
```

Następnie, za pomocą języka FQL możemy pisać zapytania do bazy. Poniższy kod przedstawia zapisanie notatki użytkownika.

Listing 9: Przykład użycia FQL w serwisie

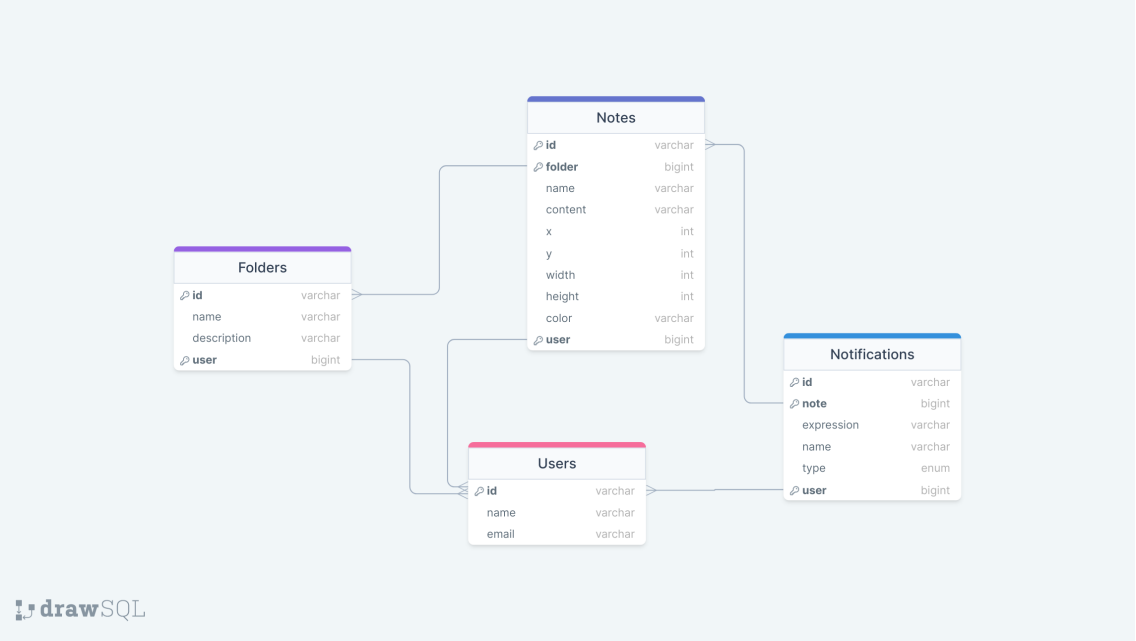
```
import { query as q } from 'nestjs-faunadb';

    const note = {
      data: {
        ...noteData,
        folder: q.Ref(q.Collection('Folders'), folderId),
        user: q.Ref(q.Collection('Users'), userId),
      },
    };

    const noteRes = await this.faunaClient.query<NoteQueryResult>(
      q.Create(q.Collection('Notes'), note),
    );
```

2.3 Lokalny development

Fauna jest usługą serverlesową, przez co nie stawia się jej samemu na swojej maszynie, jednak aby móc szybko ją sprawnie rozwijac i testować, zdecydowaliśmy się na użycie dockerowego kontenera **fauna/faunadb**. Zawiera on instancję bazy danych działającą tak samo, jak jej produkcyjna wersja, jednak można ją wystartować na swoim komputerze.
Kontener oferuje tak zwane **child databases**, z których korzystamy do testów - przed każdym testem tworzymy nową instancję, przez co mamy pewność, że mamy czyste środowisko.

2.4 Projekt UML

Relacje w FaunaDb działają trochę inaczej niż w przypadku SQL, mimo to uważamy, że poniższy diagram adekwatnie przedstawia podział tabel oraz połączenia między nimi w naszym systemie:



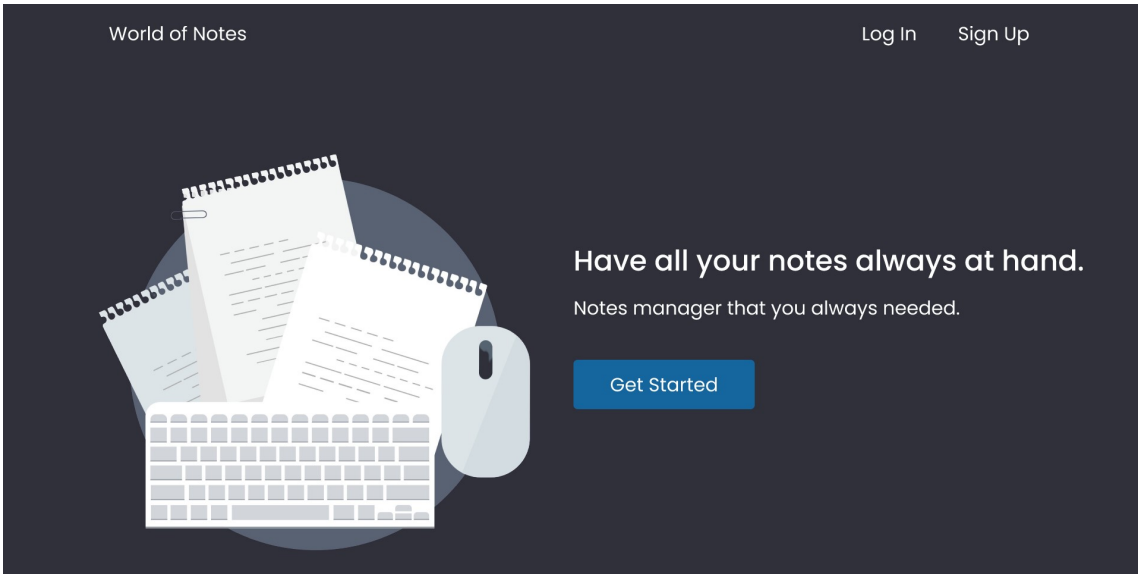
3 Interfejs użytkownika

3.1 Projekt graficzny

Widok interfejsu użytkownika został zaplanowany i zaprojektowany przy pomocy aplikacji Figma [1].

Figma jest aplikacją internetową służącą do projektowania interfejsów użytkownika. Bazuje na grafice wektorowej. Jej funkcje pozwalają tworzyć pojedyncze elementy, komponenty wraz z wariantami. Każdy zaprojektowany element automatycznie jest konwertowany na kod arkuszów stylów CSS, którego można używać w docelowym projekcie.

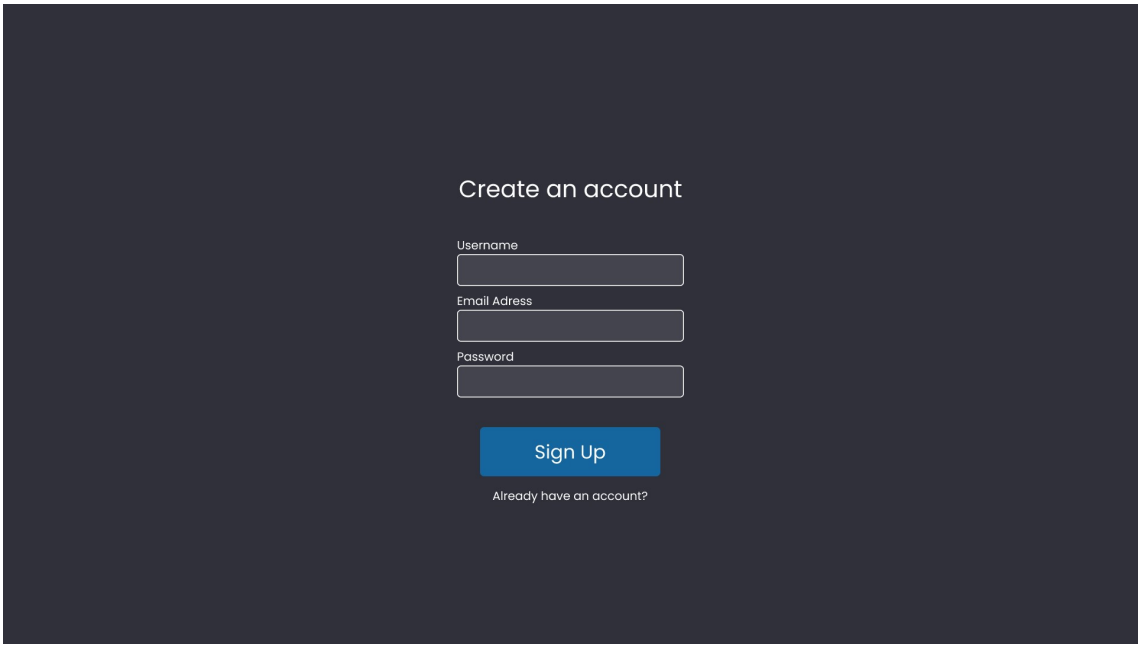
Została wstępnie zdefiniowana szata graficzna strony startowej. Zostały wybrane wiodące kolory oraz czcionka: ”Poppins”. Na stronie startowej została użyta grafika pochodząca z portalu Pixabay [2]. Grafika została użyta na zmodyfikowanej przez Pixabay licencji ”Creative Commons Zero”dostępnej pod adresem:
https://pixabay.com/service/license/



Rysunek 2: Projekt graficzny strony startowej

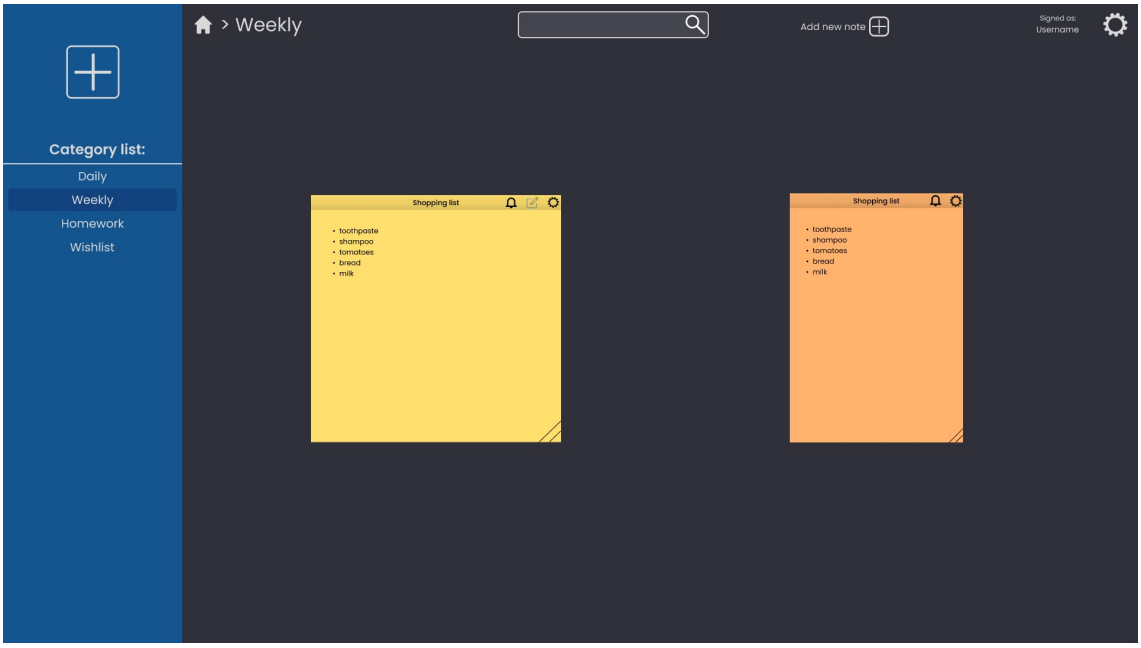
Następnie zostały zaprojektowane komponenty przycisków, pól danych wejściowych oraz dalszy plan poruszania się po aplikacji.

Został zaprojektowany widok strony rejestracji oraz logowania wraz z przejściem między tymi stronami dla wygody użytkownika w przypadku błędnego kliknięcia.



Rysunek 3: Projekt graficzny strony rejestracji

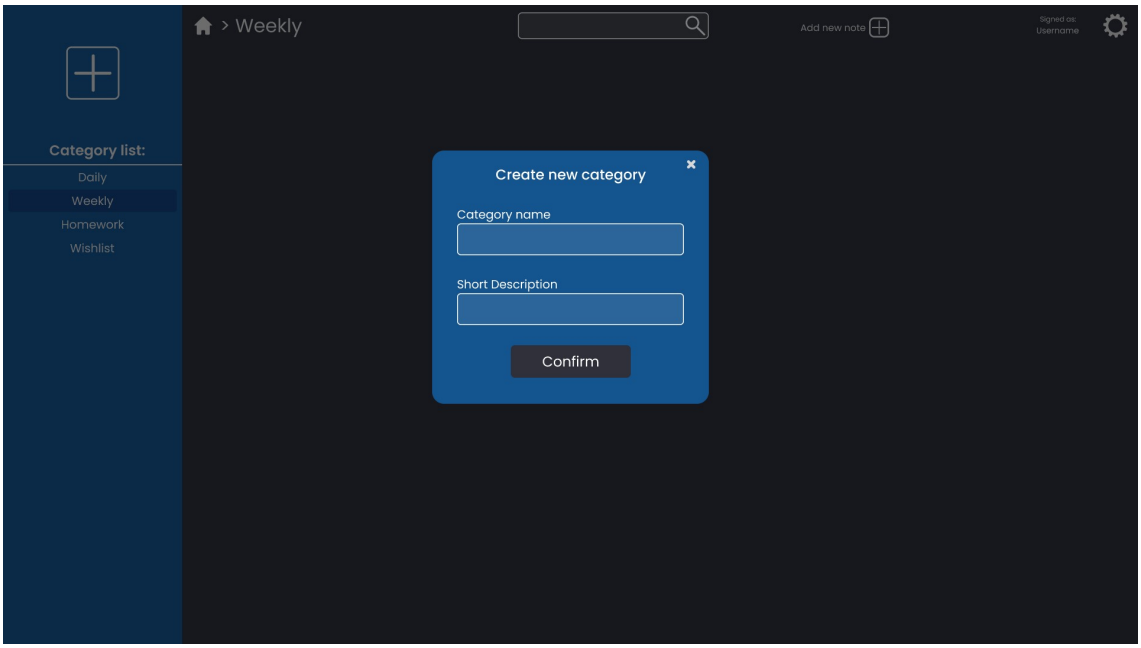
Następnie został zaprojektowany widok panelu użytkownika składający się z panelu bocznego (sidebar), który ma być zawsze widoczny i zawiera przycisk do dodawania kategorii notatek oraz listę istniejących kategorii. Header zawierający ścieżkę z informacją o obecnej lokalizacji oraz wykonywanej akcji; przycisk ustawień.
Widok planszy aplikacji, na której można rozmieszczać notatki:



Rysunek 4: Projekt graficzny panelu użytkownika z planszą notatek

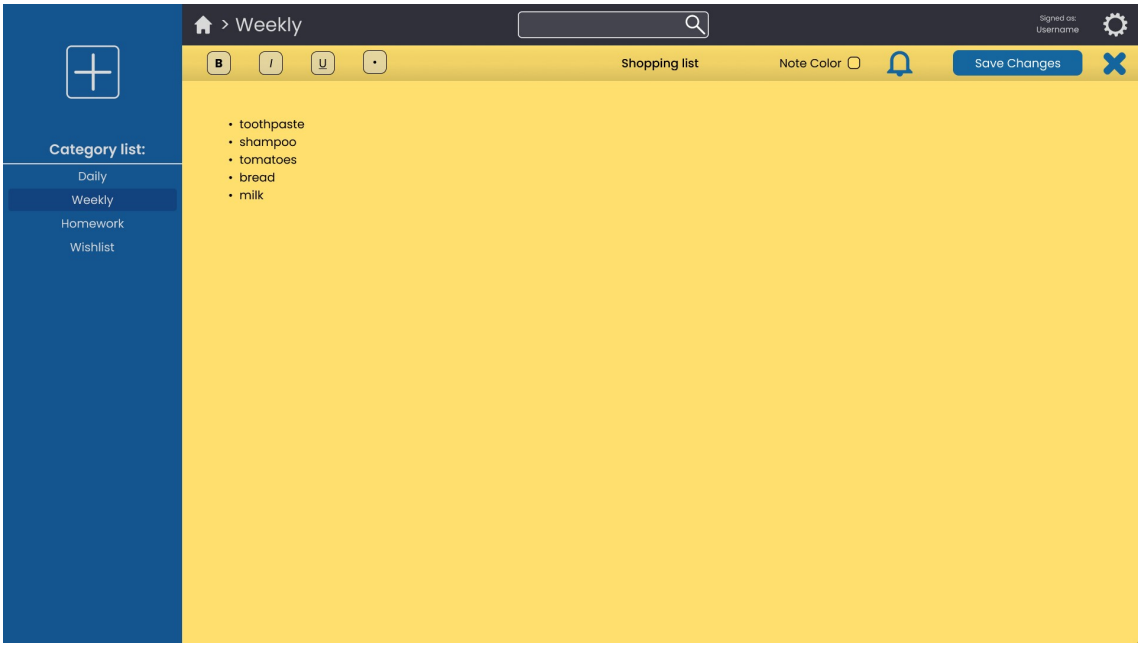
Wyskakujące okienka (popup) działają wraz z przyciemnieniem reszty aplikacji dla

lepszych odczuć użytkownika.



Rysunek 5: Projekt graficzny popupów

Wstępny widok edycji notatki wraz z ustawianiem powiadomień mailowych do notatek:



Rysunek 6: Projekt graficzny edycji notatki

Pełen projekt graficzny aplikacji znajduje się pod adresem: Projekt graficzny

3.2 Architektura i technologie

Język programowania

Aplikacja została napisana w języku **Typescript**. Jest to utrzymywany przez Microsoft open-sourcowy nadzbiór JavaScriptu. Język ten zaliczany jest to wysokopoziomowych, jednak posiada statyczne typowanie. Ta kombinacja okazała się być idealna do naszej aplikacji, ponieważ pozwala w szybkim tempie pisać bezpieczny kod.

- Typescript - strona projektu
- Typescript - dokumentacja

Środowisko uruchomieniowe

Node.js – jest to narzędzie do uruchamiania JavaScript w środowisku innym niż przeglądarka internetowa.

- Node.js - strona projektu
- Node.js - dokumentacja

Framework React

React. JS jest frameworkiem do budowania aplikacji po stronie frontendowej na platformie Node.js. Został napisany w JavaScript. Stworzyło go Meta Platforms. React zapewnia nie tylko zestaw narzędzi potrzebny dla budowania interfejsu użytkownika, ale także wprowadza pewne zasady, które gwarantują stosowanie najlepszych praktyk.

- React - strona projektu
- React - dokumentacja

Swagger

Swagger to framework, który pozwala wizualizować i korzystać z aplikacji API, przy okazji tworząc dokumentację.

- Swagger - strona projektu
- Swagger - dokumentacja

Redux

Redux to biblioteka JavaScript. Służy do zarządzania i centralizacji stanu aplikacji. Najczęściej jest używany w połączeniu z Reactem, tak jak w tym przypadku.

- Redux - strona projektu
- Redux - dokumentacja

Material UI

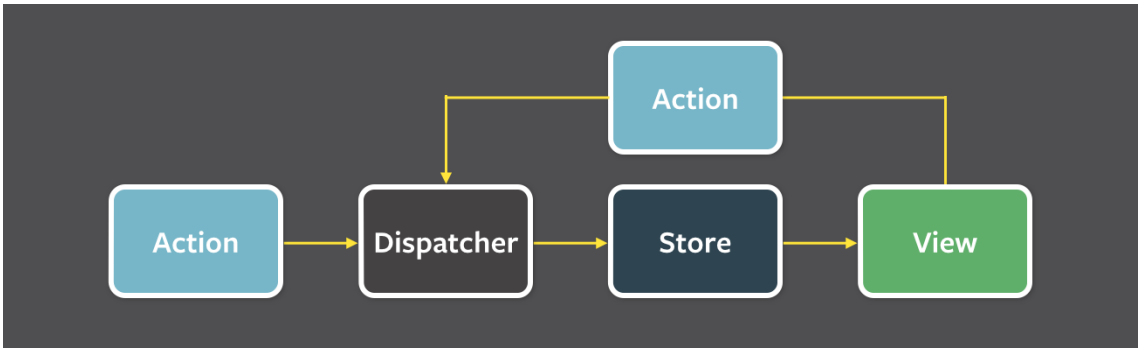
Material UI to biblioteka komponentów używanych do budowy aplikacji, których front-end opiera się o React.js. Pozwala w szybszy i bardziej uporządkowany sposób zarządzać stylami projektu.

- Material UI - strona projektu
- Material UI - dokumentacja

Architektura Flux

Architektura Flux określa sposób przechowywania oraz modyfikowania danych w aplikacji. Polega na utworzeniu obiektu centralnego - **store**, z którego danych możemy korzystać w dowolnym miejscu w projekcie. Przy podejściu tym tworzymy **reducer**, który reagując na różne **akcje** modyfikuje w określony sposób nasze dane. W przypadku naszej aplikacji zdecydowaliśmy się na użycie bilbioteki **redux**, wraz z rozszerzeniem jej funkcjonalności - redux-toolkit. Paczki te oferują narzędzia do konfiguracji store w projekcie, jak również zestaw funkcji do tworzenia reducerów oraz akcji. Ich twórcy wprowadzili specjalny obiekt nazywany **slice**, który pozwala na szybkie i czytelne tworzenie wszystkich niezbędnych komponentów reduxa.

Diagram flux:



- Flux - strona

- Redux - strona projektu
- Redux - dokumentacja
- redux-toolik - strona projektu
- redux-toolik - dokumentacja

selektory

W aplikacji, za pomocą narzędzia createSelector tworzymy specjalne funkcje zwane selektorami. Metody te przyjmują reduxowy state jako argument i zwracają dowolny wynik utworzony na jego podstawie. Dzięki **memoizacji** mamy pewność, że kalkulacje w naszych funkcjach będą się odbywać tylko w momencie, gdy zmieniają się nasze dane w store. Podejście to gwarantuje wysoką wydajność.

sagi

Za tworzenie sag odpowiada biblioteka redux-saga. Jest to narzędzie zintegrowane z reduxem, pozwalające z pomocą generatorów zarządzać efektami pobocznymi w aplikacji. Oferuje szeroki wachlarz funkcji do zarządzania asynchronicznymi operacjami oraz do integracji, komunikacji z innymi komponentami reduxa.

- redux-saga - strona projektu
- redux-saga - dokumentacja

3.3 Szczegóły implementacji

3.3.1 Struktura, podział projektu

Projekt podzielony jest na:

- **components** - biblioteka ostylowanych, reużywalnych komponentów.
- **pages** - konkretne widoki aplikacji.
- **redux** - system, który zarządza stanem aplikacji podzielony na **reducery** - czyli miejsce przechowywania stanu wraz z akcjami modyfikującymi go, **selectory** - funkcje zwracające i modyfikujące stan aplikacji oraz **redux-sagi** (funkcje generatory) służące do zarządzania side effectami, między innymi wywoływane są w nich zapytania do backendu oraz akcje zmiany stanu aplikacji.
- **theme** - użyte są tutaj narzędzia Material UI do zarządzania stylami komponentów React'a oraz tworzenia motywów dla całej aplikacji.
- **assets** - miejsce na pliki, między innymi obrazki oraz czcionki.
- **swagger** - skonfigurowany system generujący na podstawie backendu moduły do komunikacji z serwerem.
- **router** - część zarządzająca obecną lokalizacją oraz URL'em aplikacji.

3.3.2 Pobieranie danych

Poszczególne etapy pobierania i wyświetlania danych (na przykładzie logowania):

- **Konfiguracja reduxowych akcji**.
  Pierwszym krokiem jest stworzenie reducera oraz akcji.

Listing 10: Utworzenie akcji ”login”

```
1         login: (state, _action: PayloadAction<LoginReqDTO>) => state,
```

- **Dispatch akcji**
  Następnym krokiem jest emisja utworzonej przez nas akcji w odpowiednim momencie. W naszym przypadku - w funkcji reagującej na zdarzenie zatwierdzenia formularza logowania.

Listing 11: Wywołanie akcji ”login”w komponencie

```
1   const logInButtonClick = async (values: LoginValues) => {
2     dispatch(
3       sessionActions.login({
4         email: values.email,
5         password: values.password,
6       })
7     );
8   };
9
10 ...
11
12 <form onSubmit={handleSubmit(logInButtonClick)}>
```

- **Saga - obsługa operacji asynchronicznych**
  Za pomocą biblioteki redux-saga tworzymy specjalną specjalną funkcję zwaną **saga**. Konfigurujemy naszą sagę, aby reagowała na emisję akcji ”login”.

Listing 12: Podpięcie sagi ”loginSaga”pod akcję ”sessionActions.login”

```
1 takeEvery(sessionActions.login.type, loginSaga)
```

- **Wykonanie zapytania na serwer, odbiór danych z serwera**
  W sadze wywołane zostaje zapytanie na serwer, które poprzednio zostało wygenerowane narzędziem Swagger.

Listing 13: Wykonanie zapytania na serwer w ”loginSaga”

```
1  try {
2     responseLogin =
3       yield * call(
4          UserService.userControllerLogin,
```

```
        action.payload
      );
  } catch (error) {
    console.error(error);
    return;
  }
```

- **Zapisanie danych w storze**

  Jeśli serwer nie zwrócił błędu, to następnie w sadze dane z odpowiedzi (w tym przypadku token autoryzacyjny) zostają zapisane w Redux'owym storze (stanie aplikacji) poprzez Redux'ową akcję.

Listing 14: Zapisanie danych do stanu aplikacji

```
yield * put(sessionActions.setLoginInfo(true))
yield * put(
    sessionActions.setAuthToken(responseLogin.accessToken)
);
```

- **Wyświetlanie**

  W Reactowym komponencie zostaje użyty hook - **useSelector**, który wywołuje wcześniej stworzony selector, dzięki czemu mamy w komponencie dane do wyświetlenia, które wcześniej zostały pobrane z serwera.

Listing 15: Pobranie danych ze stora (stanu aplikacji) w komponencie

```
useSelector(sessionSelectors.loginInfo)
```

3.3.3 Konfiguracja Swaggera

- Import wygenerowanej przez backend dokumentacji w postaci pliku JSON.
- Wywołanie komendy dostarczonej przez bibliotekę openapi-typescript-codegen

```
"generate-swagger": "openapi --input src/swagger/swagger.json --output src/
    swagger/api"
```

- Komenda ta wygeneruje w wskazanym przez nas katalogu zbiór typów oraz funkcji do komunikacji z serwerem.

Listing 16: Wygenerowany kontroler do tworzenia kategorii notatek

```
public static foldersControllerCreate(
requestBody: CreateFolderDto,
): CancelablePromise<FolderModel> {
        return __request(OpenAPI, {
            method: 'POST',
            url: '/folders',
            body: requestBody,
            mediaType: 'application/json',
        });
    }
```

3.3.4 Stylowanie

Stylowanie komponentów odbywa się w 2 miejscach:

- wewnątrz komponentówc- ustalane style dla konkretnego elementu

Listing 17: Style wewnątrz komponentu

```
<Box sx={{
        position: "absolute",
        display: "flex",
        flexDirection: "column",
        justifyContent: "center",
        alignItems: "center",
        height: "60vh",
        top: "20vh",
        left: "50%",
        transform: "translate(-50%, 0)",
      }}
    >
```

- style współdzielone dla całej aplikacji umieszczone w folderze ”theme”

Listing 18: Nadpisane style komponentów z Material UI

```
  MuiIconButton: {
    styleOverrides: {
      root: {
        color: palette.text.primary,
        borderRadius: "7px",
      },
    },
  }
```

Listing 19: Paleta kolorów

```
export const getPalette = (textColor, primaryColor, secondaryColor) => {
  return createPalette({
    background: {
      default: "#2F303A",
    },
    primary: {
      main: primaryColor,
    },
    secondary: {
      main: secondaryColor,
    },
    success: {
      main: "#66C965",
    },
    error: {
      main: "#DB5930",
    },
    borderGrey: {
```

```
      main: muiColors.grey[300],
    },
    warning: {
      main: "#FFC34A",
    },
    text: {
      primary: textColor,
      secondary: "#545778",
    },
    grey: {
      "200": "#F0F0F4",
    },
  });
};
```

3.3.5 Widoki aplikacji

- **Strona startowa** - strona tytułowa z opisem oraz przyciskami do logowania / tworzenia konta. Po kliknięciu w jeden z przycisków przenosi użytkownika do kolejnego widoku, zmieniając path aplikacji.

- **Logowanie** - formularz z dwoma polami na adres email oraz hasło istniejącego konta. Po kliknięciu przycisku "Log In"zostaje wysłane zapytanie na serwer. Po otrzymaniu pozytywnej odpowiedzi, zostajemy przeniesieni na widok Strony startowej.

- **Tworzenie konta** - formularz z trzema polami na nazwę użytkownika, adres email oraz hasło nowego konta. Po kliknięciu przycisku "Sign Up"zostaje wysłane zapytanie na serwer z danymi nowego konta. Po otrzymaniu pozytywnej odpowiedzi, zostajemy przeniesieni na widok Strony startowej.

- **Strona domowa, strona z notatkami oraz strona z ustawieniami** - widoki dostępne po zalogowaniu. W każdym z nich mamy widoczny pasek z nazwami kategorii i przyciskiem dodania oraz modyfikacji każdej z nich oraz header na górze strony z informacją o obecnej lokalizacji oraz wykonywanej akcji. W stronie z notatkami w headerze są przyciski do usunięcia kategorii oraz dodania nowej notatki. Strony te różnią się od siebie głównym kontenerem.
  - Na stronie domowej jest to ... .
  - Na stronie z notatkami są to notatki dla wybranej kategorii lub widok edycji danej notatki po kliknięciu w przycisk edycji na notatce.
  - Na stronie z ustawieniami są ustawienia aplikacji, takie jak kolory oraz rozmiary.

Literatura

[1] Figma, Inc., *Figma, aplikacja internetowa do projektowania interfejsów. adr.:* https://www.figma.com/

[2] Braxmeier H., *Pixabay, internetowy bank zdjęć i filmów stockowych. adr.:* https://pixabay.com/

[3] Preston-Werner T., Wanstrath C., Hyett P.J., Chacon S., *GitHub, usługa hostingu repozytorium. adr.:*, https://github.com.

[4] Walke J., *React.js, Biblioteka JavaScript do budowania interfejsów użytkownika. adr.:*, https://reactjs.org/

[5] Microsoft, *Typescript, wolny i otwartoźródłowy język programowania. adr.:*, https://www.typescriptlang.org/

[6] Dahl R., *Node.js, otwartoźródłowe, wieloplatformowe środowisko uruchomieniowe JavaScript. adr.:*, https://nodejs.org/en/

[7] Myśliwiec K., *NestJS, framework Node.js do budowania aplikacji po stronie serwera. adr.:*, https://nestjs.com/

[8] https://swagger.io/

[9] Fauna Inc., *FaunaDb, rozwiązanie typu "serverless"dla bazy danych NoSQL. adr.:*, https://fauna.com/

[10] Truong K., *Serverless - czym jest i jak działa?. adr.:* https://bulldogjob.pl/readme/serverless-czym-jest-i-jak-dziala

[11] Anomaly Innovations, *Co to jest serverless?. adr.:* https://sst.dev/chapters/pl/what-is-serverless.html

[12] Amazon.com, *Amazon Web Services, platforma chmurowa. adr.:*, https://aws.amazon.com/

[13] Bartek Dybowski, 28 Lis 2016 *Podstawy Redux - zarządzanie stanem aplikacji ReactJS. adr.:* https://www.nafrontendzie.pl/podstawy-redux-zarzadzanie-stanem-reacts

[14] Yassine Elouafi, 2015, *Redux-Saga, biblioteka do zarządzania efektami ubocznymi aplikacji. adr.:* https://redux-saga.js.org/

[15] Yangshun Tay, 2022, *Flux, In-Depth Overview. adr.:* https://facebook.github.io/flux/docs/in-depth-overview/

[16] Mozilla Foundation, *Generator. adr.:* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global$_O$*bjects/Generator*