Dokumentacja techniczna aplikacji
Business Card

Wstęp

Aplikacja BusinessCard ma na celu digitalizację papierowych wizytówek oraz usprawnienie procesu ich tworzenia i udostępniania. Ponadto zapewnia możliwość przekazania dowolnej ilości wizytówek innym użytkownikom i poszerzania sieci kontaktów w każdej sytuacji biznesowej.

System aplikacji webowej składa się z trzech zasadniczych elementów:

- nierelacyjnej bazy danych MongoDB
- backendu aplikacji opartego o framework SpringBoot
- frontendu aplikacji opartego o framework Angular z wykorzystaniem NgRx

Wymiana danych pomiędzy aplikacją serwerową a kliencką odbywa się za pomocą architektury REST. Aplikacja rozszerzona jest o standard Progressive Web Application.

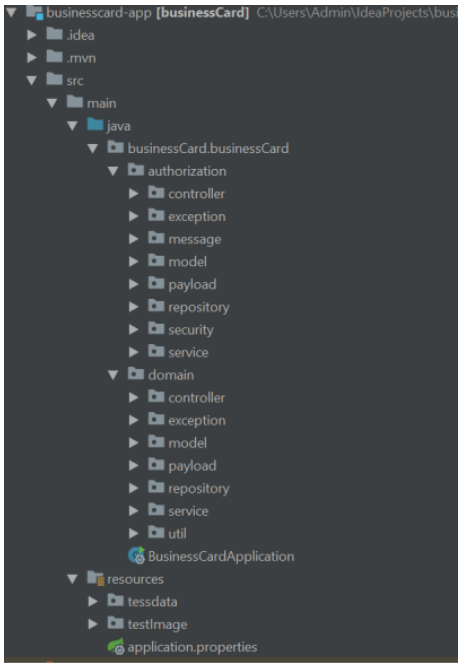
Spis treści

Rozdział 1

Backend

Serwer aplikacji oparty został o framwork Spring Boot i korzysta z zewnętrzej, nierelacyjnej bazy danych MongoDB.

1.1. Struktura aplikacji businesscard-app



Rysunek 1.1. Struktura aplikacji businesscard-app

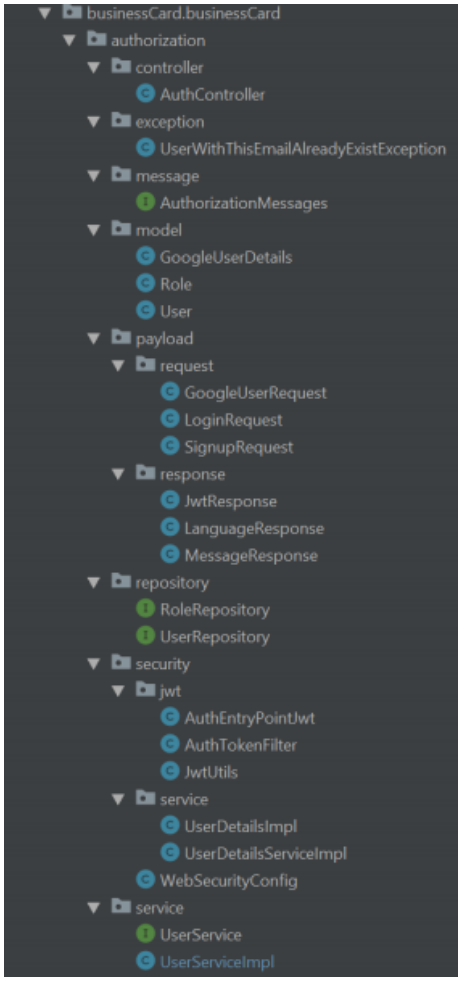
Aplikacja prowadzona jest według struktury Spring MVC. W głównym katalogu znajdują się dwa foldery:

- /authorization
- /domain

Ze względu na rozbudowanie serwisu autoryzacji funkcjonalność została wydzielona do osobnego katalogu.W folderze domain znajdują się wszystkie funkcjonalności związane z wizytówkami.

1.1.1. /authorization

Serwer autoryzacji wykorzystuje Spring Security w oparciu o tokeny JWT



Rysunek 1.2. Struktura folderu authorization

1.1.1.1. /controller

Plik AuthController odpowiedzialny jest za kontroler, zawarte są w nim metody

Tabela 1.1. AuthController

Metoda	Endpoint	Opis
POST	/api/auth/signin/local	logowanie do aplikacji za pomocą wbudowanego systemu autoryzacji
POST	/api/auth/signup	rejestracja do aplikacji za pomocą wbudowanego systemu autoryzacji
POST	/api/auth/signin/google	logowanie do aplikacji za pomocą platformy Google
POST	/api/auth/logout	wylogowanie z aplikacji

1.1.1.2. /model

Wszystkie informacje dotyczące autoryzacji przechowywane są w kolekcjach bazy danych User oraz Role, których klucze odwzorowane są w modelach obiektów User, Role.

```
@Data
@Builder
@Document(collection = "Roles")
public class Role {

    @Id
    private String id;

    private ERole name;

    public enum ERole {
        ROLE_USER,
        ROLE_ADMIN
    }
}
```

Rysunek 1.3. Obiektu Role

Obiekt Role składa się z:

- id (id)
- role (nazwy roli do wyboru: admin lub user)

Obiekt User przechowuje wszystkie potrzebne informacje o użytkowniku:

Rysunek 1.4. Obiektu User

- id
- username (nazwy użytkownika)
- email
- password (zahashowanego hasła)
- isConsentToSharePD (informacji, czy użytkownik wyraził zgodę na przetwarzanie danych osobowych)
- roles (ról uzytkownika)
- language (języka interfejsu)

1.1.1.3. /repository

Zgodnie z kolekcjami User, Role, moduł autoryzacji posiada dwa repozytoria: RoleRepository oraz UserRepository rozszerzające MongoRepository. Metody w

repozytoriach RoleRepository:

- findByName - znalezienie w kolekcji roli o zadanej nazwie

UserRepository:

- findByUserName - wyciągnięcie obiektu User z bazy danych według zadanej nazwy użytkownika
- exsistByUserName - zwrócenie informacji o istnieniu użytkownika wedgłu podanej nazwy użytkownika

- exsistByEmail - zwrócenie informacji o istnieniu użytkownika wedgłu nazwy email
- removeUserByUserName - usunięcie obiektu User z kolekcji User według zadanej nazwy użytkownika
- findByEmail - wyciągnięcie obiektu User z bazy danych według zadanej nazwy email

1.1.1.4. Folder /security

WebSecurityConfig - klasa konfiguracyjna rozszerzająca WebSecurityConfigureAdapter, który zawarty jest w module Spring Security Cofig.

W podfolderze /jwt znajdują się klasy związane z generowaniem tokenów:

- klasa AuthEntryPointJwt implementuje interfejs, zawarty w module Spring Security Core, AuthenticationEntryPoint
- klasa AuthTokenFilter odpowiedzialny jest za fitrowanie tokenów
- klasa JwtUtils posiada metody związane z generowaniem, walidowaniem, sprawdzaniem ważności tokenów

W podfolderze /service znajdują się klasy powiązane z serwisami z modułu Spring Security Core:

- klasa UserDetailsImpl rozszerza springową klasę UserDetails, nadpisując obiekt User danymi z obiektu User zawartego w aplikacji BusinessCard
- klasa UserDetailsServiceImpl implementuje springowy serwis UserDetailsService i nadpisuje metodę loadUserByUserName

1.1.1.5. Folder /payload

Folder podzielony został na podfoldery /request, /response, w których znajdują się modele obiektów wysyłanych w requestach oraz responsach endpointów zawartych w AuthController.

1.1.1.6. Folder /message

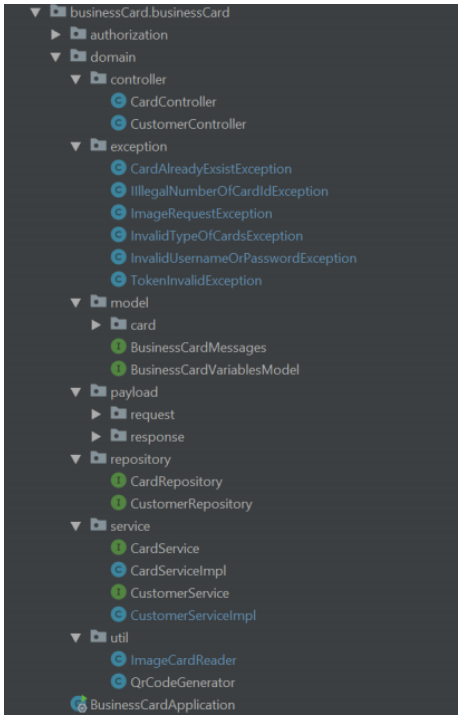
Zawiera interfejs AuthorizationMessages ze wszystkimi wiadomościami przedstawianymi użytkownikowi.

1.1.1.7. Folder /exception

Zawiera indywidualny exception UserWithThisEmailAlreadyExsistException.

1.1.2. /domain

Struktura



Rysunek 1.5. Folder domain

1.1.2.1. /controller

W folderze zawarte zostały dwa pliki kontrolera aplikacji:

1.1.2.2. /exception

W folderze zawarte zostały wszystkie customowe wyjątki powiązane z użytkownikiem i wizytówkami.

CardController - posiada endpointy związane z wizytówkami, dostępne dla zalogowanych i niezalowowanych użytkowników:

Tabela 1.2. CardController

Metoda	Endpoint	Opis
GET	/api/cards/all	zwraca wszystkie wizytówki zawarte w systemie
GET	/api/cards/id/id	zwraca wizytówkę po zadanym id
POST	/api/cards/search	zwraca wyzytówki według zadanych filtrów
POST	/api/cards/sort	zwraca posortowane wizytówki według nazwiska i imienia

CustomerController - posiada endpointy związane z użytkownikami oraz wizytówkami:

Tabela 1.3. CustomerController

Metoda	Endpoint	Opis
GET	/api/customers/all	zwraca listę wszystkich użytkowników
DELETE	/api/customers/remove/user	usuwa z bazy danych zalogowanego użytkownika
GET	/api/customers/cards/personal/all	zwraca wszystkie utworzone wizytówki zalogowanego użytkownika
GET	/api/customers/cards/wallet/all	zwraca wszystkie dodane do portfela wizytówki zalogowanego użytkownika
GET	/api/customers/cards/shared/all	zwraca wszystkie aktualnie udostępnione wizytówki dla zalogowanego użytkownika
POST	/api/customers/wallet/add	dodaje zadaną wizytówkę do portfela zalogowanego użytkownika
GET	/api/customers/wallet/card/note	zwraca notatkę zawartą w portfelu dla zadanej wizytówki
PUT	/api/customers/wallet/card/note	zapisuje przesłany tekst w notatce w portfelu dla zadanej wizytówki
GET	/api/customers/myCards/byId	zwraca wizytówkę utworzoną przez zalogowanego użytkownika według zadanego id
POST	/api/customers/myCards/add	dodaje utworzoną przez zalogowanego użytkownika wizytówkę do systemu
DELETE	/api/customers/card/remove	usuwa z portfela zalogowanego użytkownika wizytówkę wedgług zadanego id wizytówki
PUT	/api/customers/myCards/update	aktualizuje dane w wizytówce utworzonej przez zalogowanego użytkownika
POST	/api/customers/share	udostępnia wskazaną wizytówkę wskazanemu użytkownikowi
PUT	/api/customers/myCards/status/updateAll	aktualizuje statusy wszystkich wizytówek utworzonych przez zalogowanego użytkownika

Metoda	Endpoint	Opis
POST	/api/customers/read/image	wczytuje zdjęcie w formacie base64 i zwraca ciągi znaków zawarte na ilustracji
GET	/api/customers/settings	zwraca dane zawarte w ustawieniach zalogowanego użytkownika
POST	/api/customers/settings/languae	ustawia wskazany język jako język interfejsu zalogowanego użytkownika

1.1.2.3. /model

W folderze zostały zawarte modele obiektów Card, Customer, NoteWithCardId oraz SharedCard. Ponadto folder zawiera interfejsy skupiające wszystkie zmienne statyczne oraz wiadomości zawarte w innych klasach.

Rysunek 1.6. Card - zawiera wszystkie informacje o wizytówkach

Rysunek 1.7. Customer - zwiera wszystkie infomacje o użytkowniku i jego wizytówkach

1.1.2.4. /payload

Folder podzielony został na podfoldery /request, /response, w których znajdują się modele obiektów wysyłanych w requestach oraz responsach endpointów zawartego w CardController oraz CustomerController.

1.1.2.5. /repository

W folderze zawarte zostały repozytoria odpowiednie do kolekcji w bazie danych - CardRepository oraz CustomerRepository rozszerzające MongoRepository. Metody w repozytoriach:

CardRepository:

- findCardById - zwraca wizytówkę według zadanego id
- findByIsPrivateAndIsConsentToDisplay - zwraca wizytówki wg podanej informacji o prywatnosci i zgodzie na przetwarzanie danych osobowych użytkownika

CustomerRepository:

- findCustomerByUserName - zwraca wyszukanego w bazie danych użytkownika według zadanej nazwy użytkownika
- removeCustomerByUserName - usuwa wskazanego użytkownika z kolekcji Customer

1.1.2.6. /service

CardService - zawiera zbiór metod wykorzystywanych w endpoinach zawartych w kontrolerze CardController. Metody zawarte w CardService:

- findCardById - wyszukuje wizytówkę według zadanego id
- findAllCards - zwraca wszystkie wizytówki w systemie
- insertCard - dodaje wizytówkę do bazy danych
- saveCard - zapisuje wizytówkę w bazie danych
- removeCard - usuwa wizytówkę z bazy danych
- sortCardByLastname - sortuje zadane wizytówki według nazwiska
- findCards - wyszukuje wizytówki według zadanych filtrów

Metody zawarte w CustomerService:

- removeUser - usuwa użytkownika z kolekcji Customer oraz User
- findCustomerByUserName - zwraca użytkownika z kolekcji Customer wedgług zadanej nazwy użytkownika
- findAllCustomers - zwraca wszystkich użytkowników zawartych w kolekcji Customer
- addPersonalCard - dodaje wizytówkę utworzoną przez zalogowanego uzytkownika do systemu

- getCustomerallPersonalCards - zwraca wszystkie wizytówki utworzone przez zalogowanego użytkownika
- getCustomerAllCardsInWallet - zwraca wszystkie wizytówki dodane do portfela przez zalogowanego użytkownika
- getCustomerAllSharedCards - zwraca wszystkie udostępnione dla zalogowanego użytkownika wizytówki
- removeCardAndUpdateCardList - usuwa wizytówkę z portfela użytkownika
- addCardToWallet - dodaje wizytówkę do portfela użytkownika
- getPersonalCardById - zwraca wizytówkę utworzoną przez zalogowanego użytkownika weddług zadanego id
- updateCard - aktualizuje wskazaną wizytówkę zadanymi danymi
- getCustomerCardNote - zwraca informację o notatce wskazanej wizytówki dodanej do portfela
- updateCustomerCardNote - aktualizuje notatkę w zadanej wizytówce
- shareCard - udostępnia wskazaną wizytówkę wskazanemu użytkownikowi
- updateAllCardsStatus - aktualizuje status wszyskich utworzonych przez użytkownika wizytówek
- getCustomerSettingsData - zwraca informacje o danych zawartych w ustawieniach zalogowanego użytkownika
- updateUserLanguage - zmienia język interfejsu zalogowanego użytkownika
- readCardFromImage - rozpoznaje słowa we wskazanym zdjęciu

1.1.2.7. /util

W folderze zawarte została klasa: - QrCodeGenerator, w której znajduje się metoda związana z tworzeniem kodu QR dla wizytówek - generateQrCode, - ImageCardReader, w której znajduje się metoda związana z wczytywaniem danych ze zdjęć - readTextFromImage.

1.1.3. /resources

W pliku application.properties zawarte zostały informacje związane z połączeniem aplikacji z zewnętrzną bazą danych, autoryzacji z platformą Google oraz generowaniem tokenów.

1.1.3.1. /tessdata

W folderze znajdują się dane trenujące związane z funkcjonalnością OCR - Optical character recognition - zawartą w projekcie. Plik eng.taineddata zawiera dane trenujące powiązane z językiem angielskim, natomiast pol.traineddata z językiem polskim.

Rozdział 2

Baza danych

Aplikacja łączy się z nierelacyjną bazą danych MongoDB składającą się z kolekcji:

Rysunek 2.1. Struktura bazy danych

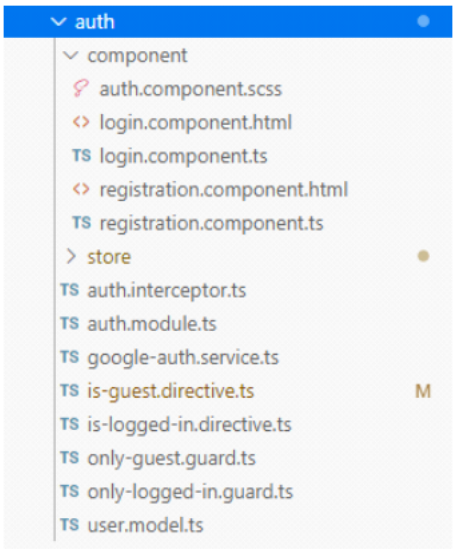
- Cards - posiada informacje o wizytówkach, analogicznie do obiektu Card,
- Customers posiada informacje o użytkowniku i jego wizytówkach, analogicznie do obiektu Customer,
- Users posiada informacje o zalogowanym użytkowniku i jego danych podczas autoryzacji, analogicznie do obiektu User,
- Roles posiada informacje o dostępnych rolach systemowych: admin, user, analogicznie do obiektu Role

Rozdział 3

Frontend

3.1. Moduł autoryzacji

Moduł autoryzacji zawiera wszystkie komponenty oraz mechanizmy dotyczące logowania i rejestracji użytkownika.



Rysunek 3.1. Folder authorization

3.1.0.1. Logowanie do systemu

Użytkownik ma możliwość zalogowania się poprzez podanie nazwy użytkownika i hasła lub za pomocą konta Google+.

```
authLogin$ = createEffect(() =>
  this.actions$.pipe(
    ofType(LoginStartAction.type),
    switchMap((authData: LoginStartActionData) => {
      return this.http.post<AuthResponseData>
      (`${environment.api}/auth/signin/local`,
        {
          username: authData.username,
          password: authData.password
        }
      ).pipe(
        map(resData => {
          return handleAuthentication(
            resData.email,
            resData.username,
            resData.language,
            resData.token,
            resData.expirationDate,
          );
        }),
        catchError(errorResponse => handleError(errorResponse))
      );
    })
  )
);

authGoogleLogin$ = createEffect(() =>
  this.actions$.pipe(
    ofType(SignInWithGoogleAction.type),
    switchMap((authData: SignInWithGoogleActionData) => {
      return this.http.post<AuthResponseData>
      (`${environment.api}/auth/signin/google`,
        {
          idGoogleToken: authData.idGoogleToken,
          expirationDate: authData.expirationDate
        }
      ).pipe(
        map(resData => {
          return handleAuthentication(
            resData.email,
            resData.username,
            resData.language,
            resData.token,
            resData.expirationDate
          );
        }),
        catchError(errorResponse => {
          return handleError(errorResponse);
        })
      );
    })
  )
`
```

Rysunek 3.2. Efekty odpowiadające za logowanie

3.1.0.2. Rejestracja do systemu

Użytkownik ma możliwość przejścia do strony rejestracji konta. Po podaniu danych osobowych oraz wyrażeniu zgody na ich przetwarzanie, użytkownik ma możliwość założenia konta w aplikacji.

```
authSignUp$ = createEffect(() =>
  this.actions$.pipe(
    ofType(SignUpStartAction.type),
    switchMap((signUpAction: SignUpStartActionData) => {
      return this.http.post<string>
      (`${environment.api}/auth/signup`,
        {
          email: signUpAction.email,
          password: signUpAction.password,
          username: signUpAction.username,
          language: signUpAction.language,
          isConsentToSharePD: signUpAction.isConsentToSharePD,
          firstName: signUpAction.firstName,
          lastName: signUpAction.lastName,
          roles: ['user']
        }
      ).pipe(
        map(() => {
          return LoginStartAction({
            username: signUpAction.username,
            password: signUpAction.password
          });
        }),
        catchError(errorResponse => handleError(errorResponse))
      );
    })
  )
);
```

Rysunek 3.3. Efekty odpowiadające za rejestrację

3.1.0.3. Uprawnienia

Użytkownik niezalogowany ma dostęp jedynie do strony logowania, rejestracji oraz zakładki search. Pozostałe strony są zabezpieczone guardem, który użytkownika niezalogowanego przekierowuje na stronę logowania. Z drugiej strony, użytkownik zalogowany nie ma możliwości wejścia na stronę logowania oraz rejestracji- odpowiada za to mechanizm OnlyLoggedInGuard. Podczas takiej próby użytkownik zostanie przekierowany do zakładki search.

```
@Injectable({
  providedIn: 'root',
})
export class OnlyGuestGuard implements CanActivate {
  constructor(private router: Router, private store: Store<AppState>) {
  }

  canActivate(): boolean | Promise<boolean | UrlTree> | Observable<boolean | UrlTree> | UrlTree {
    return this.store.select(selectAuthUser)
      .pipe(
        take(1),
        map(user => {
          if (!!user) {
            this.router.navigate(['/search']);
            return false;
          }
          return true;
        })
      );
  }
}
```

```
@Injectable({
  providedIn: 'root',
})
export class OnlyLoggedInGuard implements CanActivate {
  constructor(private router: Router, private store: Store<AppState>) {
  }

  canActivate(): boolean | Promise<boolean | UrlTree> | Observable<boolean | UrlTree> | UrlTree {
    return this.store.select(selectAuthUser)
      .pipe(
        take(1),
        map(user => {
          if (!!user) {
            const expirationDate: Date = new Date(Number(user.expirationDate));
            if (expirationDate > new Date()) {
              return !!user;
            }
          }
          this.router.navigate(['/auth/login']);
          return false;
        })
      );
  }
}
```

Rysunek 3.4. Guards

Istnieją też w aplikacji elementy, które sa pokazywane i ukrywane w zależności od tego, czy użytkownik jest zalogowany. Za obsługę tego typu akcji odpowiedzialne są dyrektywy is-guest.directive oraz is-logged-in-directive.

```
@Directive({
  selector: '[appIsGuest]',
})
export class IsGuestDirective {
  constructor(private elementRef: ElementRef, private store: Store<AppState>) {
    this.store
      .select(selectAuthUser)
      .pipe(take(1))
      .subscribe((user) => {
        if (!user) {
          this.elementRef.nativeElement.style.display = 'block';
        } else {
          this.elementRef.nativeElement.style.display = 'none';
        }
      });
  }
}
```

```
@Directive({
  selector: '[appIsLoggedIn]'
})
export class IsLoggedInDirective {

  constructor(
    private elementRef: ElementRef,
    private store: Store<AppState>
  ) {
    this.store.select(selectAuthUser)
      .pipe(take(1))
      .subscribe(user => {
        if (!user) {
          this.elementRef.nativeElement.style.display = 'none';
        } else {
          this.elementRef.nativeElement.style.display = 'block';
        }
      });
  }
}
```

Rysunek 3.5. Directives

3.1.0.4. Autoryzacja z tokenem Bearer

Każdy request, który zostaje wyemitowany w celu komunikacji z backendem, jest przechwytywany przez auth-interceptor. Jeśli istnieje w globalnym store aplikacji obiekt użytkownika, tzn istnieje zalogowany obecnie użytkownik,

do requesta dodawany jest nagłówek autoryzacji z tokenem typu Bearer. W przeciwnym wypadku, request pozostaje bez zmian.

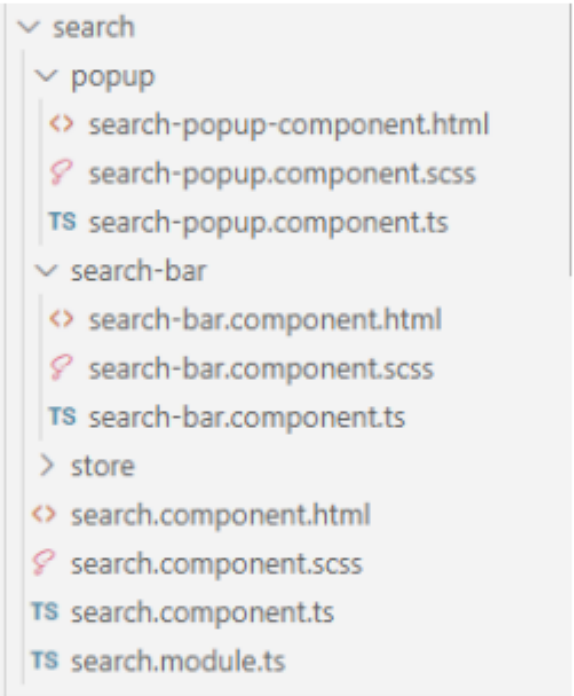
```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private store: Store<AppState>) {
  }

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return this.store.select(selectAuthUser)
      .pipe(take(1),
        exhaustMap(user => {
          if (!user) {
            return next.handle(req);
          }
          const modifiedRequest = req.clone({
            headers: new HttpHeaders().set('Authorization', 'Bearer ' + user.token)
          });
          return next.handle(modifiedRequest);
        })
      );
  }
}
```

Rysunek 3.6. Interceptor

3.2. Moduł wyszukiwarki

Moduł wyszukiwarki - umożliwia dostęp do listy wszystkich publicznych wizytówek w systemie, to znaczy takich, których właściciele wyrazili zgodę na udostępnienie w wyszukiwarce.



Rysunek 3.7. Struktura modułu wyszukiwarki

3.2.1. Pobieranie wizytówek

Komponent pobiera odpowiednią stronę listy wszystkich publicznych wizytówek dostępnych w systemie.

```
getCards$ = createEffect(() => this.actions$.pipe(
  ofType(FetchSharedCardsAction.type),
  switchMap(() => {
    return this.http
      .get<SharedCardModel[]>(`${environment.api}/customers/cards/shared/all`)
      .pipe(
        map(cards => SetSharedCardsAction({sharedCards: cards})),
        catchError(err => handleError(err))
      );
  })
));
```

Rysunek 3.8. Efekt odpowiedzialny za pobieranie wizytówek

3.2.2. Filtrowanie wizytówek

W wyszukiwarce możemy filtrować listę według podanych kryteriów. Po wpisaniu warunków wyszukiwania aplikacja przekazuje parametry do backendu, a następnie zawraca przefiltrowaną listę wizytówek.

```
searchCard$ = createEffect(() =>
  this.actions$.pipe(
    ofType(SearchCardAction.type),
    switchMap((searchData: SearchCardActionData) => {
      return this.http
        .post<any>(
          `${environment.api}/cards/search`,
          this.createBody(searchData)
        )
        .pipe(
          map((pagination) => {
            return SetSearchCardsAction({
              cards: pagination.content,
              isFirstPage: pagination.first,
              isLastPage: pagination.last,
              currentPage: pagination.number,
            });
          }),
          catchError((err) => handleError(err))
        );
    })
  )
);
```

Rysunek 3.9. Efekt odpowiedzialny za filtrowanie wizytówek

3.2.3. Sortowanie wizytówek

Podobnie po wybraniu kierunku sortowania aplikacja komunikuje się z backendem, który zwraca posortowaną listę wizytówek.

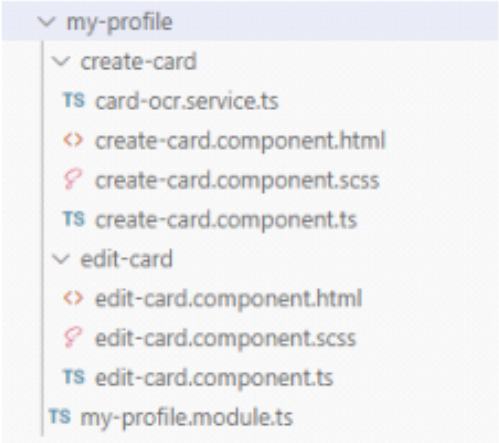
```
sortCard$ = createEffect(() =>
  this.actions$.pipe(
    ofType(SortSearchCardsAction.type),
    map((searchData: SortSearchCardsActionData) => searchData),
    withLatestFrom(this.store$.select(selectSearchCards)),
    switchMap(([searchData, searchCards]) => {
      return this.http
        .post<any>(
          `${environment.api}/cards/sort?ascending=${searchData.order}`,
          searchCards,
          { params: { page: searchData.page.toString() } }
        )
        .pipe(
          map((pagination) => {
            return SetSearchCardsAction({
              cards: pagination.content,
              isFirstPage: pagination.first,
              isLastPage: pagination.last,
              currentPage: pagination.number,
            });
          }),
          catchError((err) => handleError(err))
        );
    })
  )
);
```

Rysunek 3.10. Efekt odpowiedzialny za sortowanie wizytówek

3.3. Moduł portfela

3.3.1. Kreator wizytówki

Moduł kreatora wizytówki - zawiera komponenty związane z tworzeniem i edycją wizytówek.



Rysunek 3.11. Struktura kreatora wizytówki

Komponent tworzenia wizytówek udostępnia reaktywny formularz, mający na celu wprowadzenie i walidację danych, a następnie przekazanie ich do endpointu tworzenia wizytówki.

```
addOwnCard$ = createEffect(() => this.actions$.pipe(
  ofType(AddOwnCardAction.type),
  switchMap((actionData: AddOwnCardActionData) => {
    return this.http.post(`${environment.api}/customers/myCards/add`, actionData.card)
      .pipe(
        map(() => {
          this.router.navigate(['/wallet']);
          return FetchOwnCardsAction();
        }),
        catchError(err => handleError(err))
      );
  })
));
```

Rysunek 3.12. Efekt odpowiadający za dodanie wizytówki

Komponent tworzenia wizytówki daje możliwość wczytania zdjęcia wizytówki z urządzenia i przesyła je do backendu. W odpowiedzi otrzymujemy listę wczytanych ze obrazu wyrazów, które użytkownik może poprzeciągać na odpowiednie miejsca wizytówki, za pomocą mechanizmu drag and drop.

3.3.2. Edycja wizytówki

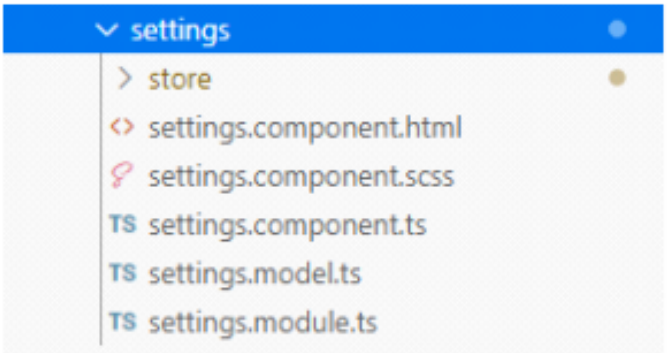
Komponent edycji wizytówki daje możliwość zmiany danych wprowadzonych podczas tworzenia wizytówki.

```
updateOwnCard$ = createEffect(() =>
  this.actions$.pipe(
    ofType(UpdateOwnCardAction.type),
    switchMap((actionData: UpdateOwnCardActionData) => {
      return this.http
        .put(
          `${environment.api}/customers/myCards/update?cardId=${actionData.card.id}`,
          actionData.card
        )
        .pipe(
          map(() => {
            this.router.navigate(['/wallet']);
            return FetchOwnCardsAction();
          }),
          catchError((err) => handleError(err))
        );
    })
  )
);
```

Rysunek 3.13. Efekt odpowiadający za edytowanie wizytówki

3.4. Moduł ustawień

Moduł ustawień - zawiera komponent ustawień konta użytkownika.



Rysunek 3.14. Struktura ustawień

Podczas inicjalizacji komponentu z backendu zostają pobrane informacje dotyczące danych osobowych zalogowanego uzytkownika, aktualny język interfejsu oraz statystyki dotyczące dokonań w aplikacji.

```
@Injectable()
export class SettingsEffects {
  getSettings$ = createEffect(() => this.actions$.pipe(
    ofType(GetSettingsAction.type),
    switchMap(() => {
      return this.http
        .get<SettingsModel>(`${environment.api}/customers/settings`)
        .pipe(
          map(data => SetSettingsAction({settings: data})),
          catchError(err => handleError(err))
        );
    })
  ));
```

Rysunek 3.15. Efekt odpowiadający za pobranie danych z ustawień

3.4.1. Język interfejsu

Użytkownik ma możliwość zmiany języka interfejsu.

```
updateLanguage$ = createEffect(() =>
  this.actions$.pipe(
    ofType(UpdateLanguageAction.type),
    switchMap((updateLanguage: UpdateLanguageActionData) => {
      return this.http
        .post<string>(`${environment.api}/customers/settings/language`, { language: updateLanguage.language })
        .pipe(
          map(data => SetLanguageAction({language: data['language']})),
          catchError(err => handleError(err))
        )
    })
  )
);
```

Rysunek 3.16. Efekt odpowiadający za zmianę języka

3.4.2. Usunięcie konta

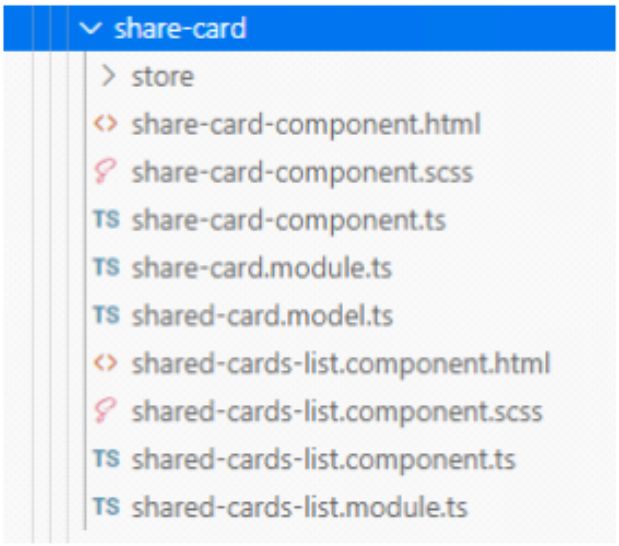
Użytkownik ma możliwość trwałego usunięcia konta.

```
deleteAccount$ = createEffect(() =>
  this.actions$.pipe(
    ofType(DeleteAccountAction.type),
    switchMap(() => {
      return this.http.delete(`${environment.api}/customers/remove/user`)
        .pipe(
          map(() => {
            localStorage.removeItem(LOCAL_STORAGE_USER_EMAIL_KEY);
            localStorage.removeItem(LOCAL_STORAGE_USER_NAME_KEY);
            localStorage.removeItem(LOCAL_STORAGE_USER_TOKEN_KEY);
            localStorage.removeItem(LOCAL_STORAGE_USER_EXPIRATION_KEY);
            this.router.navigate(['/auth/login']);
            return ClearUserAction();
          }),
          catchError(err => handleError(err)));
    }))
);
```

Rysunek 3.17. Efekt odpowiadający za usunięcie konta

3.5. Moduł udostępnionych

Moduł udostępnionych wizytówek - zawiera listę wizytówek udostępnionych danemu użytkownikowi oraz komponent dialogu umożliwiającego udostępnienie wizytówki użytkownikowi poprzez wpisanie jego nazwy.



Rysunek 3.18. Struktura modułu udostępnionych

```
getCards$ = createEffect(() => this.actions$.pipe(
  ofType(FetchSharedCardsAction.type),
  switchMap(() => {
    return this.http
      .get<SharedCardModel[]>(`${environment.api}/customers/cards/shared/all`)
      .pipe(
        map(cards => SetSharedCardsAction({sharedCards: cards})),
        catchError(err => handleError(err))
      );
  })
));
```

Rysunek 3.19. Komponent pobiera listę udostępnionych wizytówek

```
getCards$ = createEffect(() => this.actions$.pipe(
  ofType(FetchSharedCardsAction.type),
  switchMap(() => {
    return this.http
      .get<SharedCardModel[]>(`${environment.api}/customers/cards/shared/all`)
      .pipe(
        map(cards => SetSharedCardsAction({sharedCards: cards})),
        catchError(err => handleError(err))
      );
  })
));
```

Rysunek 3.20. Komponent pobiera listę udostępnionych wizytówek

```
addCard$ = createEffect(() =>
  this.actions$.pipe(
    ofType(AddSharedCardToWalletAction.type),
    switchMap((cardData: AddSharedCardActionData) => {
      return this.http
        .post(`${environment.api}/customers/wallet/add`, cardData)
        .pipe(
          map(() => FetchSharedCardsAction()),
          catchError((err) => handleError(err))
        );
    })
  )
);

deleteCard$ = createEffect(() =>
  this.actions$.pipe(
    ofType(DeleteSharedCardAction.type),
    switchMap((cardData: DeleteSharedCardActionData) => {
      return this.http
        .delete(
          `${environment.api}/customers/card/remove?cardId=${cardData.cardId}&type=${CardTypeEnum.SharedCard}`
        )
        .pipe(
          map(() => FetchSharedCardsAction()),
          catchError((err) => handleError(err))
        );
    })
  )
);
```

Rysunek 3.21. Każdą udostępnioną wizytówkę możemy zaakceptować, co skutkuje dodaniem jej do portfela lub odrzucić, czyli usunąć listy udostępnionych bez dalszych akcji.

```
shareCard$ = createEffect(() => this.actions$.pipe(
  ofType(ShareCardAction.type),
  switchMap((cardData: ShareCardActionData) => {
    return this.http.post<void>(`${environment.api}/customers/share`, cardData)
      .pipe(
        map(() => ShareCardSuccessAction()),
        catchError(err => handleError(err))
      );
  })
));
```

Rysunek 3.22. Komponent dialogu zapewnia możliwość udostępnienia wizytówki innemu uzytkownikowi poprzez podanie jego nazwy.