**6.Opis aplikacji**

Zdefiniowanie wymagań funkcjonalnych oraz diagramów opisujących architekturę systemu, pozwala na rozpoczęcie implementacji aplikacji. Tworzenie kodu opierającego się o wcześniej stworzony projekt, pozwala na uniknięcie dużej liczby błędów oraz stworzenie optymalnego rozwiązania. Spójna wizja jest bardzo pomocna w pisaniu kodu czytelnego oraz pozwalającego na przyszłe rozszerzenie funkcjonalności, dzięki wzorcom projektowym, zastosowanym w implementacji. By aplikacja mogła dobrze działać, należy przygotować środowisko wykonawcze, łatwe do zarządzania oraz działające w jak najmniejszej liczbie konfiguracji, potrzebnej do uruchomienia danego systemu. Bardzo ważnym punktem, z perspektywy użytkownika, jest nie tylko dobrze działająca aplikacja, ale interfejs, pozwalający na intuicyjne korzystanie z funkcjonalności. W tym celu, w projekcie aplikacji, został stworzony prototyp interfejsu, na podstawie którego, zaimplementowane zostały widoki w warstwie klienckiej. Ostatnim elementem opisanym w tym rozdziale są testy zdefiniowane i przeprowadzone w ramach sprawdzania działania systemu. Istnieją różne rodzaje testów, które są istotnym czynnikiem wpływającym na kontrolę nad błędami pojawiającymi się w trakcie działania. Testy jednostkowe z łatwością mogą pokazać miejsce, w którym zmiana mogła spowodować błąd, natomiast testy akceptacyjne, przeprowadzone w formie testów interfejsowych, sprawdzają główne funkcjonalności systemu. Rozdział ten opisuje wszystkie, wyżej wymienione fazy tworzenia aplikacji.

**6.1. Środowisko wykonawcze**

Biorąc pod uwagę, że aplikacja tworzona w ramach tej pracy, jest to aplikacja webowa, jej środowisko wykonawcze można podzielić na dwie części – środowisko wykonawcze po stronie klienta oraz serwera.

Środowiskiem wykonawczym po stronie klienta dla aplikacji webowej, jest dowolna przeglądarka internetowa, posiadająca silnik, uruchamiający kod pisany w Javascripcie. Najpopularniejszymi przeglądarkami są Mozilla, Chrome, Internet Explorer oraz Safari. Ważnym aspektem dobrze działającej aplikacji webowej, jest możliwość uruchomienia jej na każdej z tych przeglądarek. Różnice jakie można napotkać w interpretowaniu kodu przeznaczonego dla warstwy klienckiej, mogą generować problemy w napisaniu rozwiązania działającego na wszystkich dostępnych interpreterach. Jedną z zalet użycia frameworka Javascriptowego jest zapewnienie, że problem ten nie wystąpi dla tworzonej aplikacji. Angular, który został użyty przy implementacji, zapewnia spójny wygląd oraz prawidłowe działanie funkcjonalności na każdej z wyżej wymienionych przeglądarek.

Środowisko wykonawcze, którego wybranie jest zadaniem dla osoby tworzącej aplikację, jest to środowisko po stronie serwera. Biorąc pod uwagę silne połączenie serwerów z wybraną technologią, w jakiej napisana jest warstwa serwerowa, wybór narzędzia jest dokonywany wcześniej. Implementacja serwera używając platformy .NET, determinuje użycie serwera IIS w celu hostowania tworzonej aplikacji webowej.

IIS jest serwerem webowym stworzonym przez firmę Microsoft, działający na systemach Windows, akceptującym zdalne żądania klientów, wysyłając do nich odpowiednią treść odpowiedzi. Odpowiedzią na żądanie klienta może być statyczna strona HTML, tekst, obrazki oraz inne zasoby dostępne w internecie. Jednakże IIS dostarcza mechanizmy nie tylko generujące odpowiednie odpowiedzi, ale także zapewniające bezpieczeństwo oraz optymalne działanie serwera. Biorąc pod uwagę fakt, że każde żądanie jest obsłużone przez IIS, administrator jest w stanie sprawdzić tożsamość klienta oraz jego uprawnienia do zasobów, wymienionych w żądaniu. W celu ułatwienia zarządzania serwerem, wprowadzone zostały pojęcia puli aplikacji oraz procesu wykonawczego. Proces wykonawczy odpowiada za wygenerowanie wszystkich żądań i odpowiedzi pochodzących z serwera, będąc głównym punktem działania aplikacji serwowanej przez IIS. Pula aplikacji jest to zbiór procesów wykonawczych, pozwalających na organizację oraz łatwiejsze przypisanie uprawnień dla danego systemu działającego w ramach IIS’a.

Ostatnim elementem, wchodzącym w skład środowiska wykonawczego, jest Docker. Docker jest to open-sourcowy projekt, stworzony w 2014 roku. Pozwala on na definicję środowiska uruchomieniowego dla aplikacji, jako przenośnych kontenerów, posiadających całą konfigurację zdefiniowaną wewnątrz. Kontenery stworzone za pomocą Dockera, uruchomione na różnych maszynach, działają w ten sam sposób. W aplikacji tworzonej w ramach tej pracy, Docker został użyty do uruchomienia serwera bazodanowego oraz stworzenia architektury Lambda za pomocą tego serwera oraz Sparka. Dzięki takiemu podejściu, możliwe jest użycie stworzonej architektury bazodanowej dla różnych aplikacji oraz na różnych maszynach, bez wymogu zbędnej konfiguracji.

**6.2. Najważniejsze rozwiązane problemy**

Tworząc aplikacje webową, deweloper napotyka się na wiele problemów związanych z implementacją. Globalnym problemem, bardzo ważnym z perspektywy czytelności oraz przyszłych modyfikacji, są zasady, jakimi deweloper kieruje się przy tworzeniu swojego kodu. Pisząc aplikację tworzoną w ramach tej pracy, kierowano się zasadami SOLID oraz CQRS.

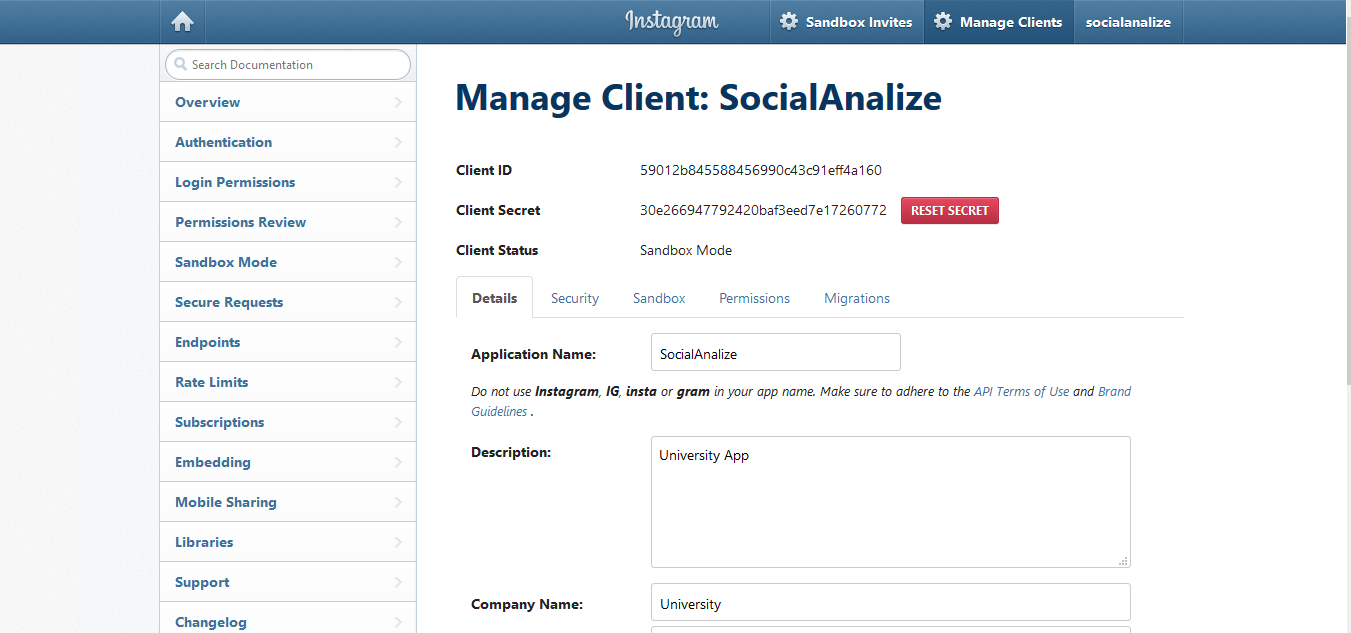
SOLID, oznacza 5 zasad, określonych przez Roberta C. Martina, którymi osoba wytwarzająca oprogramowanie powinna się kierować. Brzmią one następująco [17]:

* Zasada jednej odpowiedzialności – klasa powinna zajmować się jedną rzeczą np. aktualizacją bazy danych. Prowadzi to do podziału złożonych funkcji na funkcje wykonującą atomowe, pojedyncze działania
* Zasada otwarte – zamknięte – klasy powinny być otwarte na rozszerzenia lecz zamknięte na modyfikację. Prowadzi do użycia interfejsów i klas abstrakcyjnych w jak największej liczbie przypadków
* Zasada podstawienia Liskova – obiekty klas powinny być zastąpione instancjami klas pochodnych, bez generowania błędów w aplikacji
* Zasada segregacji interfejsów – interfejsy powinny dotyczyć konkretnej dziedziny. Nie powinno się tworzyć interfejsów posiadających opis różnych dziedzin
* Zasada odwrócenia zależności – klasa powinna bazować na klasach abstrakcyjnych, interfejsach, nie na konkretnych implementacjach

Drugim użytym wzorcem, jest CQRS (Command Query Responsibility Segregation). Wzorzec ten pozwala na wykorzystanie innego modelu danych do pobierania danych oraz ich aktualizacji. Podstawowym podejściem używanym przy tworzeniu modelów, służących do pracy z danymi, jest podejście CRUD’owe. Polega ono na tworzeniu wspólnych modeli dla wszytkich operacji oraz za ich pomocą oraz pomocą repozytoriów, na komunikacji ze źródłami danych. Dla prostych scenariuszy jest to podejście prawidłowe, jednakże gdy wymagania względem aplikacji są większe, kod tworzony w ten sposób może stać się nieczytelny. W celu uniknięcia takiej sytuacji, stworzony został wzorzec CQRS, definiujący modele związane z zapytaniami oraz komendami. Modele związane z zapytaniami służą jedynie do odczytania danych z bazy, natomiast w celu aktualizacji lub dodania nowych danych używa się komend. W sytuacji gdy, po stronie serwera jest duża ilość logiki związanej ze źródłem danych, wzorzec CQRS bardzo dobrze sprawdza się w celu utworzenia czytelnego i zrozumiałego kodu.

Przy tworzeniu aplikacji w ramach tej pracy, pierwszym, konkretnym, problemem była autentykacja użytkownika, odbywająca się za pomocą jednego z trzech, analizowanych portali społecznościowych. Dla każdego z nich, proces autentykacji wygląda w ten sam sposób i składa się z części rejestrującej aplikację na stronie deweloperskiej, związanej z portalem oraz części, w której aplikacja musi wykonać trzy żądania w celu uzyskania tokenu, pozwalającego wykonywać żądania do serwera.

Pierwszym etapem tworzenia mechanizmu autentykacji, za pomocą zewnętrznej aplikacji społecznościowej, jest utworzenie konta, a następnie klienta reprezentującego aplikacje na deweloperskiej stronie dla danego portalu społecznościowego.



Rys.6.1. Strona deweloperska Instagrama

Rysunek 6.1 przedstawia klienta, utworzonego w ramach tej pracy. Posiada on dane, takie ja client id oraz client secret, będące kluczami, potrzebnymi w później wysłanych żądaniach, mających na celu uzyskania tokenu dostępu dla użytkownika. Następnym elementem, jaki należy zdefiniować, jest adres powrotny do budowane w ramach niniejszej pracy aplikacji, po potwierdzeniu uprawnień przez użytkownika.

Posiadając informacje ze strony deweloperskiej, można przejść do implementacji autentykacji w budowanym systemie. Najpierw jednak należy zdefiniować słuchacza zdarzeń po stronie klienta, wywołującego akcje autentykacji po stronie serwera. Zawartość wywołanej funkcji przedstawiono na Listingu 6.1:

loginInstagramUser() {

window.location.href = 'http://localhost:50985/api/instagramAuthentication/authenticate';

}

Listing.6.1.Funkcja logująca użytkownika

Warstwą serwerową aplikacji, tworzonej w ramach tej pracy, jest to projekt Web API, więc wywołana funkcja przekierowuje użytkownika na konkretną akcję kontrolera po stronie serwera. Listing 6.2 pokazuje akcje kontrolera po stronie serwera, obsługującą autentykację.

[HttpGet]

[Route("authenticate")]

public ActionResult Get()

{

var clientId = "59012b845588456990c43c91eff4a160";

var clientSecretId = "30e266947792420baf3eed7e17260772";

var redirectUri = @"http://localhost:50985/api/instagramAuthentication";

var config = new InstagramConfig(clientId, clientSecretId, redirectUri, realtimeUri);

var scopes = new List<OAuth.Scope>();

scopes.Add(OAuth.Scope.Likes);

scopes.Add(OAuth.Scope.Comments);

scopes.Add(OAuth.Scope.Basic);

scopes.Add(OAuth.Scope.Follower\_List);

scopes.Add(OAuth.Scope.Relationships);

scopes.Add(OAuth.Scope.Public\_Content);

var link = OAuth.AuthLink(config.OAuthUri + "authorize", config.ClientId, config.RedirectUri, scopes, InstaSharp.OAuth.ResponseType.Code);

return Redirect(link);

}

[HttpGet]

public ActionResult Get(string code)

{

var accessToken = queryBus.Send<GetInstagramAccessToken, string>(new GetInstagramAccessToken() {

ClientId = "59012b845588456990c43c91eff4a160",

ClientSecretId = "30e266947792420baf3eed7e17260772",

Code = code,

ReturnUrl = @"http://localhost:50985/api/instagramAuthentication"

}).Result;

return Redirect("http://localhost:50985?accessToken=" + accessToken);

}

Listing.6.2.Funkcja obsługująca autentykację po stronie serwera

Pierwsza funkcja odpowiada za stworzenie przekierowania na poprawną stronę logowania portalu społecznościowego. Url stworzony w ramach tej funkcji posiada, klucze pobrane ze strony deweloperskiej oraz zdefiniowany tam adres powrotu. Druga funkcja jest wywołana przez portal społecznościowy za pomocą adresu powrotu, dodając do niego parametr zawierający kod, który służy do wysłania żądania HTTP w celu uzyskania tokenu. W ramach wzorca CQRS, wywoływany jest handler, wykonujący żądanie po token dostępu, a następnie przekierowuje użytkownika na stronę aplikacji frontowej, z tokenem jako parametrem adresu.

Przedstawiony na Listing 6.3 kod, tworzący parametry zapytania HTTP typu POST w celu uzyskania tokenu dostępu. Dzięki temu kluczowi, aplikacja w ramach tego użytkownika, jest w stanie pobrać dane z serwerów danego portalu społecznościowego.

private async Task<string> GetAccessToken(GetInstagramAccessToken authData)

{

var client = new HttpClient();

var postValues = new List<KeyValuePair<string, string>>

new KeyValuePair<string, string>("client\_id",authData.ClientId),

new KeyValuePair<string, string>("client\_secret",authData.ClientSecretId),

new KeyValuePair<string, string>("grant\_type","authorization\_code"),

new KeyValuePair<string, string>("redirect\_uri",authData.ReturnUrl),

new KeyValuePair<string, string>("code", authData.Code)

};

// now encode the values

var content = new FormUrlEncodedContent(postValues);

var authLinkUri = new Uri(@"https://api.instagram.com/oauth/access\_token");

// make request for auth token

var response = await client.PostAsync(authLinkUri, content);

var parsedResponse = await response.Content.ReadAsStringAsync();

var json = JObject.Parse(parsedResponse);

var accessToken = json["access\_token"].ToString();

return accessToken;

}

Listing.6.3.Kod uzyskujący Access Token dla użytkownika

Głównym problemem, w ramach aplikacji tworzonej w tej pracy, było zaimplementowanie mechanizmu Business Inteligence - Architektury Lambda. Biorąc pod uwagę to, że warstwą Speed tej architektury są to widoki tworzone w PostgreSQL, w tym rozdziale poniżej (Listing 6.4) przedstawiony jest prosty przykład kodu w języku Scala, wykonywany przez Sparka, w celu przetworzenia danych. Mechanizm ten, w zaimplementowanej architekturze, służy jako warstwa Batch, która zajmuje się przetworzeniem danych oraz utworzeniem gotowych do odczytania raportów na temat osoby obserwowanej.

def transformLocations(spark: SparkSession, df: DataFrame): DataFrame = {

import spark.implicits.\_

val jsons = df

.map(t => t.getString(2))

val locations = spark.read.json(jsons)

.select(explode($"data").alias("data"))

.select(

monotonically\_increasing\_id().as("Id"),

$"data.user.id".cast("long").as("UserId"),

$"data.user.username".as("UserName"),

from\_unixtime($"data.created\_time").cast("timestamp").as("Created"),

$"data.location.latitude".cast("decimal(9,6)").as("Latitude"),

$"data.location.longitude".cast("decimal(9,6)").as("Longitude")

)

locations.show()

return locations

}

Listing.6.4.Kod w Scali przetwarzający dane

Kod definiuje prostą transformację danych lokalizacyjnych, za pomocą Scali, wykonywaną przez Sparka. Wyciągnięte dane są następnie zapisane do tabeli, którą aplikacja wykorzystuje do budowy raportu związanego z lokalizacją. Kod pobierający surowe dane, wywołujący metodę „transformLocations” oraz zapisujący wynik tej metody do bazy jest pokazany na Listingu 6.5:

def processLocations(spark: SparkSession) {

val df = Postgres.read(spark, "import.instagram\_media\_recent")

val locations = transformLocations(spark, df);

Postgres.write(locations, "data.UserLocation")

}

Listing.6.5.Kod zapisujący przetworzone dane do bazy danych

Patrząc z perspektywy problemów występujących przy implementacji, można je podzielić na problemy globalne, związane z czytelnością oraz dobrym poukładaniem kodu oraz na konkretne problemy połączone z funkcjonalnościami. Na pierwszy typ problemów, rozwiązaniem jest podążanie za dobrymi praktykami kodowania, natomiast drugi typ problemów, często wymaga zaimplementowania specyficznego mechanizmu, pozwalającego na optymalne spełnienie danego wymagania funkcjonalnego.

**6.3. Testy jednostkowe, akceptacyjne oraz środowiskowe**

Dobrze działająca aplikacja, pozwalająca na łatwe rozszerzenie funkcjonalności, musi być aplikacją dobrze przetestowaną. Istnieje wiele rodzajów testów, sprawdzających różne etapy wytwarzania aplikacji. Najczęściej spotykanymi typami testów, są testy jednostkowe. Dobrze napisany test jednostkowy, odpowiada za sprawdzenie tylko jednej rzeczy. Jeżeli testuje zbyt obszerną część funkcjonalności, powoduje to duże trudności w utrzymaniu oraz słabą czytelność przy analizie raportu z przeprowadzonych testów. Drugim typem testów, występujących w aplikacji tworzonej w ramach tej pracy, są to testy akceptacyjne. Sprawdzają one działanie aplikacji w perspektywie wymagań funkcjonalnych, nie wchodząc w detale techniczne. W przypadku tego systemu, testami akceptacyjnymi będą to testy „end-to-end”, czyli testy Selenium.

Selenium jest to framework, pozwalający na komunikację z przeglądarką. Dzięki niemu, programista jest w stanie automatycznie sterować tym, co dzieje się na stronie internetowej. Istnieją dwa sposoby tworzenia testów Selenium – poprzez nagrywanie oraz pisanie kodu. Nagrywanie testu, polega na wyklikaniu tego, co dany test ma robić. W tym sposobie framework Selenium automatycznie generuje kod testu. Drugim sposobem, jest manualne napisanie testu przez dewelopera. Ostatnim typem testów, przeprowadzanych w końcowym okresie wytwarzania aplikacji, są to testy środowiskowe. Polegają one na udostępnieniu systemu dla kilku klientów, w celu sprawdzenia przez nich funkcjonalności oraz poprawności stworzonego interfejsu. W aplikacji tworzonej w ramach tej pracy, do stworzenia testów, zostaną użyte narzędzia związane z Angularem – do testów jednostkowych Jasmina wraz z Karmą oraz do testów akceptacyjnych Protractor.

Testy jednostkowe, w aplikacji pisanej w ramach tej pracy, zostaną stworzone po stronie klienta, za pomocą frameworka javscriptowego o nazwie Jasmine wraz ze środowiskiem testowym o nazwie Karma. Za pomocą tych narzędzi, deweloper jest w stanie stworzyć zestaw testów, sprawdzających działanie całej warstwy klienta, a następnie za pomocą Karmy przedstawić raport, pokazujący skuteczność tych testów za pomocą wskaźnika pokrycia kodu. W poniższych kodach źródłowych zostaną przedstawione przykładowe testy napisane za pomocą Jasmine.

beforeEach(async(() => {

TestBed.configureTestingModule({

providers: [

AuthenticationService,

{

provide: window,

useClass: MockWindow

}

]

})

.compileComponents();

}));

Pierwszym krokiem w napisaniu testów, jest zdefiniowanie operacji, które zostaną wykonane przed każdym testem, przygotowując zamienniki dla klas, odwołujących się do zewnętrznych serwisów i stron. W tym przypadku, został stworzony zamiennik dla zmiennej globalnej window. Biorąc pod uwagę, że w serwisie używamy tyko zmiennej href, zamiennik wygląda następująco:

export class MockWindow {

public location: {

href: string;

};

}

W tym momencie, deweloper jest przygotowany do zdefiniowania testów, sprawdzających serwis logowania.

it('should create', () => {

expect(service).toBeTruthy();

});

it('should redirect to proper page when instagram login', () => {

service.loginInstagramUser();

expect(window.location.href)

.toEqual('http://localhost:50985/api/instagramAuthentication/authenticate');

});

Powyżej zostały przedstawione dwa przykładowe testy, sprawdzające serwis logowania. Pierwszy sprawdza, czy dany serwis został stworzony, natomiast drugi czy po wywołaniu akcji logowania do Instagrama, użytkownik został przekierowany na odpowiednią, zewnętrzną stronę.

Następnym typem testów, które zostały stworzone, są to testy Selenium. Za pomocą frameworka angularowego Protractor, zostały napisane testy, sprawdzające większą część funkcjonalności. Poniżej pokazany zostanie przykład zaimplementowanych testów.

navigateToDashboard() {

return browser.get('/dashboard');

}

getDashboardRootContainer() {

return element(by.className('dashboard-root'));

}

getDashboardDataBtn() {

return element(by.className('refresh-btn'));

}

getDashboardUserData() {

return element(by.className('user-data'));

}

getDashboardNavigationBtn() {

return element(by.className('nav-btn'));

}

Powyżej został przedstawiony kod, za pomocą którego test Selenium jest stworzony. Służy on do sterowania działaniem przeglądarki, odpowiednio nawigując oraz pobierając elementy dostępne na stronie.

it('should redirect to dashboard and display proper page', () => {

page.navigateToDashboard();

expect(page.getDashboardRootContainer()).toBeTruthy();

expect(page.getDashboardUserData()).toBeTruthy();

expect(page.getDashboardDataBtn()).toBeTruthy();

expect(page.getDashboardNavigationBtn()).toBeTruthy();

});

Przykładowy test został przedstawiony powyżej. Jak można zauważyć, wykorzystuje wcześniej zdefiniowane funkcje w obiekcie page. Za ich pomocą, na początku przekierowuje przeglądarkę na stronę dashboardu, następnie wyciągając po kolei znaczące elementy, sprawdza czy pojawiły się one na stronie.

Ostatnimi testami przeprowadzonymi w ramach tej pracy, były to testy środowiskowe. Polegały one na udostępnieniu aplikacji dla potencjalnych klientów, w celu przetestowania działania oraz poprawności interfejsu. Pierwszą rzeczą, jaka została zauważona przy przeprowadzeniu tych testów, był problem z mechanizmem odświeżenia danych. Jak można było się dowiedzieć, mechanizm ten jest mało intuicyjny i niewygodny w użyciu. Jednakże biorąc pod uwagę, trudności jakie występują podczas pobierania danych bez potrzeby ciągłego zalogowania użytkownika, niestety mechanizm ten jest niezbędny do działania aplikacji. Drugim często zadawanym pytaniem w ramach, użycia aplikacji, było to pytanie o informacje, jakie aplikacja stworzona w ramach tej pracy otrzymuje na temat zalogowanego użytkownika. Często zewnętrzne serwisy po przekierowaniu na ich zewnętrzną stronę, wyświetlają informacje do jakich danych aplikacja będzie miała dostęp. Było to niepokojące dla użytkowników korzystających z systemu.

**6.4. Przykład działania**

W ramach niniejszej pracy, została utworzona aplikacja, udostępniająca raporty na temat użytkowników korzystających z portali społecznościowych. Portalami, które zostały poddane analizie były Facebook, Twitter i Instagram. W tym rozdziale, pokazane jest działanie oraz wygląd aplikacji końcowej. Przykłady widoków, dotyczą głównie raportów na temat obserwowanych osób.

Pierwszym widokiem (Rys.6.2) jest widok startowy aplikacji, dostępny dla niezalogowanego klienta. Udostępnia on trzy możliwości za pomocą których, użytkownik może dostać się do aplikacji. Po wybraniu jednej z nich, użytkownik zostaje przekierowany na zewnętrzną stronę potwierdzającą jego tożsamość, a następnie z powrotem do aplikacji.



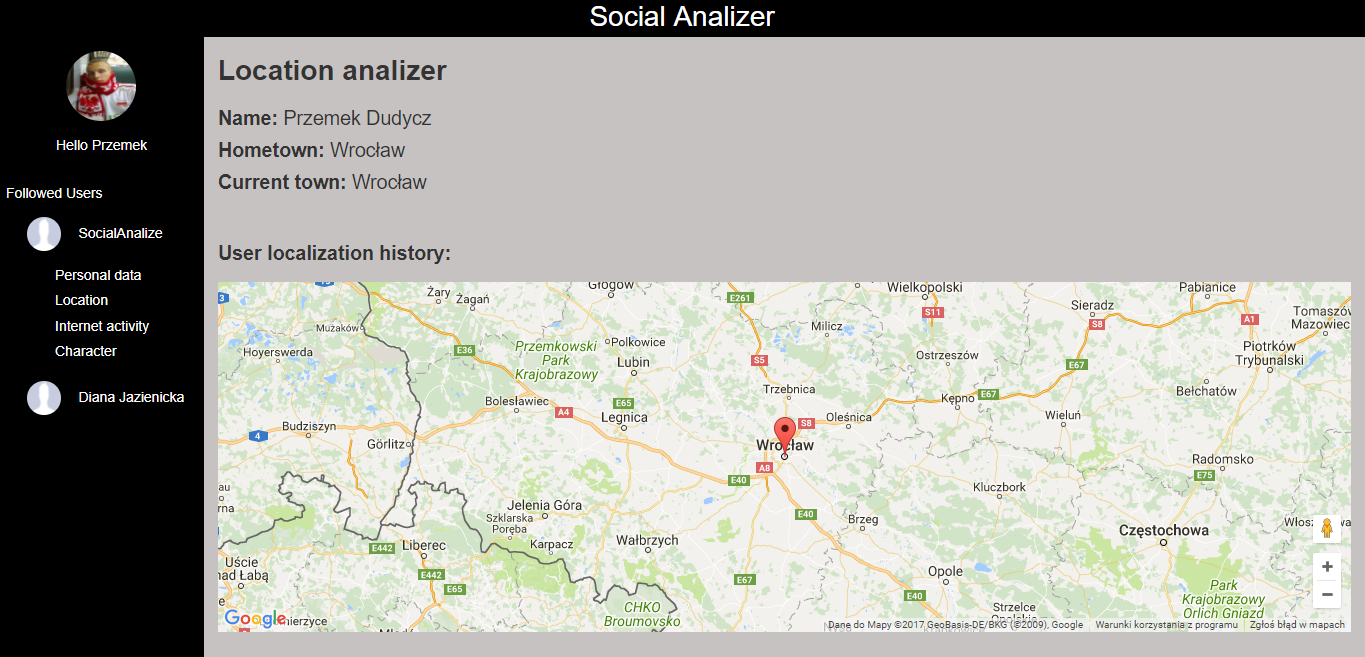
Rys.6.2. Strona startowa aplikacji

Rysunek 6.3 przedstawia widok strony nawigacyjnej w ramach obserwowanej osoby. Jest to bardzo ważna strona, ponieważ są tutaj przyciski, za pomocą których następuje odświeżanie danych, czyli pobranie ich z portali społecznościowych i przetworzenie przez Architekturę Lambda.



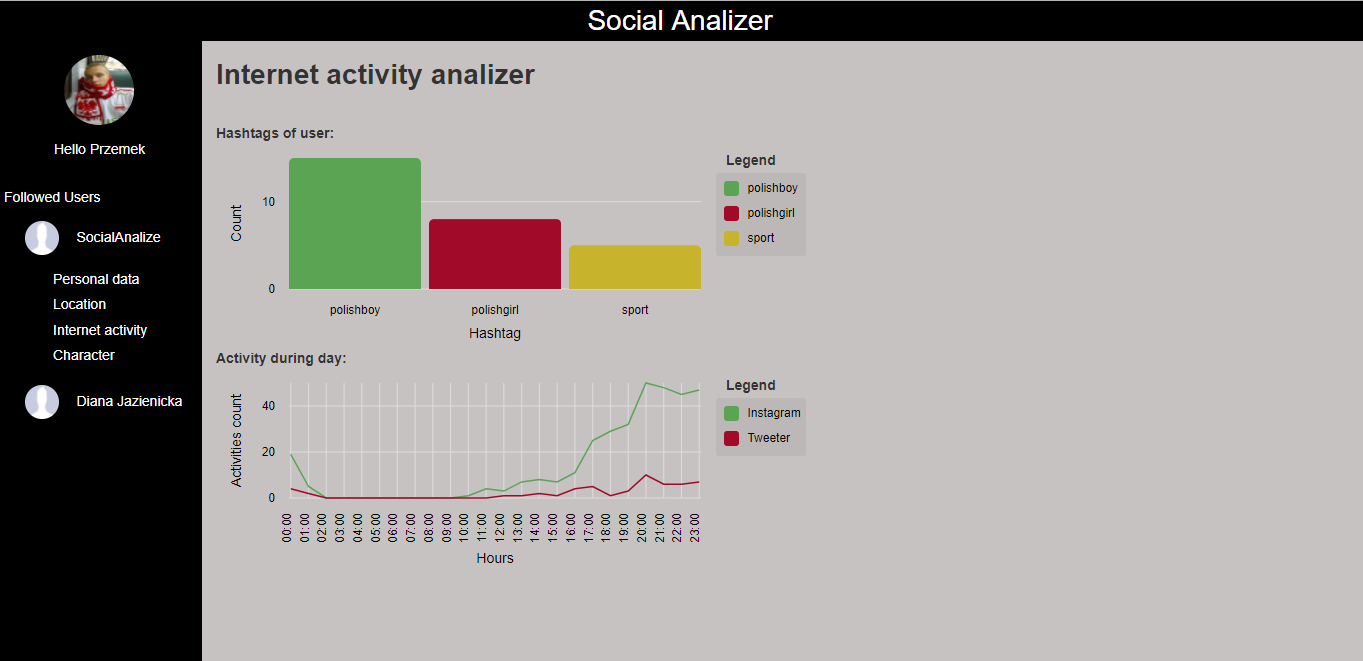
Rys.6.3. Strona nawigacyjna osoby obserwowanej

Na rysunku 6.4 przedstawiony został raport dotyczący danych lokalizacyjnych. Na początku raportu podane są podstawowe dane takie jak imię, nazwisko oraz miasta z których użytkownik pochodzi oraz w którym aktualnie przebywa. Poniżej pokazana jest mapa, na której markerami zaznaczone są miejsca, w których użytkownik przebywał. Po naciśnięciu na marker, pojawiają się szczegóły dotyczące lokalizacji.



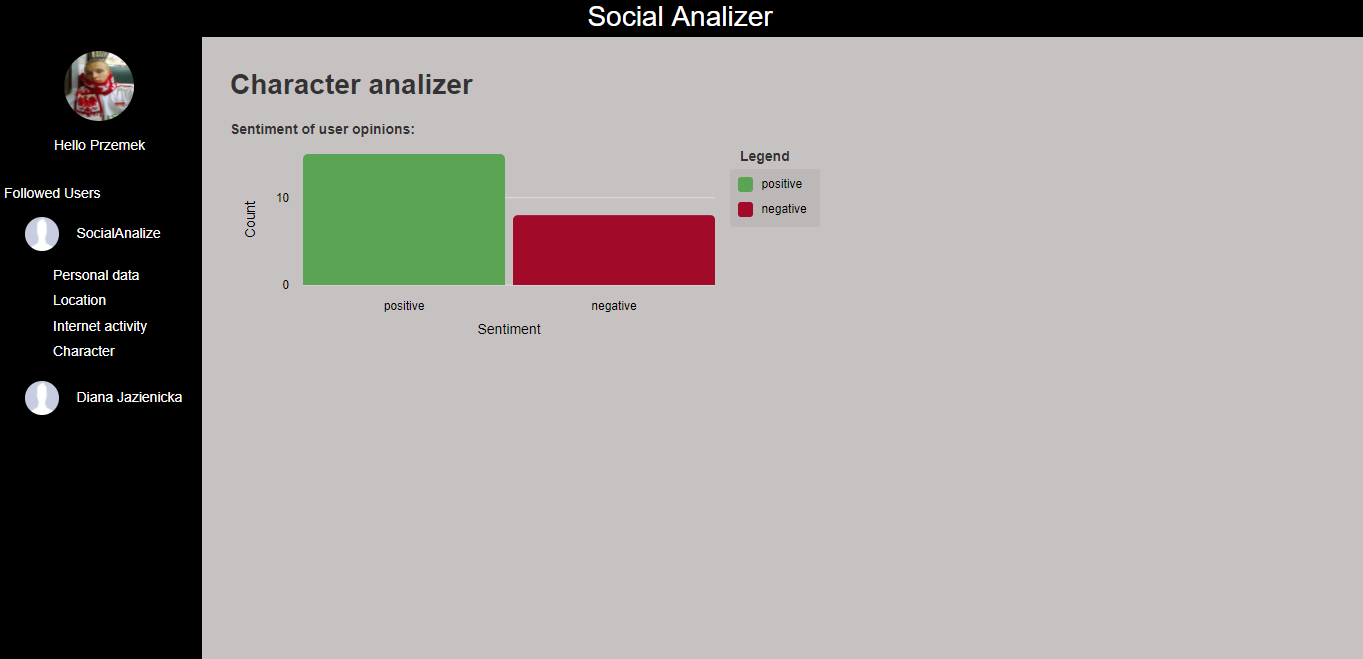
Rys.6.4. Strona raportu opartego o dane lokalizacyjne

Następna strona (Rys.6.5) przedstawia informacje zebrane przez aplikacje Social Analizer dotyczące aktywności internetowej. Przedstawia dwa wykresy – wykres hashtagów oraz wykres aktywności w trakcie dnia. Wykres hashtagów, przedstawia tematy oraz słowa kluczowe z nimi związane, jakimi użytkownik najbardziej się interesuje oraz którymi posługuje się na portalach społecznościowych. Natomiast wykres aktywności w trakcie dnia, pokazuje liniowy wykres, wskazujący na godziny w których użytkownik jest najbardziej aktywny.



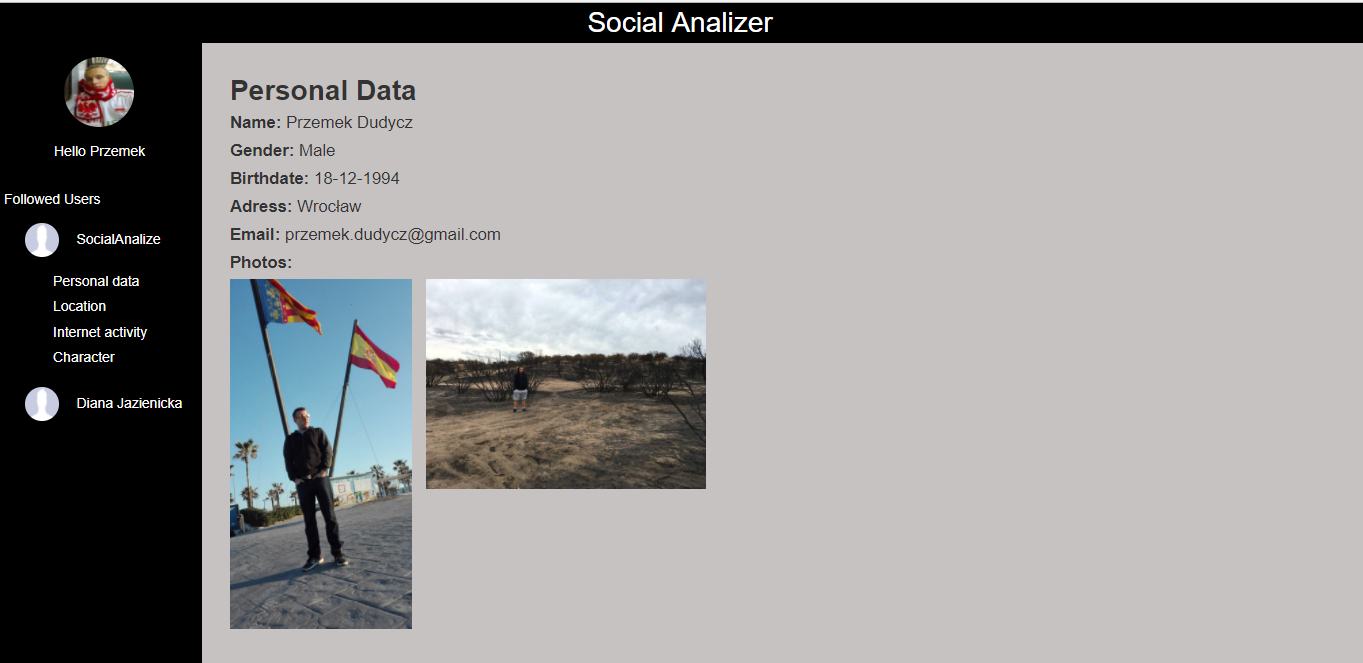
Rys.6.5. Strona raportu opartego o dane aktywności internetowej

Raport dotyczący danych związanych z charakterem, jest przedstawiony na rysunku 6.6. Widok ten, przedstawia wykres słupkowy, ukazujący stosunek treści pozytywnych do treści negatywnych udostępnianych przez użytkownika.



Rys.6.6. Strona raportu opartego o dane związane z charakterem

Ostatnim raportem, dostępnym w tworzonej w ramach tej pracy aplikacji, jest raport danych personalnych. Raport ten przedstawia podstawowe informacje na temat osoby, takie jak imię i nazwisko, płeć czy data urodzenia, oraz zebrane zdjęcia powiązane z profilami danego użytkownika.



Rys.6.7. Strona raportu opartego o dane personalne

**6.5. Podsumowanie**

Stworzenie dobrze działającej aplikacji jest zadaniem bardzo trudnym. Biorąc pod uwagę ilość występujących problemów, deweloper potrzebuje kierować się odpowiednimi zasadami oraz wzorcami, w celu stworzenia dobrze poukładanego kodu, pozwalającego na modyfikacje i odczytanie go w przyszłości. W tym celu zostały wykorzystane odpowiednie wzorce projektowe takie jak zasady SOLID czy CQRS. Korzystając z tych wzorców, zostały rozwiązane problemy związane z autentykacją, zapisem i odczytem danych z bazy oraz komunikacją przeprowadzoną z warstwą kliencką. Bardzo ważną cechą korzystania z wzorców jest łatwość, z jaką można napisać testy do konkretnej funkcjonalności. W ramach tego rozdziału, opisane zostały podstawowe testy z trzech zdefiniowanych typów – testów jednostkowych, akceptacyjnych i środowiskowych. Testy jednostkowe i akceptacyjne posłużyły do automatycznego sprawdzenia poprawnego działania systemu, natomiast testy środowiskowe posłużyły do otrzymania informacji zwrotnej od potencjalnych klientów, na temat funkcjonalności oraz poprawności interfejsu.