Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria systemów informatycznych

Aplikacja internetowa do rozliczania wspólnych wydatków

Jarosław Glegoła
Numer albumu 293092

promotor
dr inż. Roman Podraza

WARSZAWA 2021

Aplikacja internetowa do rozliczania wspólnych wydatków

Streszczenie. Celem tej pracy jest opisanie procesu projektowania i tworzenia aplikacji internetowej. Aplikacja udostępnia możliwość łatwiejszego rozliczania się z wydatków. W pracy opisana jest architektura serwera oraz aplikacji klienckiej, użyte technologie i paradygmaty, sposoby testowania oraz wizualna prezentacja aplikacji.

Słowa kluczowe: Aplikacja internetowa, GraphQL, Spring Framework, Architektura heksagonalna, React

A web application for settling common expenses

Abstract. The main goal of this thesis was to describe the process of designing and creating a web application. The application allows users to settle down expenses. In the thesis was described the architecture of the server and the client side, technologies and paradigms used, how the application was tested and visual presentation of the web application.

Keywords: Web application, GraphQL, Spring Framework, Hexagonal architecture, React

załącznik nr 3 do zarządzenia
nr 28 /2016 Rektora PW

Politechnika Warszawska

……..
miejscowość i data

……..
imię i nazwisko studenta

……..
numer albumu

…….…….
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

1. Wprowadzenie

Zdarza się w naszym życiu, że razem z innymi osobami kupujemy rzeczy, za które płaci tylko jedna osoba. W takiej sytuacji dosyć uciążliwe może być proszenie innych osób o zwrot kwoty, obliczanie owej kwoty, którą należy zwrócić płatnikowi, oraz śledzenie wszystkich płatności i wydatków, w których uczestniczyliśmy. W takiej sytuacji dobrym pomysłem byłoby użycie aplikacji lub programu komputerowego, który pomagałby nam w śledzeniu takich elementów naszego życia. W ramach tej pracy została utworzona właśnie taka aplikacja i została nazwana Aprint.

1.1. Opis przypadku biznesowego

1.1.1. Opis hipotetycznej sytuacji

Aby bardziej zilustrować przypadek biznesowy, możemy wyobrazić sobie sytuację, w której dwóch współlokatorów (nazwijmy ich Rafał i Kacper) mieszka w jednym mieszkaniu i obydwaj wydają pieniądze na rzeczy potrzebne na utrzymanie całego mieszkania. Pewnego dnia Rafał kupuje środki czystości. Jak możemy się domyślić, środki czystości będą używane przez Kacpra i Rafała, natomiast w tej hipotetycznej sytuacji tylko Rafał zapłacił w całości za owe produkty. Z tego powodu Kacper jest winny części kwoty Rafałowi. Obydwaj używając aplikacji, mogą w prosty sposób rozliczyć się z tego wydatku.

1.1.2. Użycie aplikacji

Na samym początku Rafał musi utworzyć obiekt wydatku na stronie internetowej, używając do tego dedykowanego formularza, wypełniając takie informacje jak nazwa wydatku, jego opis, datę kiedy ten wydatek został stworzony oraz osoby, które razem z nim z brali udział w wydatku (w tym wydatku tylko Kacpra).

Gdy wydatek zostanie utworzony, Kacper będzie mógł potwierdzić lub odrzucić udział w owej transakcji. Będzie mógł to zrobić na dedykowanym ekranie płatności, na którym będą odpowiednie przyciski oraz informacje dotyczące płatności, takie jak opis, data utworzenia wydatku itp. Jeżeli Kacper potwierdzi to, że brał udział w tym wydatku, będzie mógł nacisnąć przycisk *potwierdzam płatność*, a jeżeli się z tym nie zgadza, naciśnie przycisk *odrzucam płatność*.

Gdy wszyscy uczestnicy wydatku (w tym przypadku tylko Kacper) potwierdzi lub odrzuci udział, inicjator wydatku będzie mógł zatwierdzić uczestników, czyli kliknąć przycisk na ekranie wydatku o treści "potwierdzam użytkowników". To potwierdzenie nie mogło zostać zaimplementowane automatycznie, ponieważ inicjator wydatku może nie zgodzić się z uczestnikami i w takim wydatku może z nimi porozmawiać lub wyjaśnić niespójności.

Następnym krokiem tego scenariusza jest już sama akcja zapłaty za wydatek. Uczestnik wydatku Kacper może przejść na ekran płatności, na którym będzie już obliczona kwota,

którą Kacper musi zapłacić Rafałowi. Po płatności ze strony Kacpra będzie mógł on kliknąć przycisk "potwierdzam płatność"i uregulować w ten sposób należność.

Ostatnim elementem scenariusza jest zakończenie wydatku przez inicjatora wtedy, kiedy wszyscy uczestnicy uregulują swoje płatności. Będzie on mógł to zrobić na ekranie wydatku po kliknięciu w przycisk zakończ wydatek.

1.2. Układ pracy

Praca jest podzielona na 4 rozdziały, gdzie w pierwszym zostaną opisane wymagania, jakie aplikacja do zarządzania wspólnymi wydatkami powinna spełniać. W drugim rozdziale zostaną opisane użyte technologie i podejścia, które zostały użyte podczas pisania aplikacji. W kolejnym rozdziale zostaną opisane różne sposoby automatycznego testowania aplikacji internetowej. W ostatnim rozdziale zostanie przedstawiony wygląd i poszczególne ekrany aplikacji.

2. Specyfikacja wymagań

2.1. Słownik pojęć

Użytkownik
Osoba, która złożyła konto w aplikacji, posiadająca unikalny e-mail oraz inne dane osobowe.
Wydarzenie
Obiekt reprezentujący wydarzenie, do którego mogą dołączać użytkownicy aplikacji. Wydarzenie posiada nazwę, godzinę rozpoczęcia i zakończenia, listę uczestników, opis, współrzędne geograficzne miejsca wydarzenia, oraz listę wydatków.
Grupa
Obiekt reprezentujący grupę użytkowników, która chciałaby rozliczać wspólne wydatki, bez ustalonych ram czasowych lub ustalonego miejsca. Posiada nazwę, opis, listę uczestników oraz listę wydatków.
Wydatek
Obiekt reprezentujący pojedynczy wydatek utworzony w prawdziwym życiu przez użytkownika, który zawiera nazwę, opis, kwotę uiszczoną za daną transakcję, datę płatności oraz osoby, które według osoby tworzącej wydatek wspólnie brały w nim udział.
Inicjator wydatku
Osoba, która stworzyła wydatek w aplikacji.
Uczestnik wydatku
Użytkownik, który jest na liście uczestników w danym wydatku i posiada swoją własną płatność należącą do tego wydatku.
Płatność
Część wydatku, która posiada informacje dotyczące uczestnika wydatku. Płatność może znajdować się w stanie:

- oczekującym - użytkownik został zaproszony do wydatku i może on potwierdzić lub odrzucić w niej udział
- zaakceptowanym - użytkownik potwierdził udział w wydatku i zgodził się na zapłacenie części kwoty
- odrzuconym - użytkownik nie zgodził się na udział w wydatku
- opłaconym - użytkownik po zaakceptowaniu płatności opłacił swoją część
- zakończonym - po zakończeniu wydatku, do którego należy płatność, jest ona w stanie zakończonym

Zaproszenie
W trakcie zakładania grupy lub wydarzenia użytkownik ma możliwość wskazania uczestników. Po utworzeniu obiektu każdemu wybranemu użytkownikowi wysyłane jest zaproszenie, które wybrany użytkownik może zaakceptować (potwierdzić udział w wydarzeniu lub grupie) lub je odrzucić.

Znajomy
Użytkownik, który został zaproszony do listy znajomych. Tylko znajomi mogą być zapraszani do grup wydarzeń i wydatków.

2.2. Wymagania funkcjonalne

WF1.
Użytkownik może założyć konto w aplikacji, podając adres e-mail, hasło oraz imię i nazwisko.
WF2.
Użytkownik może zalogować się do aplikacji z użyciem adresu e-mail i hasła podanego przy rejestracji.
WF3.
Użytkownik może dodawać innych użytkowników do znajomych.
WF4.
Użytkownik może zakładać wydatki, podając takie informacje jak:

- nazwa
- rodzaj wydatku:
  - wydatek w ramach wydarzenia
  - wydatek w ramach grupy
  - wydatek bez grupy
- uczestników wydatku
- kwoty wydanej w ramach wydatku
- kwoty, która przypada na każdego użytkownika
- datę opłacenia wydatku
- opis

WF5.
Użytkownik może zarządzać stanem swoich płatności. Może je: akceptować, odrzucać i potwierdzać płatność.
WF6.
Użytkownik może zarządzać stanem swoich wydarzeń i grup. Może dodawać nowych uczestników, usuwać uczestników, zmieniać datę wydarzenia, opis, miejsce.
WF7.
Użytkownik może zarządzać stanem swoich wydatków. Może zmieniać ich opis datę, nazwę i kwotę.
WF8.
Użytkownik dostaje powiadomienia za każdym razem gdy:

- dostaje nowe zaproszenie do grupy lub wydarzenia
- zmienia się stan wydatku
- zmienia się stan płatności

WF9.
Użytkownik może zarządzać zaproszeniami, czyli przeglądać listę zaproszeń, akceptować lub odrzucać poszczególne zaproszenia
WF10.
Użytkownik posiada statystyki dotyczące swoich wszystkich wydatków, czyli jaką kwotę powinien oddać wszystkim innym użytkownikom oraz ile wszyscy inni użytkownicy powinni mu oddać.
WF11.
Użytkownik posiada podsumowanie bilansu wydatków dla każdego użytkownika osobno.

2.3. Wymagania niefunkcjonalne

WN1.
Aplikacja działa na najnowszych przeglądarkach: Chrome, Firefox oraz Edge.
WN2.
Aplikacja jest renderowana po stronie klienta z wykorzystaniem reaktywnej biblioteki Javascript.
WN3.
Komunikacja aplikacji z serwerem odbywa się poprzez bezpieczne połączenie https.
WN4.
Aplikacja komunikuje się z serwerem zgodnie ze specyfikacją GraphQL.
WN5.
Aplikacja poprawnie wyświetla się na rozdzielczościach z przedziału 360px - 1600px.
WN5.
Rozmiar skryptów klienckich nie powinien przekraczać 2MB.
WN6.
Autoryzacja odbywa się przy użyciu tokenów JWT.

3. Użyte technologie i narzędzia

Aplikacja jest podzielona na dwie główne części: część serwerową i część kliencką. Obie części są niezbędne do tego, aby użytkownik mógł w pełni korzystać ze wszystkich elementów systemu.

Część serwerowa jest odpowiedzialna za przetwarzanie danych, operacje na bazie danych, zarządzanie zapytaniami pochodzącymi z aplikacji klienckiej oraz zapewnieniem bezpieczeństwa użytkownika. Aplikacja została napisana w języku Kotlin z użyciem bardzo popularnej biblioteki Spring. Dodatkowo została także użyta biblioteka graphql-kotlin, która udostępnia przejrzysty interfejs umożliwiający w prosty sposób tworzenie serwisu, z którym można komunikować się zgodnie ze specyfikacją GraphQL.

Część kliencka jest odpowiedzialna za wyświetlanie aktualnych danych użytkownikowi, udostępnia przejrzysty interfejs, dzięki któremu użytkownik może zarządzać swoim kontem oraz powiązanymi z nim obiektami.

3.1. Warstwa serwera

Aktualnie istnieje wiele różnych podejść do pisania aplikacji serwerowych, które mają ułatwiać pisanie, rozwijanie i utrzymywanie większych systemów. Jednym z takich podejść jest architektura heksagonalna, która została użyta do stworzenia aplikacji.

3.1.1. Architektura

Podczas tworzenia aplikacji serwerowej kierowałem się podejściem zwanym architekturą heksagonalną (lub architektura portów i adapterów). Architektura heksagonalna została udokumentowana przez Alistaira Cockburna w 2005 [1]. Aktualnie jest bardzo popularnym podejściem do tworzenia mikroserwisów, ponieważ umożliwia ona w łatwy sposób rozbudowywanie aktualnego kodu aplikacji, jak i wprowadzanie zmian wynikających ze zmian w architekturze i wymagań systemu. Główne cechy architektury heksagonalnej to:

- oddzielenie części obsługi klientów serwisu, logiki biznesowej oraz strony serwerowej
- podział komponentów serwera na tzw. porty i adaptery

Logika biznesowa

Główną i najważniejszą częścią systemu jest logika biznesowa. Reguły biznesowe i zasady nimi rządzące nie powinny być w żaden sposób mocno bazować na innych komponentach systemu. Powinny zawierać tylko i wyłącznie elementy związane z logiką systemu, a elementy infrastruktury (jak np. bazy danych, zewnętrzne serwisy) powinny być zależne od komponentu logiki biznesowej. Odizolowanie tej części ma kilka zalet:

- Logika biznesowa jest bardzo łatwo testowalna jednostkowo. Jako że testy nie bazują na zewnętrznych narzędziach lub serwisach, tylko na ich interfejsach, możliwe jest w

łatwy sposób pisanie testów symulujących działanie zewnętrznych narzędzi. Rzeczy takie jak komunikacja z bazą danych lub operacje na żądaniach klienckich nie są wymagane do pisania takich testów, więc pisze się je szybko i są one bardzo czytelne.
- Kod źródłowy odpowiedzialny za logikę domenową jest czytelny i prosty, ponieważ nie zawiera elementów infrastruktury. Z tego powodu nawet osoby, które znają zasady biznesowe systemu, ale nie są osobami technicznymi, mogą być w stanie zrozumieć fragmenty kodu domenowego.

Wydruk 1. Przykład kodu domenowego aplikacji w języku Koltin

```
1  fun deleteParty(id: Long, currentUserId: Long): Party? {
2          val party = partyRepository.getTopById(id)
3
4          if (party?.owner?.id != currentUserId) {
5              throw UnauthorisedException()
6          }
7
8          partyRepository.removeParty(id)
9
10         return party
11 }
```

Jak można zauważyć, w powyższym kodzie nie ma odwołań do bazy danych ani innych części infrastruktury systemu. W kodzie zawarte są jedynie zasady dotyczące działania systemu.
- Bardzo łatwo można wymieniać części systemu niezwiązane z logiką biznesową np. bazy danych lub zewnętrzne serwisy. W dobie mikroserwisów, kontrakty z zewnętrznymi serwisami zmieniają się podczas działania aplikacji, więc gdy zmienia się np. interfejs do zewnętrznego serwisu, to jeżeli będziemy się trzymać kontraktu, na którym bazuje logika biznesowa, to nie będą potrzebne zmiany w głównej części aplikacji, a jedynie zmiany w adapterach aplikacji.

Adaptery

Adaptery są elementami które "wychodzą na świat", czyli są one odpowiedzialne za komunikację z zewnętrznymi serwisami. Adapterem jest np. część systemu odpowiadająca za zdefiniowanie kwerend i mutacji GraphQL'owych, komponenty komunikujące się z zewnętrznymi serwisami lub części komunikujące się z bazą danych. Adaptery są zależne od części domenowej aplikacji, czyli implementują interfejsy zdefiniowane w module logiki biznesowej. Komponenty logiki biznesowej używają adapterów z użyciem własnych klas reprezentujących elementy systemu. Żeby adaptery mogły korzystać z tych modeli,

muszą one je konwertować na swoją reprezentację. Przykładowo powiedzmy, że mamy model biznesowy *Grupa*.

Wydruk 2. Klasa *Grupa* w reprezentacji modelu domeny i adapteru

```
1 data class Grupa(
2     val id: Long?,
3     val imie: String?,
4 )
5 data class GrupaAdapter(
6     val id: Long?,
7     val imie: String?,
8     val rowId: String?,
9 )
```

Klasa *Grupa* jest częścią domeny aplikacji, a klasa *GrupaAdapter* częścią adaptera. Te dwie klasy różnią się tym, że klasa adapteru posiada dodatkowy atrybut związany z bazą danych *rowId*. Klasa domenowa nie potrzebuje tego atrybutu, a domena nawet nie powinna wiedzieć, że taki atrybut istnieje. W takim wypadku, jeżeli te dwie klasy się różnią, przy komunikacji pomiędzy częścią domenową i adapterową musi nastąpić translacja klas. Część domenowa nie powinna być zależna od innych części systemu, dlatego tę odpowiedzialność wykonują adaptery. Komponent domenowy, w przykładzie może komunikować z adapterem przez taki interfejs:

Wydruk 3. Interfejs domenowy adaptera bazy danych

```
1 interface AdapterBazyDanych {
2     fun zapiszNowaGrupa(grupa: Grupa): Grupa
3 }
```

Jak widzimy, komponent domenowy nie posługuje się nigdy klasą adaptera, tylko adapter, który implementuje ten interfejs, przeprowadza konwersję argumentu *grupa* przy wywoływaniu funkcji oraz przy zwracaniu wartości z tej funkcji.

Porty

Porty są zwykłym kontraktem pomiędzy komponentem logiki biznesowej a adapterami. Porty nie posiadają logiki, tylko służą do oddzielenia odpowiedzialności komponentów systemu. Przykładem portu może być wcześniej podany przykład interfejsu *AdapterBazyDanych* w wydruku 3.

3.1.2. Podział na pakiety

Aplikacja serwerowa jest podzielona na trzy główne pakiety:

- domain

- resolvers
- adapters

Domain
Pakiet domenowy udostępnia tzw. serwisy, czyli klasy, które są odpowiedzialne za logikę biznesową i udostępniają swój interfejs resolwerom. W pakiecie są też zdefiniowane modele biznesowe, czyli klasy, na których operują serwisy. Dodatkowo domena tworzy też porty bazodanowe, czyli interfejsy, udostępniające funkcje do komunikacji z bazą danych.
Resolvers
Resolwery są odpowiedzialne za komunikację z klientem serwera. Definiują one zbiór wszystkich możliwych typów kwerend oraz mutacji. Dodatkowo ten pakiet definiuje tzw. obiekty transferu danych (and. Data transfer object), które definiują typy obiektów, które serwer udostępnia klientowi. Resolwery są zazwyczaj prostymi klasami, które są odpowiedzialne za serializację i deserializację danych oraz wywoływaniem odpowiednich metod serwisów domenowych.
Adaptery
Adaptery są odpowiedzialne głównie za obsługę bazy danych. Klasy w zawarte w tym pakiecie implementują porty zdefiniowane w części domenowej. Udostępniają zbiór funkcji, które są używane przez serwisy, i które dają możliwość zarządzania odpowiednimi obiektami, czyli umożliwiają dodawanie, usuwanie lub edycję poszczególnych obiektów.

Aplikacja zawiera 6 obiektów bazodanowych reprezentujących cały system. Są to:

- wydatek
- powiadomienie
- grupa
- zaproszenie
- płatność
- użytkownik

Dla każdego obiektu jest stworzony w każdym pakiecie katalog odpowiedzialny za obsługę danego obiektu. Przykładowo dla obiektu *Grupa* zostanie stworzone:

- GrupaService - klasa, która udostępnia metody obsługujące grupy w domenie
- GrupaQuery - klasa, która używa *GrupaService* oraz definiuje kwerendy dostępne dla grupy
- GrupaMutation - klasa, która używa *GrupaService* oraz definiuje mutacje dostępne dla grupy
- GrupaRepository - port, który definiuje zbiór możliwych funkcji, których można użyć na bazie danych
- PersistentGrupaRepository - interfejs, który wykorzystuje interfejs Hibernate do definiowania możliwych funkcji, dzięki którym programista może komunikować się z bazą danych

- PgSqlGrupaRepository - klasa adapteru, która implementuje *GrupaRepository* i używa *PersistentGrupaRepository* do komunikacji z bazą danych

3.1.3. Główne użyte technologie

Do stworzenia aplikacji serwerowej zostały użyte następujące technologie:

Kotlin
Głównym językiem programowania, jaki został użyty do napisania aplikacji serwerowej, jest Kotlin [2]. Kotlin jest obiektowym, statycznie typowanym językiem programowania. Kotlin jest oparty na JVM, więc technologie i biblioteki, które zostały stworzone w języku Java, mogą być także używane w języku Kotlin.

Spring
Spring [3] jest darmowym i otwartym szkieletem aplikacyjnym do budowania aplikacji serwerowych w językach opartych na JVM. Posiada on dużą ilość potrzebnych przy tworzeniu aplikacji serwerowych funkcjonalności.
Podstawową funkcją udostępnianą przez Spring jest kontener do wstrzykiwania zależności. Spring umożliwia nam tworzenie pojedynczych komponentów, które później mogą być *wstrzyknięte* jako zależności do innych komponentów. Jeżeli komponent *A* zależy od komponentu *B*, czyli *A* używa *B*, aby móc zrealizować swoją własną funkcję, *A* nie musi sam tworzyć instancji *B*, tylko może np. dać znać Springowi, że jest zależny od komponentu *B* poprzez umieszczenie *B* np. jako parametr w konstruktorze. W takiej sytuacji, gdy obiekt *A* będzie tworzony, kontener będzie automatycznie wywoływał konstruktor *A* z odpowiednimi parametrami. Jest to bardzo pomocne przy pisaniu testów jednostkowych aplikacji. Przy użyciu techniki wstrzykiwania zależności, w testach jednostkowych bardzo łatwo jest symulować działanie innych części systemu, ponieważ jako parametr konstruktora możemy podać sztuczny serwis, który ma ustalony sposób działania przy wywoływaniu poszczególnych funkcji.
Dodatkowo Spring posiada bardzo dużo innych narzędzi, dzięki którym można budować duże serwisy, takie jak:

1. **Spring Boot** - narzędzie pozwalające w bardzo szybki sposób skonfigurować serwer aplikacji wraz z wszystkimi potrzebnymi elementami potrzebnymi do działania takiego serwisu np. autoryzacji lub komunikację z bazą danych. Dzięki niemu możemy otrzymać podstawową konfigurację do modułów wymienionych poniżej, czyli np. Spring Security lub Spring Data JPA
2. **Spring Security** - moduł, która odpowiada za bezpieczeństwo aplikacji. Domyślnie po zainstalowaniu tego modułu mamy dostęp do narzędzi, które pozwalają nam np. autoryzować użytkownika, konfigurować CORS, blokować niektóre adresy aplikacji itp.

3. **Spring Data JPA i Hibernate** - JPA jest to standard ORM dla języka Java. Dzięki JPA mamy mechanizmy, które pozwalają nam zarządzać bazą danych z poziomu kodu programu, bez użycia SQL. Przy takim podejściu klasy w języku Kotlin mogą być mapowane na elementy tabel w bazie danych przy zapisie lub ich edycji. Dodatkowo przy odczycie tych danych elementy tabel w bazie danych mogą być mapowane z powrotem na klasy Kotlinowe. Dzięki temu w łatwy sposób możemy zapewnić spójność pomiędzy modelami bazodanowymi a modelami w Kotlinie. Jako że JPA jest tylko standardem, aby móc robić takie rzeczy w Springu potrzebujemy narzędzia, które będzie ten standard implementować. Jednym z takich narzędzi które wybrałem, jest biblioteka Hibernate, która jest domyślnie wspierana przez Spring i może być łatwo skonfigurowana z użyciem Spring Boot.

PostgreSQL
PostgreSQL [4] jest jedną z kilku najpopularniejszych darmowych baz danych. Jest to relacyjna baza danych, która jest wspierana przez Spring. Baza danych jest obsługiwana z języka Kotlin i przechowywane są w niej wszystkie dane aplikacji.

GraphQL-Kotlin
Biblioteka GraphQL-Kotlin [5] jest zbudowana na innej bibliotece *graphql-java*, która ułatwia tworzenie aplikacji serwerowych udostępniających interfejs poprzez standard GraphQL. Udostępnia ona domyślnie funkcjonalność odbierania i parsowania zapytań GraphQL'owych, posiada funkcje pozwalające obsługiwać błędy w zapytaniach oraz tworzenia odpowiedzi do klienta zgodnie z tym standardem.

3.2. Warstwa komunikacji - GraphQL

W projekcie, zamiast użycia bardzo popularnego standardu do komunikacji między klientem a serwerem jakim jest REST, został użyty standard GraphQL [6].

3.2.1. Opis GraphQL

GraphQL jest specyfikacją utworzoną przez Facebooka, który opisuje sposób, w jaki dwie jednostki mogą się ze sobą komunikować. Jest to język zapytań, który może zostać użyty przez klienta, aby otrzymać dane, które wskaże w zapytaniu, w przeciwieństwie do REST, gdzie odpowiedzi serwera dla każdego endpointu są ustalone z góry. GraphQL nie jest protokołem sieciowym, a tylko kontraktem między jednostkami.

Serwer udostępniając interfejs GraphQL, definiuje tzw. schemat GraphQL, czyli statycznie i mocno typowany opis wszystkich możliwych operacji oraz typów dostępnych klientowi używającego danego interfejsu.

Wydruk 4. Przykład schematu GraphQL

```
type Grupa {
```

```
    id: String!;
    nazwa: String!;
    uczestnicy: [Uzytkownik!];
    opis: String;
}
type Uzytkownik {
    id: String!;
    nazwisko String!;
}
type Powiadomienie {
    id: String!;
    tresc: String!;
}
type Query {
    pobierzGrupe(idGrupy: String!): Grupa
}
type Mutation {
    zapiszNowaGrupa(grupa: Grupa!): Boolean
}
type Subscription {
    pobierajPowiadomienia(): Powiadomienie
}
```

Jak widać, każdy atrybut ma swój typ i każdy argument każdej operacji też jest opisany własnym typem. W niektórych implementacjach te typy mogą być sprawdzane podczas działania aplikacji serwerowej, czyli gdy klient wywoła operację, w której argument jest błędnego typu, serwer może zwrócić klientowi błąd o niezgodności typów. Jest to bardzo pomocne w pisaniu aplikacji, ponieważ możemy używać narzędzi (np. kompilatorów), które sprawdzają za nas poprawność danych na podstawie schematu GraphQL, zmniejszając przy tym możliwość pomyłki.

GraphQL wyróżnia trzy rodzaje operacji, których klient może użyć, aby uzyskać odpowiedź od serwera.

1. kwerendy
2. mutacje
3. subskrypcje

Wszystkie możliwe operacje muszą być zdefiniowane w schemacie GraphQL serwera. W Wydruku 4 te operacje są zdefiniowane w typach odpowiednio *Query*, *Mutation*, *Subscription*. Klient może tworzyć zapytania oparte tylko o operacje zdefiniowane w schemacie.

Kwerendy

Kwerendy są konstrukcjami pozwalającymi odpytywać serwer o dane, nie modyfikując ich. Dzięki nim możemy tworzyć zapytania, które zwracają nam informacje o obiektach znajdujących się w bazie danych.

Wydruk 5. Przykład kwerendy w języku zapytań GraphQL

```
query(idGrupy: String!) {
  pobierzGrupe(idGrupy: $idGrupy) {
    id
    nazwa
    uczestnicy {
      id
      nazwisko
    }
  }
}
```

Powyższa kwerenda przyjmuje jeden parametr, jakim jest identyfikator grupy. Dzięki temu identyfikatorowi możemy użyć danej kwerendy wielokrotnie w zależności od podanego parametru. Możemy też zauważyć, że powyższa kwerenda nie pobiera wszystkich atrybutów grupy. W typie *Grupa* jest także atrybut *opis*, ale jeżeli klient aktualnie nie potrzebuje tej informacji, może nie zamieszczać jej w kwerendzie. Dzięki takiemu zabiegowi, przy pobieraniu obiektów, klient jest w stanie optymalizować ruch sieciowy, pobierając tylko te dane, których potrzebuje.
Kwerendy mogą też posiadać zagnieżdżone typy. Grupa zawiera listę uczestników, więc w kwerendzie możemy też wyszczególnić, jakich atrybutów potrzebujemy w każdym obiekcie użytkownika.

Mutacje

Mutacje są strukturalnie podobne do kwerend, natomiast ich celem jest tworzenie lub zmiana stanu obiektów w aplikacji.

Wydruk 6. Przykład mutacji w języku zapytań GraphQL

```
mutation {
  zapiszNowaGrupa(grupa: {
    id: "0",
    nazwa: "nazwa",
    uczestnicy: [],
    opis: null
  })
}
```

Powyższa mutacja ma za zadanie zapisanie nowej grupy w bazie danych o podanych parametrach.

Subskrypcje
Subskrypcje są mechanizmem, który pozwala pobierać dane używając podejścia typu Push. W przypadku wywołania przez klienta subskrypcji, serwer nawiązuje stałe połączenie z klientem np. poprzez gniazda. Gdy nastąpi jakaś zmiana w systemie (np. utworzenie nowego powiadomienia), serwer ma za zadanie powiadomić klienta o tym wydarzeniu.

Wydruk 7. Przykład subskrypcji w języku zapytań GraphQL

```
1  subscribtion {
2    pobierajPowiadomienia() {
3      id
4      opis
5    }
6  }
```

Powody, dla których wybrałem GraphQL:

- Schemat GraphQL jest swego rodzaju kontraktem pomiędzy serwerem a klientem. Można ten schemat traktować jako dokumentację, której poprawność mogą zapewnić zewnętrzne automatyczne narzędzia, które ten schemat generują na podstawie kodu źródłowego aplikacji.
- Schemat jest mocno typowany, więc zmniejsza się ilość pomyłek popełnianych przez programistów, ponieważ znowu jesteśmy w stanie użyć narzędzi do automatycznej generacji typów opartych na schemacie.
- To podejście niweluje zjawisko nadmiernego pobierania danych, ponieważ w zapytaniu wskazujemy dokładnie, jakich danych potrzebujemy.
- Istnieje dużo implementacji standardu GraphQL w różnych językach, a w szczególności do Kotlina i Spring, które zostały użyte do implementacji aplikacji serwerowej.

3.2.2. Użycie GraphQL w aplikacji serwerowej

Do zaimplementowania GraphQL w aplikacji serwerowej została użyta biblioteka *graphql-kotlin*. Dzięki niej mogłem w łatwy sposób napisać interfejs spełniający wymagania GraphQL.

Operacje definiuje się poprzez tworzenie klas, które dziedziczą po odpowiednich klasach bazowych: *Query*, *Mutation*, *Subscription*.

Wydruk 8. Przykład definiowania kwerendy z użyciem *graphql-kotlin*

```
1 @Component
2 class ExpenseQuery : Query {
```

```
3       fun pobierzGrupe(
4               grupaId: String,
5       ): Grupa? {
6           return serwis.pobierzGrupe(grupaId)
7       }
8   }
```

Analogicznie definiujemy mutacje i subskrypcje.

Bardzo ważną funkcjonalnością tej biblioteki jest to, że automatycznie generuje ona schemat GraphQL na podstawie klas definiujących operacje oraz typów, które te operacje używają. Podstawowe (prymitywne) typy takie jak String lub Int są bezpośrednio zamieniane na odpowiednie typy zdefiniowane w GraphQL. Jeżeli jednak serwer posiada niestandardowy typ, który jest częścią systemu i który ma być walidowany przy zapytaniach klienta, programista może takie typy definiować z użyciem tej biblioteki. emphkotlin-graphql. Dzięki takiemu narzędziu możemy być pewni, że schemat będzie zawsze aktualny z kodem aplikacyjnym.

3.2.3. Użycie GraphQL w aplikacji klienckiej

Obsługę zapytań po stronie klienta aplikacji zaimplementowałem z użyciem biblioteki Apollo Client. Ta biblioteka udostępnia prosty interfejs do komunikowania się z serwisami z poziomu kodu Javascript. Używając Apollo Client możemy umieszczać kwerendy, mutacje i subskrypcje w kodzie źródłowym, a następnie wywoływać odpowiednie metody pochodzące z biblioteki, aby wysyłać żądania do serwera.

Bardzo ciekawym narzędziem, którego też użyłem podczas implementacji części klienckiej, jest *GraphQL Code Generator*. Jest to narzędzie, które na podstawie udostępnionego schematu GraphQL serwera jest w stanie generować odpowiednie typy w języku Typescript. Dzięki takiemu narzędziu w kodzie klienckim mam zawsze aktualny kontrakt z aplikacją serwerową. W przypadku stosowania REST aktualizacja odbywała się manualnie, więc była podatna na błędy.

W języku Javascript bardzo często zdarza się błąd wynikający z odnoszenia się do atrybutu, który nie istnieje dla danego obiektu. Zdarza z tego powodu, że kontrakt, którego używaliśmy, już nie jest aktualny. Automatyczna generacja kodu z połączeniem ze statycznym typowaniem języka Typescript jest w stanie w dużym stopniu zapobiegać takim błędom.

3.3. Warstwa kliencka

Aby użytkownik mógł używać aplikacji Aprint, potrzebuje on interfejsu, dzięki któremu będzie mógł zarządzać swoimi wydatkami. Aplikacja jest aplikacją internetową, której można używać poprzez przeglądarkę internetową. Aplikacja została stworzona z użyciem języka Typescript.

Główną zaletą aplikacji internetowych jest ich przenośność. Jeżeli programista napisze aplikację, która jest przystosowana do ekranów urządzeń mobilnych i komputerów, może być ona używana na praktycznie każdym urządzeniu, które posiada przeglądarkę internetową i dostęp do internetu. Gdyby aplikacja została napisana np. w formie aplikacji na Androida, ograniczyłbym ilość użytkowników mogących korzystać z aplikacji.

3.3.1. Renderowanie po stronie serwera a jednostronicowa aplikacja

Aby użytkownik mógł używać aplikacji w przeglądarce, po wejściu na stronę internetową serwer musi wysłać do klienta pliki definiujące strukturę, wygląd oraz logikę aplikacji. Pliki te to pliki HTML opisujące strukturę, arkusze stylów opisujące wygląd aplikacji oraz skrypty Javascript, które definiują zachowanie aplikacji. W obecnych czasach dwa najpopularniejsze sposoby na tworzenie i udostępnianie takich plików to:

1. Renderowanie po stronie serwera (ang. Server Side rendering)
2. Jednostronicowa aplikacja (ang. Single Page Application)

Renderowanie po stronie serwera

W przypadku renderowania po stronie serwera pliki wysyłane użytkownikowi są zawarte w szablonach. Szablony są to pliki w języku HTML, w których zapisywane są specjalne wyrażenia, które serwer przy obsłudze zapytania może dynamicznie zamieniać na dane użytkownika. Do takich szablonów serwer może też dołączać także arkusze stylów oraz skrypty Javascript.

Wydruk 9. Przykład szablonu w Freemaker

```
<html>
<body>
  <h1>Witaj ${uzytkownik.imie}!</h1>
  <p>Twoj stan wydatkow:</p>
  <p class="stan">
      Jestes winny ${uzytkownik.jestWinny}
  </p>
  <p class="stan">
      Inni sa ci winni ${uzytkownik.saMuWinni}
  </p>
</body>
</html>
```

Gdy serwer użyje takiego szablonu, może wypełnić podany szablon danymi użytkownika i zwrócić użytkownikowi poprawny kod HTML.

Wydruk 10. Przykład wyniku zwróconego z szablonu

```
<html>
```

```
<body>
  <h1>Witaj Rafal!</h1>
  <p>Twoj stan wydatkow:</p>
  <p class="stan">
      Jestes winny 35.00
  </p>
  <p class="stan">
      Inni sa ci winni 30.00
  </p>
</body>
</html>
```

W taki sposób jesteśmy w stanie tworzyć strukturę po stronie serwera indywidualnie dla każdego użytkownika.

Jednostronicowa aplikacja

W przypadku jednostronicowej aplikacji przeważającą część struktury strony definiujemy z użyciem skryptów Javascript. Serwer w przypadku otrzymania zapytania dotyczącego strony internetowej z przeglądarki zwraca użytkownikowi bardzo okrojoną strukturę HTML oraz dołącza do tej struktury kod Javascript, który przy uruchamianiu strony internetowej dynamicznie tworzy elementy w dokumencie strony przy użyciu interfejsu przeglądarki. Przy takim podejściu programista ma większą kontrolę nad aplikacją, dlatego to podejście jest aktualnie stosowane przy budowaniu interaktywnych i bardziej skomplikowanych interfejsów użytkownika.
Taką strukturę w kodzie Javascript zazwyczaj definiuje się używając specjalnych bibliotek Javascript, które ułatwiają taki sposób definicji strony.

Wydruk 11. Przykład aplikacji z użyciem biblioteki javascriptowej

```
function Aplikacja(uzytkownik) {
  return (
    <h1>Witaj {uzytkownik.imie}</h1>
    <p>Twoj stan wydatkow:</p>
    <p class="stan">
      Jestes winny {uzytkownik.jestWinny}
    </p>
    <p class="stan">
      Inni sa ci winni {uzytkownik.saMuWinni}
    </p>
  )
}
```

```
ReactDOM.render(
  <Aplikacja uzytkownik={uzytkownik} />,
  document.getElementById('root')
);
```

Wybór konkretnego rozwiązania

W aplikacji postanowiłem użyć technologii jednostronicowej aplikacji z kilku powodów:

- zachowanie strony bardziej przypomina aplikację na komputer lub telefon
- przy użytkowaniu aplikacji przejście pomiędzy widokami jest bardziej płynne, z racji tego, że nie musimy odświeżać całej strony przy każdej zmianie widoku
- jest dużo narzędzi i bibliotek do tworzenia jednostronicowych aplikacji
- w aplikacji jest więcej dynamicznych części niż statycznej zawartości, więc aplikacja jednostronicowa bardziej się sprawdzi w przypadku aplikacji do zarządzania wydatkami

3.3.2. Wybór biblioteki do tworzenia interfejsów użytkownika

Bardzo często aplikacje internetowe są budowane z użyciem bibliotek javascriptowych. Jedną z takich bibliotek jest React [7], która została użyta do stworzenia aplikacji Aprint.

React udostępnia programiście interfejs do deklaratywnego tworzenia widoków użytkownika. Struktura aplikacji jest napisana za pomocą komponentów, czyli głównie funkcji w języku Javascript, które zwracają obiekty reprezentujące część wirtualnego DOM-u przeglądarki. Dodatkowo React udostępnia nam też tzw. reaktywność, czyli automatyczną aktualizację widoków, przy każdej zmianie stanu aplikacji.

Reaktywność w React

Biblioteka React udostępnia nam deklaratywny mechanizm aktualizacji widoków aplikacji, po zmianie stanu. Używając udostępnionej przez bibliotekę React funkcji zarządzania stanem, możemy w łatwy sposób utrzymywać spójność widoków ze stanem.

Wydruk 12. Użycie mechanizmu zarządzania stanem w React

```
function Aplikacja() {
  const [wartosc, ustawWartosc] = React.useState(1);

  return (
    <div>
      <h2>{wartosc}</h2>
      <button onClick={() => ustawWartosc(wartosc - 1)}>
        -
      </button>
      <button onClick={() => ustawWartosc(wartosc + 1)}>
        +
```

```
        </button>
      </div>
    );
}
```

Powyższy przykład przedstawia przykład aplikacji, która wyświetla licznik. W tagu *h2* jest wyświetlana aktualna wartość licznika, a dwa przyciski pozwalają ją zwiększyć lub zmniejszyć o 1. W tym przykładzie możemy zauważyć, że nie aktualizujemy widoku (wartości licznika) ręcznie, tylko aktualizujemy stan aplikacji. Jeżeli ten stan zmienimy (używając funkcji *ustawWartosc*) React automatycznie aktualizuje widok w taki sposób, aby wyświetlał wartość zgodną z tym, co jest przetrzymywane w stanie. Do tego celu React wykorzystuje technikę zwaną wirtualnym DOM-em.

Wirtualny DOM

Wirtualny DOM jest techniką pozwalającą w prosty i szybki sposób przeprowadzać operacje na elementach struktury aplikacji internetowej. Elementy rodzime przeglądarki są zazwyczaj skomplikowanymi i rozbudowanymi obiektami. Operacje na nich, takie jak dodawanie, edycja czy usuwanie, mogą trwać dosyć długo, ponieważ przy każdej takiej operacji przeglądarka musi obliczyć klasy CSS, obliczyć poprawne położenie elementów a na końcu umieścić w poprawnych miejscach konkretne elementy. Wirtualny DOM pozwala nam przyspieszyć takie operacje.

Wirtualny DOM jest tym, na co wskazuje nazwa: wirtualną reprezentacją drzewa DOM przeglądarki. Dzięki niej programista jest w stanie przedstawić aplikację w postaci zwykłego obiektu Javascript. Obiekt ten zawiera informacje na temat wszystkich elementów, które aktualnie znajdują się w strukturze DOM.

Gdy programista używa biblioteki React inicjalnie tworzy ona taką wirtualną reprezentację DOM na podstawie komponentów, które napisaliśmy i zawartego w nich stanu, a następnie tworzy je w przeglądarce, używając do tego rodzimego interfejsu przeglądarki. Następnie przy każdej zmianie stanu aplikacji, React odtwarza ten sam proces, ale już z innymi danymi pochodzącymi ze zmienionego stanu. Taka czynność (czyli odtworzenie całej struktury DOM) wykonana na dokumencie przeglądarki byłaby bardzo kosztowna, natomiast wykonanie takiej operacji na zwykłych obiektach jest bardzo szybkie.

Gdy React posiada już dwie wersje (przed zmianą stanu i po jego zmianie) jest on w stanie obliczyć różnice pomiędzy dwoma reprezentacjami używając heurystycznego algorytmu [8].

Gdy React zna już różnice spowodowane zmianą stanu, jest on w stanie nanieść niezbędne zmiany w prawdziwej strukturze DOM, aby odwzorować stan aplikacji w widokach.

3.3.3. Użyte technologie

Podczas pisania części klienckiej użyłem następujących technologii:

Typescript
Typescript [9] jest językiem programowania, który jest nadzbiorem języka Javasrcipt. Javasrcipt jest językiem programowania, który jest wykorzystywany przez przeglądarki do kontrolowania stron internetowych przez programistów. Jedną z cech Javascript jest to, że jest on dynamicznie typowany. Podczas pisania aplikacji w Javascript zmienne nie mają zdefiniowanego typu, przez co podczas pisania aplikacji mogą pojawiać się błędy związane z niepoprawnym używaniem zmiennych. Typescript jest nadzbiorem Javascript i bardzo ważną rzeczą, którą dodaje do Javascript, jest właśnie statyczne typowanie. Typescript zawiera kompilator, który sprawdza poprawność kodu już na etapie pisania kodu w przeciwieństwie do Javascript, gdzie potencjalne błędy syntaktyczne zostaną ukazane programiście dopiero przy uruchamianiu programu. W przypadku dużych aplikacji internetowych liczących wiele tysięcy linii kodu pomoc kompilatora jest wyjątkowo pomocna. Dodatkowym atutem jest wsparcie narzędzi programistycznych. Gdy zintegrowane środowisko deweloperskie ma informacje na temat typów klas, obiektów oraz interfejsów aplikacji, jest w stanie łatwiej podpowiadać np. odpowiednie atrybuty obiektu podczas pisania kodu, co znacznie przyspiesza pisanie aplikacji.

Eslint
Eslint [10] jest narzędziem ułatwiającym pisanie kodu w Typescript i Javascript. Eslint analizuje składnię kodu programisty i wyświetla komunikaty, które mogą powodować błędy w aplikacji np. zadeklarowanie dwa razy tej samej zmiennej lub brak zdefiniowanego typu w deklaracji funkcji. Eslint jest też w stanie analizować elementy kodu niezwiązane z logiką aplikacji jak wcięcia w kodzie lub białe znaki, przez co pomaga utrzymać styl kodu w całej aplikacji. Eslint posiada też funkcjonalność naprawiania błędów, co może znacznie przyspieszyć tworzenie aplikacji.

Create React App
Jest to biblioteka, która jedną komendą pozwala utworzyć podstawową strukturę projektu React. Podczas pisania aplikacji utworzenie takiego wstępnego projektu może być czasochłonne z powodu dużej liczby konfiguracji, które trzeba utworzyć, aby móc używać Reacta w przeglądarce. Create React App [11] udostępnia taką konfigurację, która zawiera np. skonfigurowany proces budowania aplikacji, automatyczne odświeżanie aplikacji po każdej zmianie w kodzie, skonfigurowane narzędzia tj. eslint, Typescript.

AntD
AntD jest biblioteką stworzoną dla biblioteki React, która udostępnia zbiór komponentów, które spełniają zasady systemu Ant Design. Ant Design jest zbiorem zasad, którymi programista może się kierować podczas implementacji interfejsu użytkownika, aby interfejs wydawał się nowoczesny i przystępny tj. odpowiednia kolorystyka, rozmieszczenie i

wygląd elementów itp. AntD natomiast udostępnia komponenty zaimplementowane przy użyciu biblioteki React, które te zasady spełniają.

Apollo Client
Wcześniej wspomniany Apollo Client [12], który udostępnia przystępny interfejs do komunikacji poprzez GraphQL.

GraphQL Code Generator
Wcześniej wspomniana biblioteka, która na podstawie schematu GraphQL udostępnianego przez serwer, generuje odpowiednie typy w języku Typescript dla aplikacji.

4. Testowanie

Testy automatyczne są bardzo ważną częścią każdej aplikacji. Pozwalają one w łatwiejszy sposób wprowadzać zmiany do aplikacji, mając większą pewność, że zmiany nie wprowadziły niepożądanych błędów do kodu źródłowego. Dodatkowo podczas procesu pisania kodu aplikacji, dzięki takim testom programista jest w stanie w łatwy sposób sprawdzić poprawność rozwiązań w wielu różnych sytuacjach.

Aplikacja została przetestowana na trzy różne sposoby:

1. Jednostkowo
2. Integracyjnie
3. Z użyciem wizualnej regresji

4.1. Testowanie jednostkowe aplikacji klienckiej

Testowanie jednostkowe pozwala w prosty sposób przetestować poszczególne komponenty aplikacji w izolacji. Komponenty w bibliotece React to zwykłe funkcje, więc można by pomyśleć, że możemy je testować jak funkcje, jednak jest kilka rzeczy, o których musimy pamiętać.

4.1.1. Testowanie komponentów React

Jednym z powodów, dla którego trudno jest testować komponenty jak zwykłe funkcje, jest to, że komponenty mogą posiadać stan. Jak widać na wydruku 12, używając funkcji *React.useState* dołączamy do funkcji stan, który może się różnić przy różnych wywołaniach danego komponentu. Z tego powodu kolejne wywołania tej samej funkcji mogą zwracać inny wynik.

Kolejnym powodem, dla którego trudno by było przetestować komponent, bazując tylko i wyłącznie na jego zwróconym wyniku, jest to, że wartością zwracaną przez komponent jest zwykły obiekt, który reprezentuje część wirtualnego DOM-u. Natomiast my jako programiści interpretujemy tę część wirtualnego DOM-u jako prawdziwe elementy przeglądarki, dlatego asercje, które sprawdzałyby poprawność wirtualnego DOM-u mogłyby być nieczytelne i trudne w utrzymaniu.

Dodatkowo komponenty mogą używać interfejsu przeglądarki, więc aby w całości przetestować komponent, musimy być w stanie też sprawdzić, jak komponenty wchodzą w interakcje z przeglądarką.

Aby móc poprawnie i efektywnie przetestować jednostkowo komponenty aplikacji zostały użyte dwie biblioteki: *react-testing-library* [13] oraz *Jest* [14].

4.1.2. Jest

Jest to biblioteka, która udostępnia programiście interfejs, który umożliwia testowanie kodu napisanego w języku Javascript, w szczególności komponenty React. Aby zasymulować środowisko przeglądarki, *Jest* używa sztucznego środowiska o nazwie *jsdom*, dzięki

któremu programista może testować kod w środowisku, w którym kod Javascript będzie wykonywany, czyli w przeglądarce. Biblioteka *Jest* została stworzona przez Facebook'a i jest nadal rozwijana.

Jest posiada dużo funkcji, które ułatwiają testowanie kodu Javasrcipt.

Interfejs do tworzenia asercji

Jest udostępnia prosty i zrozumiały interfejs, dzięki któremu możemy tworzyć bardzo czytelne testy. Nazwy asercji są bardzo intuicyjne i czyta się je jak prawdziwe zdania w języku angielskim.

Wydruk 13. Przykład testu napisanego w bibliotece *Jest*

```
1  test('funkcja powinna byc zawolana', () => {
2    // zakladajac
3    let funkcja = jest.fn();
4
5    // kiedy
6    funkcja();
7
8    // wtedy
9    expect(funkcja).toEqual(funkcja);
10   expect(funkcja).toHaveBeenCalledTimes(1);
11 })
```

Równoległe uruchamianie testów

Jest pozwala na uruchamianie testów jednocześnie w wielu wątkach, przez co testy wykonują się nawet kilka razy szybciej, niż gdyby odpalało się je pojedynczo.

Migawki

Jest udostępnia interfejs, dzięki któremu możemy tworzyć tzw. migawki (ang. snapshots). Migawki są techniką pozwalającą testować zmiany, które mogły nastąpić pomiędzy kolejnymi uruchomieniami testów.

Wydruk 14. Przykład testu migawkowego

```
1  function funkcjaDoMigawki() {
2    return '<div>Witaj swiecie!</div>';
3  }
4
5  test('funkcja powinna zwracac poprawny wynik', () => {
6    expect(funkcjaDoMigawki).toMatchSnapshot();
7  })
```

Powyżej w teście widzimy, że funkcja *funkcjaDoMigawki* zwraca tag *div* z tekstem w środku. W teście moglibyśmy sprawdzić zawartość zwróconej wartości przez funkcję, lecz wymagałoby to ręcznego wpisania wartości w asercji. W przykładzie jest to tylko jedna linijka, ale w przypadku komponentów React, które mogą zwracać nawet kilkadziesiąt linijek, wpisywanie ręczne zwracanej wartości mogłoby się okazać dosyć trudnym zadaniem. Migawki rozwiązują ten problem. *toMatchSnapshot* podczas pierwszego uruchomienia zapamiętuje wartość zwróconą przez funkcję, a przy następnych uruchomieniach testu sprawdza, czy wynik zgadza się z wynikiem z poprzedniego uruchomienia. Jeżeli te dwie wartości nie będą się zgadzać, *Jest* zwróci programiście, że test nie przeszedł. W takiej sytuacji programista może zaakceptować nowy wygląd migawki lub poprawić błąd, jeżeli zmiana była niezamierzona. Takie testy są właśnie stworzone po to, aby nie wprowadzać niepożądanych zmian do kodu aplikacji.

4.1.3. React-testing-library

React-testing-library natomiast rozwiązuje problem czytelności i łatwości testowania komponentów. Dzięki tej bibliotece jesteśmy w stanie testować zachowanie komponentów, używając do tego bardzo prostego interfejsu, który pozwala na wykonywanie takich operacji jak:

- łatwe tworzenie komponentów w wirtualnym środowisku
- wykonywanie akcji użytkownika np. klikanie, najeżdżanie myszką itp.
- łatwe wyszukiwanie elementów i sprawdzanie ich stanu lub atrybutów

Wydruk 15. Przykład testu jednostkowego przy użyciu React-testing-library

```
1     test('gdy nazwa wydatku jest za dluga powinien pojawic sie blad', () => {
2     // given
3     const { getByLabelText, queryByText } = render(<FormularzWydatku />);
4
5     // when
6     const poleZNazwa = getByLabelText('Nazwa wydatku');
7
8     await userEvent.type(poleZNazwa, 'x'.repeat(51));
9
10    // then
11    const blad = queryByText(
12      "Pole nie moze miec wiecej niz 50 znakow."
13    );
14
15    expect(blad).toBeInTheDocument();
16  });
```

Powyżej w teście możemy zauważyć, że symulujemy wpisywanie do pola *Nazwa wydatku* 51 znaków, a następnie sprawdzamy, czy w komponencie wyświetlił się odpowiedni błąd. Dzięki takim testom komponentów możemy w łatwy sposób przetestować zachowanie komponentów w wielu różnych scenariuszach.

Testowanie jednostkowe ma natomiast kilka ograniczeń:

- nie pozwala sprawdzać integracji aplikacji z serwisami zewnętrznymi np. komunikacji z serwerem
- jako że Jsdom tylko symuluje środowisko przeglądarki, w aplikacji mogą pojawiać się błędy, które pojawiają się w prawdziwej przeglądarce. Takich błędów testy jednostkowe nie są w stanie wykazać

4.2. Testowanie integracyjne aplikacji klienckiej

Testowanie integracyjne pozwala rozwiązać te wyzwania, które wystąpiły przy testowaniu jednostkowym.

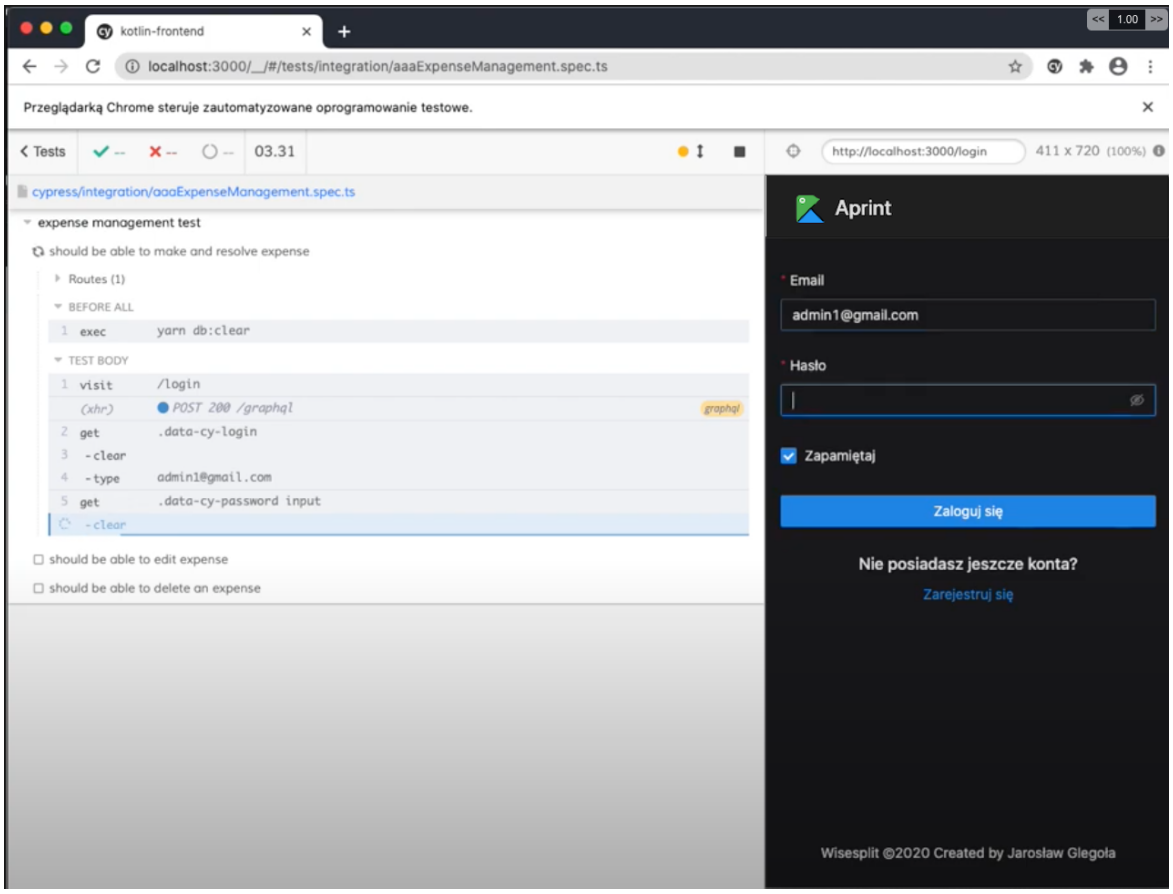
Do testów integracyjnych użyłem biblioteki o nazwie *Cypress.io* [15]. Biblioteka ta pozwala pisać testy, które potem są wykonywane w prawdziwej instancji przeglądarki. Cypress udostępnia interfejs, dzięki któremu możemy wykonywać akcje użytkownika (np. wpisywanie tekstu, klikanie itp.). Pozwala także sprawdzać poprawne działanie zewnętrznych serwisów, jak i komunikację z nimi.

Wydruk 16. Przykład testu integracyjnego z użyciem biblioteki Cypress

```
1   it('wydatek powinien sie usunac', () => {
2     // register
3     cy.visit('/login');
4     logIn(1);
5
6     // create new expense
7     cy.contains('Przykladowy opis zmieniony').click();
8     cy.contains('Usun').click();
9     cy.contains('OK').click();
10
11    cy.contains('Przykladowy wydatek zmieniony').should('not.exist');
12    cy.contains('Przykladowy opis zmieniony').should('not.exist');
13  });
```

Powyżej przedstawiony jest przykładowy test integracyjny aplikacji. Przy użyciu interfejsu Cypress możemy sterować przeglądarką i tworzyć testy, które symulują akcje użytkownika w instancji przeglądarki.

Cypress udostępnia też podgląd testów *na żywo*. Dzięki temu, gdy test będzie nieudany, możemy w łatwy sposób stwierdzić, w którym miejscu aplikacji wystąpił błąd. Na rysunku

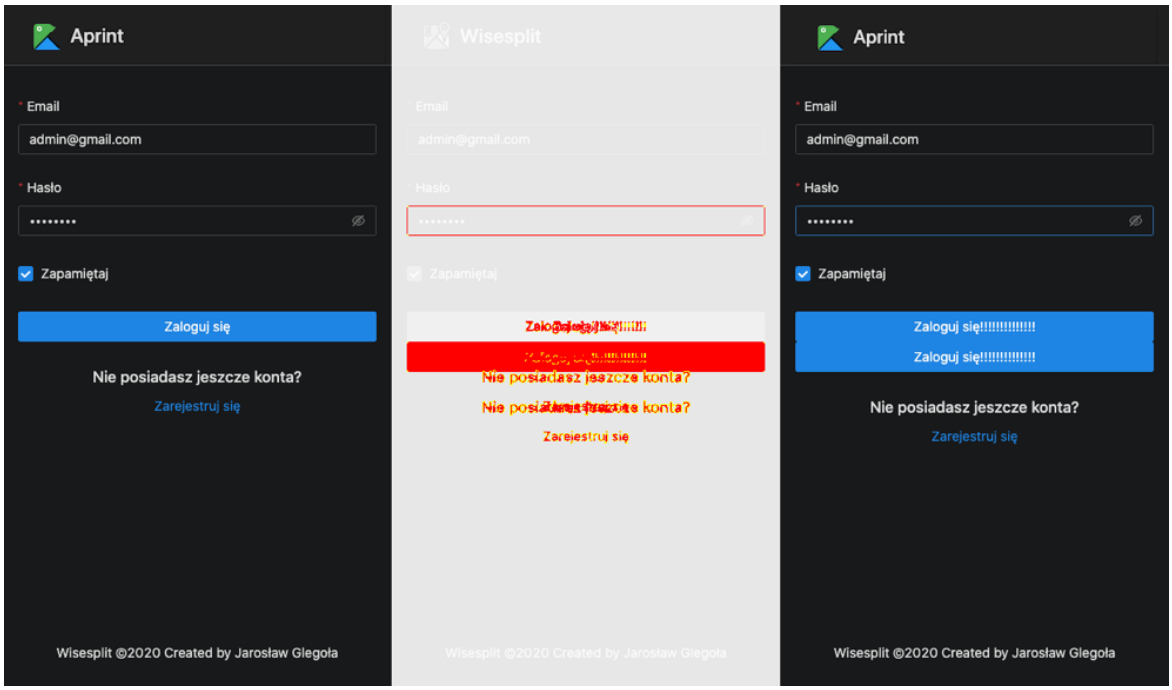


Rysunek 4.1. Panel Cypress.io

4.1 widzimy przykład testu w panelu Cypress. Po lewej stronie są wyświetlane wszystkie komendy wywoływane przez bibliotekę Cypress, a po prawej stronie wynik tych akcji w przeglądarce.

4.3. Testy wizualnej regresji aplikacji klienckiej

Testy wizualnej regresji zapewniają wizualną poprawność interfejsu użytkownika. Dzięki nim mogłem programowo symulować sytuacje biznesowe, a następnie przy pomocy biblioteki Cypress.io robić zrzuty ekranu. Testy wizualnej regresji działają podobnie jak testy migawkowe, tylko zamiast zapamiętywania wartości zwróconej przez funkcję Cypress zapamiętuje zrzut ekranu przeglądarki. Gdy pierwszy zrzut ekranu aplikacji został zrobiony podczas pierwszego uruchomienia testu, każde następne uruchomienie tego samego testu będzie robiło kolejny zrzut, a następnie porównywało z poprzednim. W taki sposób możemy sprawdzać, czy po zmianach w kodzie nie nastąpiła niepożądana zmiana w wyglądzie aplikacji. Jeżeli takie dwa zrzuty się różnią, test wyświetli komunikat o błędzie, a następnie wyświetli różniące się zrzuty ekranu i wyróżni wszystkie różniące się piksele obu zdjęć. Jeżeli natomiast zmiana wyglądu była zamierzona, programista może



Rysunek 4.2. Przykład wyniku nieudanego testu wizualnej regresji

zaakceptować zmiany w teście i od tego momentu w następnych testach porównywać z nowym wyglądem.

4.4. Testy integracyjne aplikacji serwerowej

Aplikację serwerową testowałem głównie integracyjnie. Do pisania testów wybrałem język Groovy i bibliotekę Spock w środowisku JUnit. Testy integracyjne serwera nie różnią się w dużej mierze od testów integracyjnych aplikacji klienckiej. Jedyną różnicą jest to, że nie posiadamy wizualnej części. Testy integracyjne testują wszystkie elementy aplikacji serwerowej: zapytania klienckie, logikę biznesową oraz integrację z bazą danych.

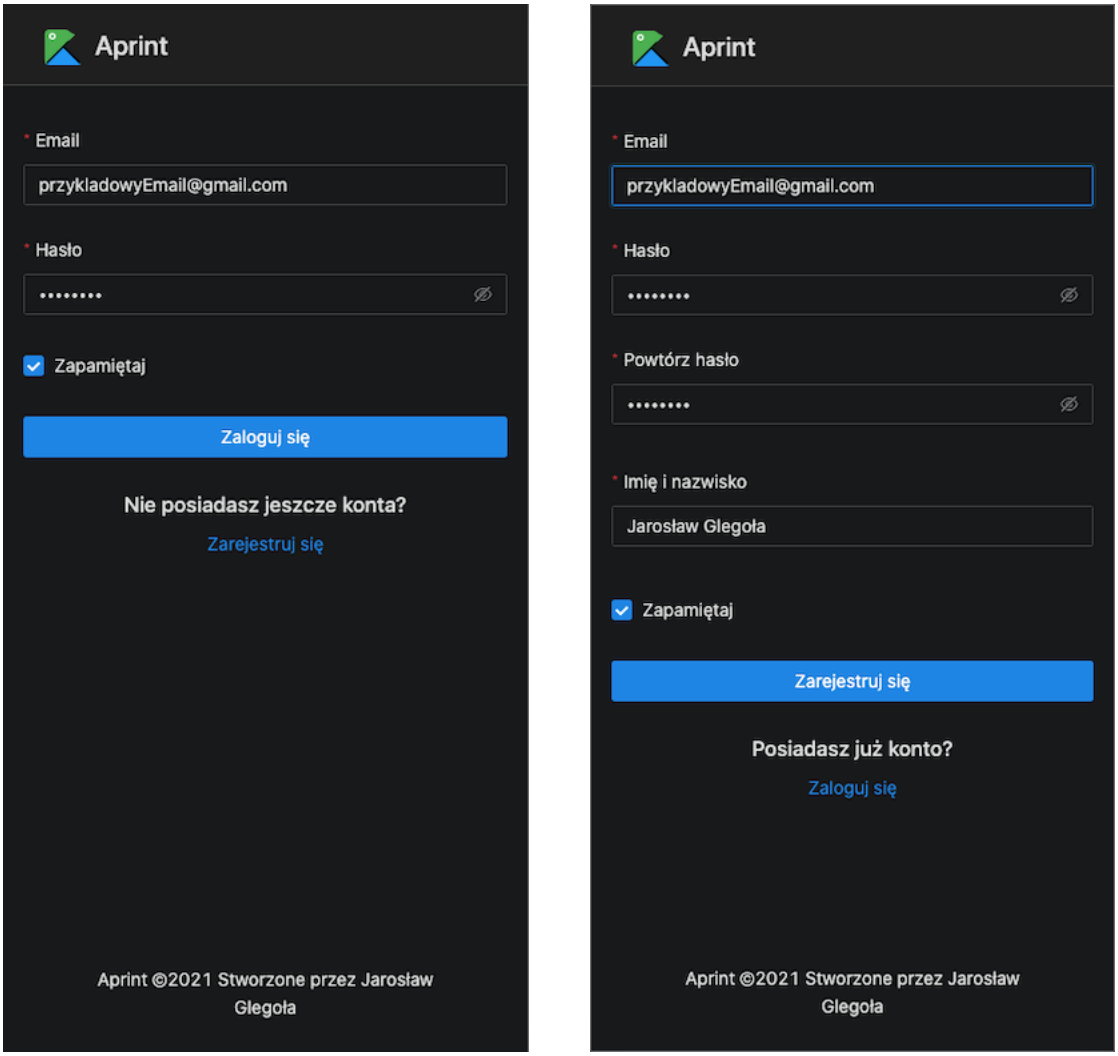
5. Prezentacja aplikacji

W tym rozdziale zostaną przedstawione poszczególne ekrany występujące w aplikacji w różnych stanach. Do poszczególnych sekcji będą też podane wymagania, które dane ekrany spełniają. W każdym podrozdziale zaprezentuję jeden ekran aplikacji.

Aplikacja została stworzona w stylu ciemnym, zgodnie ze wymaganiami specyfikacją Ant Design. Przykłady zostaną zaprezentowane w wersji na telefony komórkowe, lecz aplikacja działa poprawnie z ekranami tabletów oraz większych ekranów komputerów i laptopów.

5.1. Ekran logowania

Ekran logowania jest połączony z ekranem rejestracji. Na ekranie widnieje przycisk *Zarejestruj się*, który po kliknięciu sprawia, że pojawiają się dwie dodatkowe kontrolki do wprowadzania tekstu, gdzie użytkownik musi powtórzyć hasło oraz podać imię i nazwisko, które nie są wymagane przy formularzu logowania. W kroku rejestracji widnieje przycisk *Zaloguj się*, który ukrywa te dwie kontrolki. Zmiana pomiędzy logowaniem a rejestracją nie powodują utraty danych zawartych w elementach. Po poprawnym wypełnieniu formularza użytkownik zostaje przekierowany na ekran listy wydatków. Ekran spełnia wymagania WF1 oraz WF2.



(a) Krok logowania **(b)** Krok rejestracji

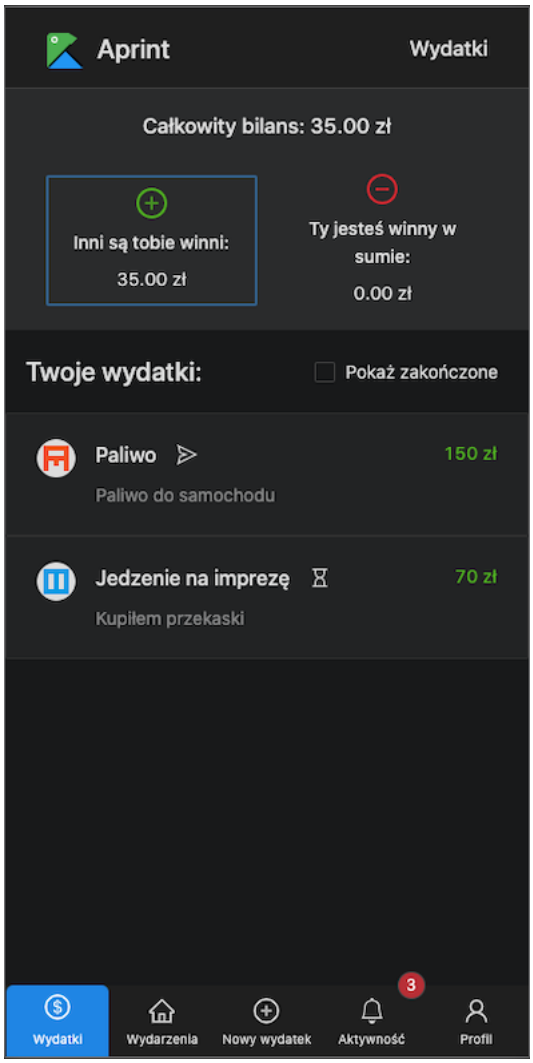
Rysunek 5.1. Ekran logowania i rejestracji

5.2. Wydatki

5.2.1. Ekran listy wydatków

Na tym ekranie użytkownik ma dostęp do listy wszystkich wydatków, w których uczestniczy. Na górze ekranu widzimy całkowity bilans kwoty, którą powinien zapłacić oraz kwotę, którą powinien otrzymać od innych użytkowników oraz całkowity bilans tych dwóch kwot. Te dwa kafelki są interaktywne i kliknięcie jednego nich zmienia listę wydatków. Ta część odpowiada za wymaganie WF10.

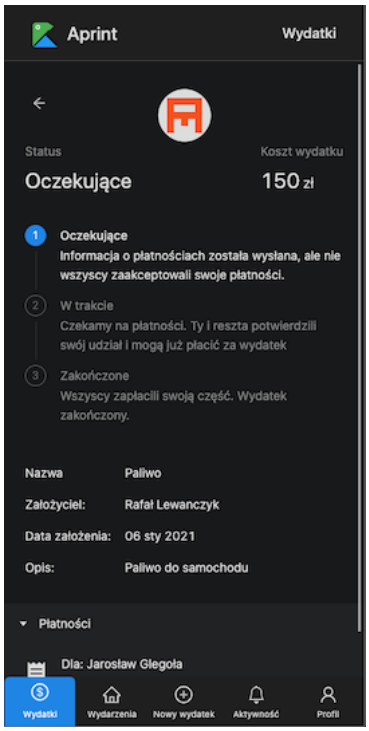
Jeżeli zaznaczona jest opcja *Inni są tobie winni*, na liście wydatków pojawiają się wydatki, których inicjatorem był dany użytkownik, a po zmianie na opcję *Ty jesteś winny w sumie* pojawiają się na liście wydatki, w których użytkownik uczestniczy. Domyślnie zakończone wydatki są ukryte, ale użytkownik ma możliwość zaznaczenia opcji *Pokaż zakończone*, która pokazuje wszystkie wydatki, włączając w to te zakończone.



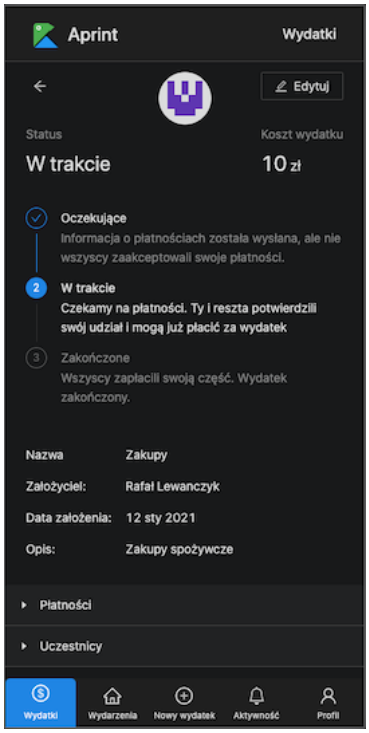
Rysunek 5.2. Ekran listy wydatków

5.2.2. Ekran wydatku

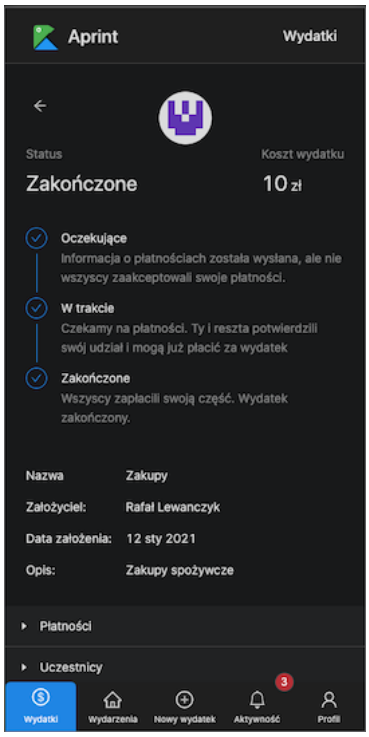
Na ekranie wydatku są ukazane tylko jego dane. Na ekranie jest też pokazany aktualny stan wydatku na komponencie zwanym *krokami*. Dzięki temu komponentowi stan wydatku jest bardziej zrozumiały dla użytkownika. Dodatkowo są też ukazani uczestnicy oraz wszystkie płatności stworzone w ramach wydatku. Inicjator wydatku może z tego poziomu zarządzać wydatkiem, czyli akceptować płatności oraz może zakończyć wydatek. Z tego poziomu może też przejść do ekranu formularza edycji wydatku.



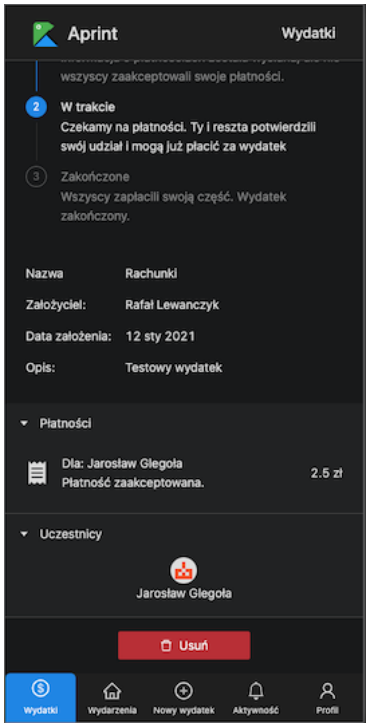
(a) Wydatek w stanie oczekującym



(b) Wydatek rozpoczęty



(c) Wydatek zakończony

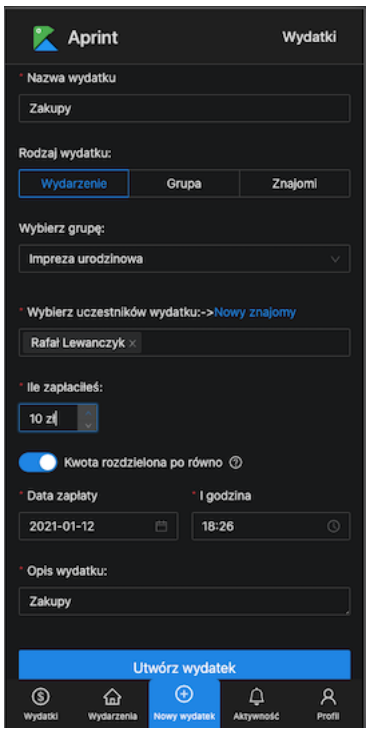


(d) Widok płatności i uczestników wydatku

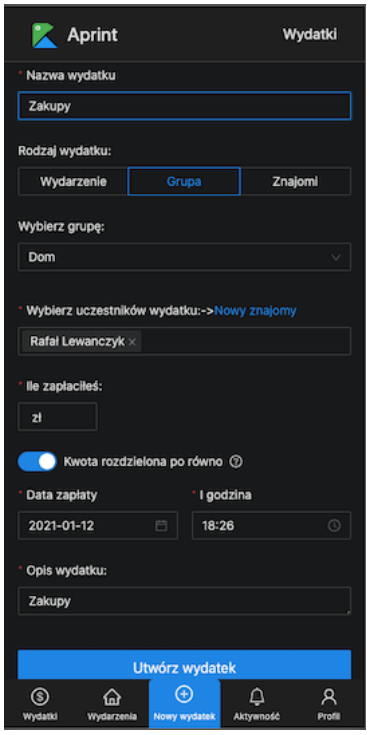
Rysunek 5.3. Ekrany wydatku

5.2.3. Formularz wydatku

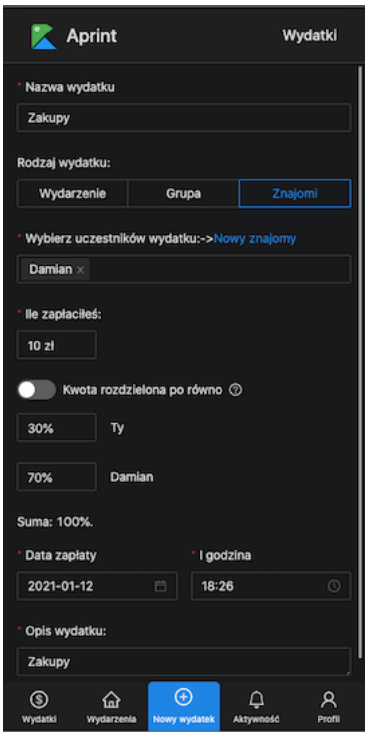
Ekran tworzenia wydatku pozwala na tworzenie nowych wydatków oraz ich edycji. Wydatki możemy tworzyć dla wydarzeń, grup lub wybierając tylko znajomych. Dane dla każdego typu wydatku się różnią, więc formularz będzie wyglądał inaczej dla każdego typu. Ten formularz spełnia wymaganie WF4 oraz WF7.



(a) Formularz wydatku w dla wydarzenia



(b) Formularz wydatku dla grupy

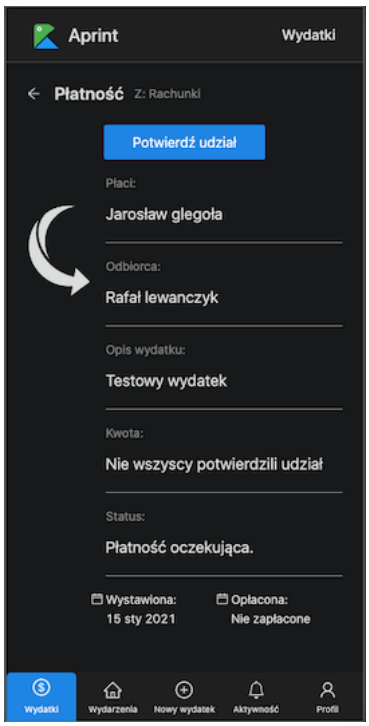


(c) Formularz wydatku dla znajomych

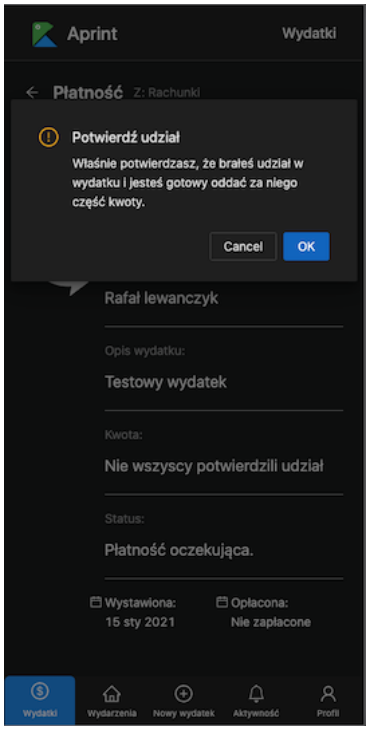
Rysunek 5.4. Formularz wydatku

5.3. Ekran płatności

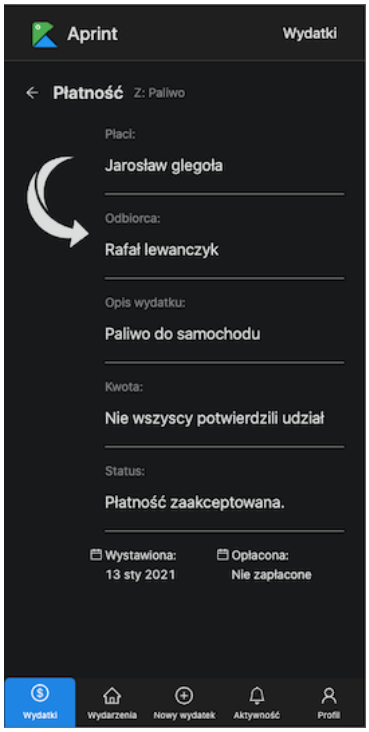
Gdy użytkownik zostanie zaproszony do wydatku, jest tworzony dla niego obiekt płatności, który jest przedstawiony na ekranie płatności. Na tym ekranie są zawarte informacje wydatku, do którego należy płatność, oraz informacje o samej płatności. Z poziomu tego ekranu użytkownik może też zarządzać stanem swojej płatności, czyli akceptować płatność, potwierdzać wpłatę lub odrzucać płatność. Ekran spełnia wymaganie WF5.



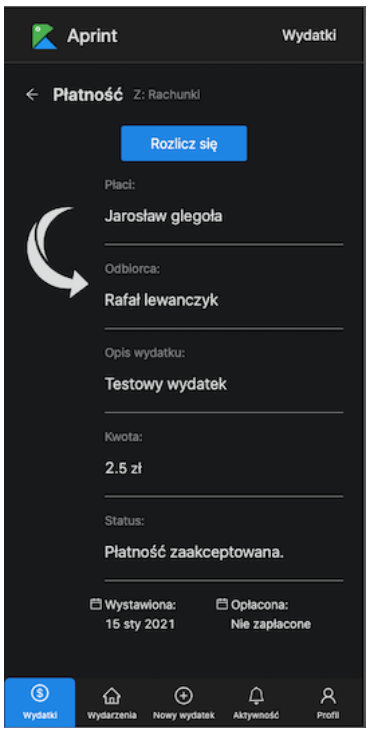
(a) Płatność niepotwierdzona



(b) Potwierdzanie płatności



(c) Płatność potwierdzona



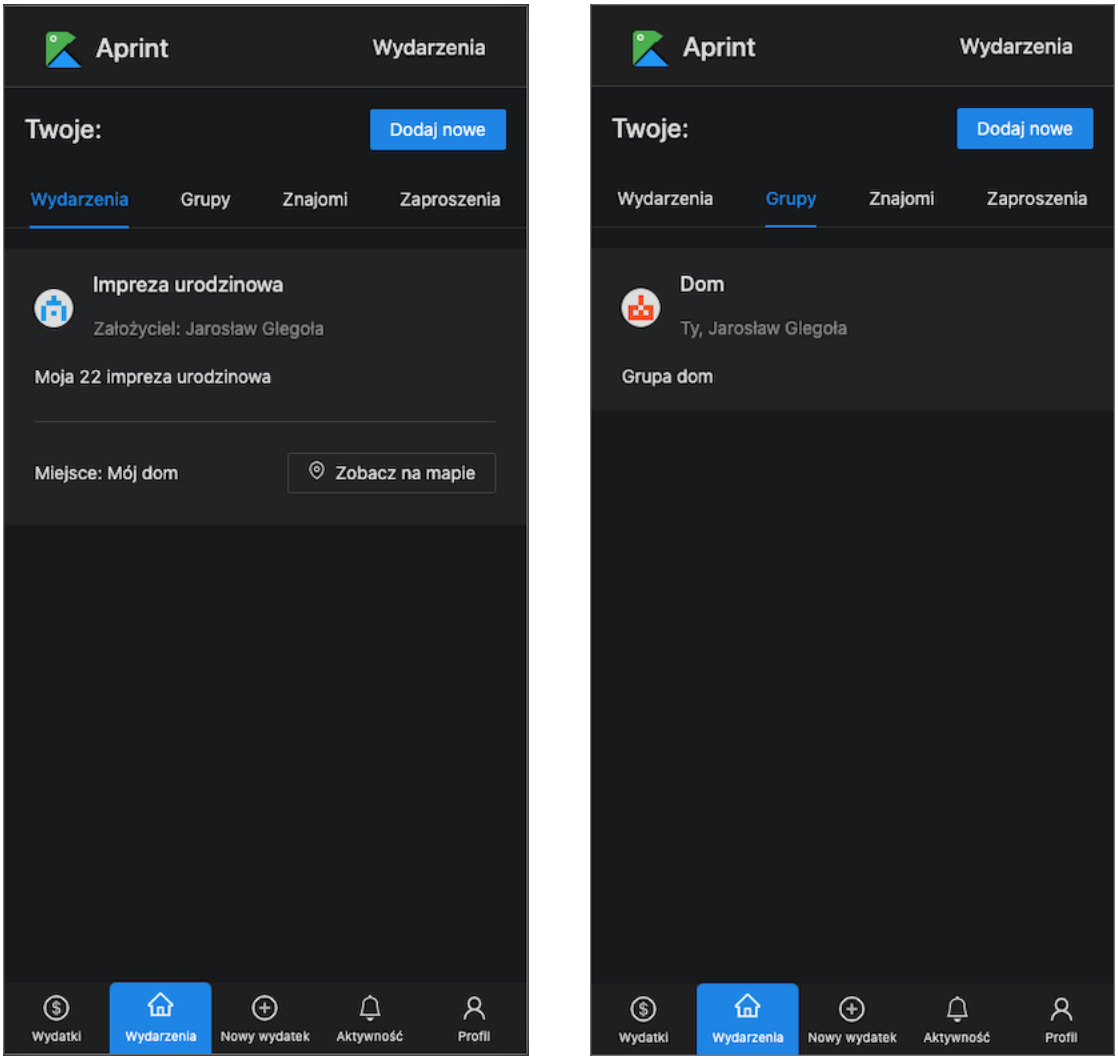
(d) Płatność gotowa do zapłaty

Rysunek 5.5. Ekran płatności

5.4. Ekran wydarzeń, grup i grup znajomych

5.4.1. Listy wydarzeń, grup i grup znajomych

Na głównym ekranie list znajdują się zakładki, dzięki którym użytkownik może wybrać typ elementu. Na tym ekranie mogą pojawić się listy wydarzeń, grup, grup znajomych i zaproszeń. Po kliknięciu na każdy element listy możemy przejść na ekran pojedynczego elementu.



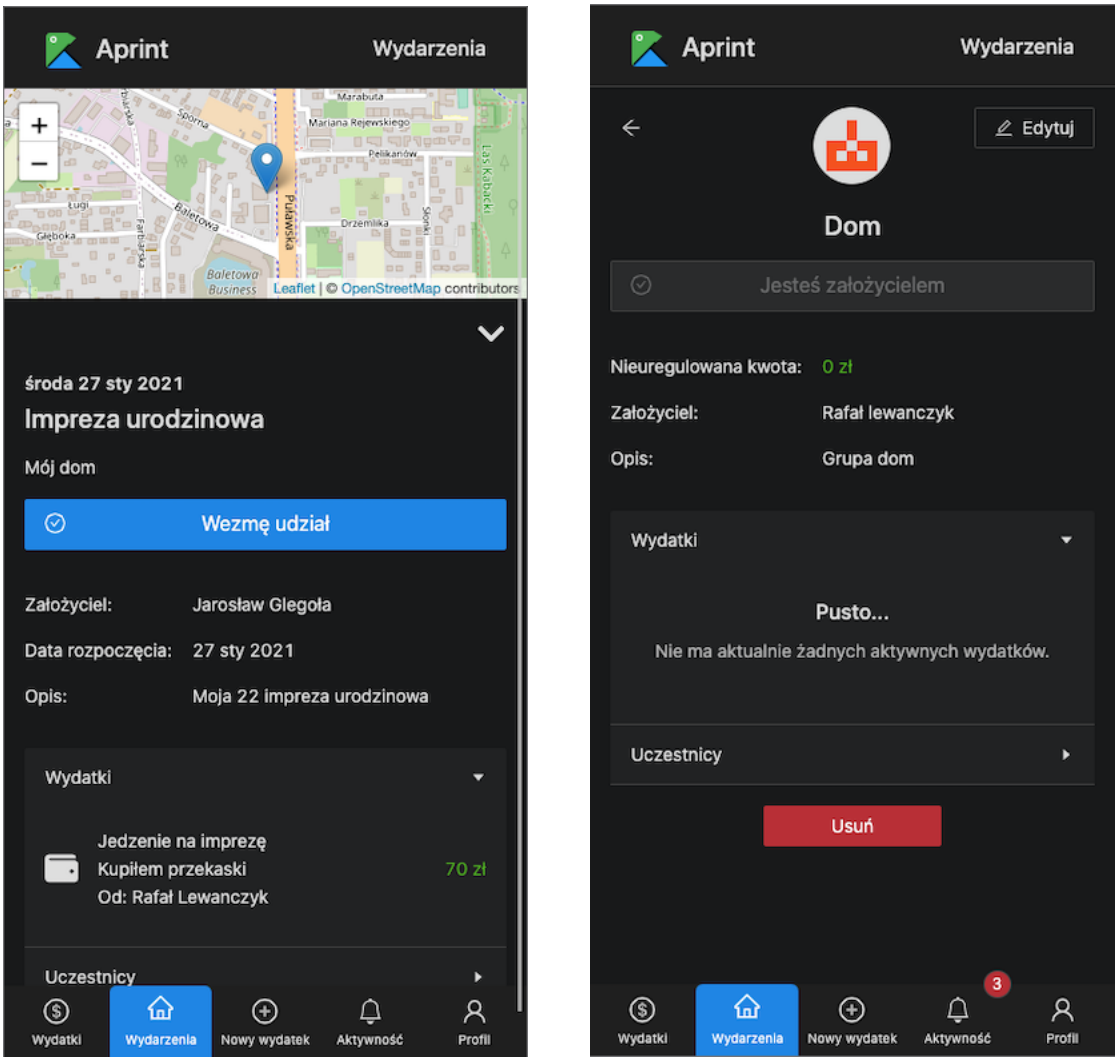
(a) Lista wydarzeń **(b)** Lista grup

Rysunek 5.6. Listy wydarzeń i grup

5.4.2. Ekrany wydarzeń, grup i grup znajomych

Poniżej ukazane są zrzuty ekranu widoków pojedynczych wydarzeń i grup. Na tym widoku są ukazane informacje na temat elementu, lista wydatków oraz wszyscy uczestnicy. Użytkownik z tego poziomu może dołączyć do elementu klikając przycisk *wezmę udział*, gdy użytkownik jeszcze nie zaakceptował zaproszenia, lub opuścić wydarzenie klikając ten sam przycisk, gdy zaproszenie już zaakceptował. Ekran spełnia wymaganie WF6.

Inicjator z tego poziomu może dodawać nowych uczestników, usuwać użytkowników z elementu oraz usuwać dany element.

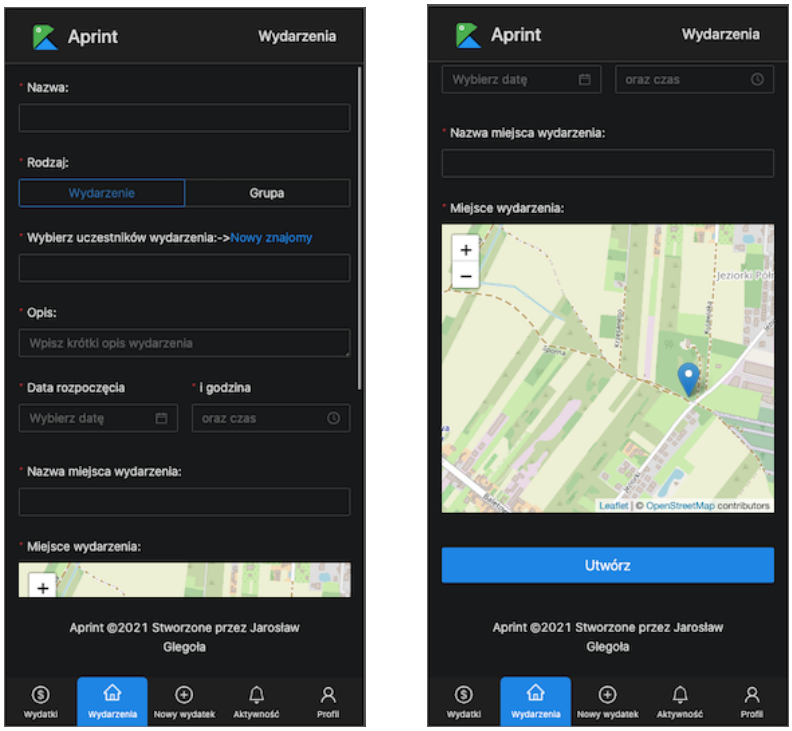


(a) Ekran wydarzenia **(b)** Ekran grupy

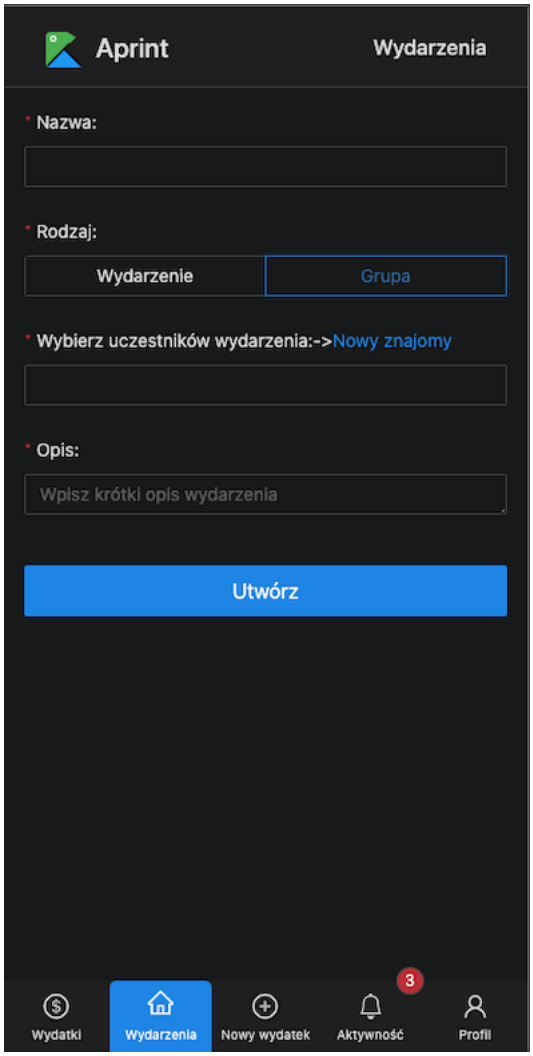
Rysunek 5.7. Ekrany wydarzeń i grup

5.4.3. Formularz dodawania elementu

Formularz dodawania elementu umożliwia dodawanie nowego elementu oraz edycję już utworzonego elementu. Formularz może być wyświetlany w dwóch trybach: *Wydarzenie* oraz *Grupa*. W trybie *wydarzenie* mamy możliwość dodania wydarzenia posiadającego datę rozpoczęcia, miejsca wydarzenia oraz możliwości wyboru miejsca na mapie. W trybie *Grupa* mamy możliwość dodania tylko nazwy uczestników i opisu.



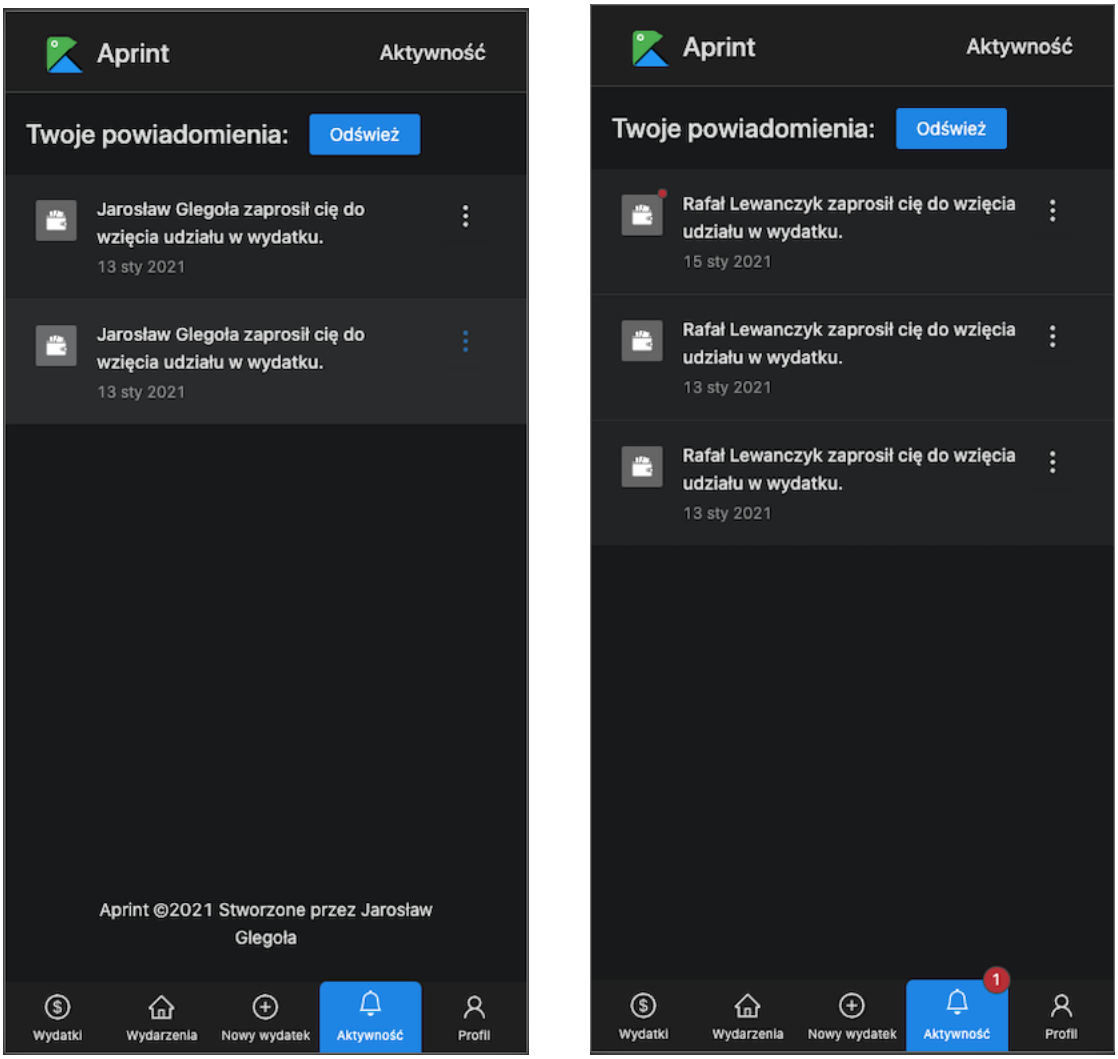
Rysunek 5.8. Formularz w trybie typu *Wydarzenie*



Rysunek 5.9. Formularz w trybie typu *Grupa*

5.5. Aktywność

Panel aktywności wyświetla użytkownikowi wszystkie dostępne powiadomienia w formie listy. Po kliknięciu w elementy aplikacja przekieruje użytkownika do odpowiednich stron np. po kliknięciu w pierwszy element ukazany na rysunku 5.10a zostaniemy przekierowywani do wydatku, do którego użytkownik został dołączony. Dodatkowo, gdy są na ekranie nowe powiadomienia, wyświetlają się przy nich czerwone kropki, aby zwrócić na nie uwagę użytkownika (przykład na rysunku 5.10b). Każde powiadomienie może być usunięte przez użytkownika. Powiadomienia są posortowane w taki sposób, aby najnowsze pojawiały się na samej górze ekranu. Ekran spełnia wymaganie WF8.



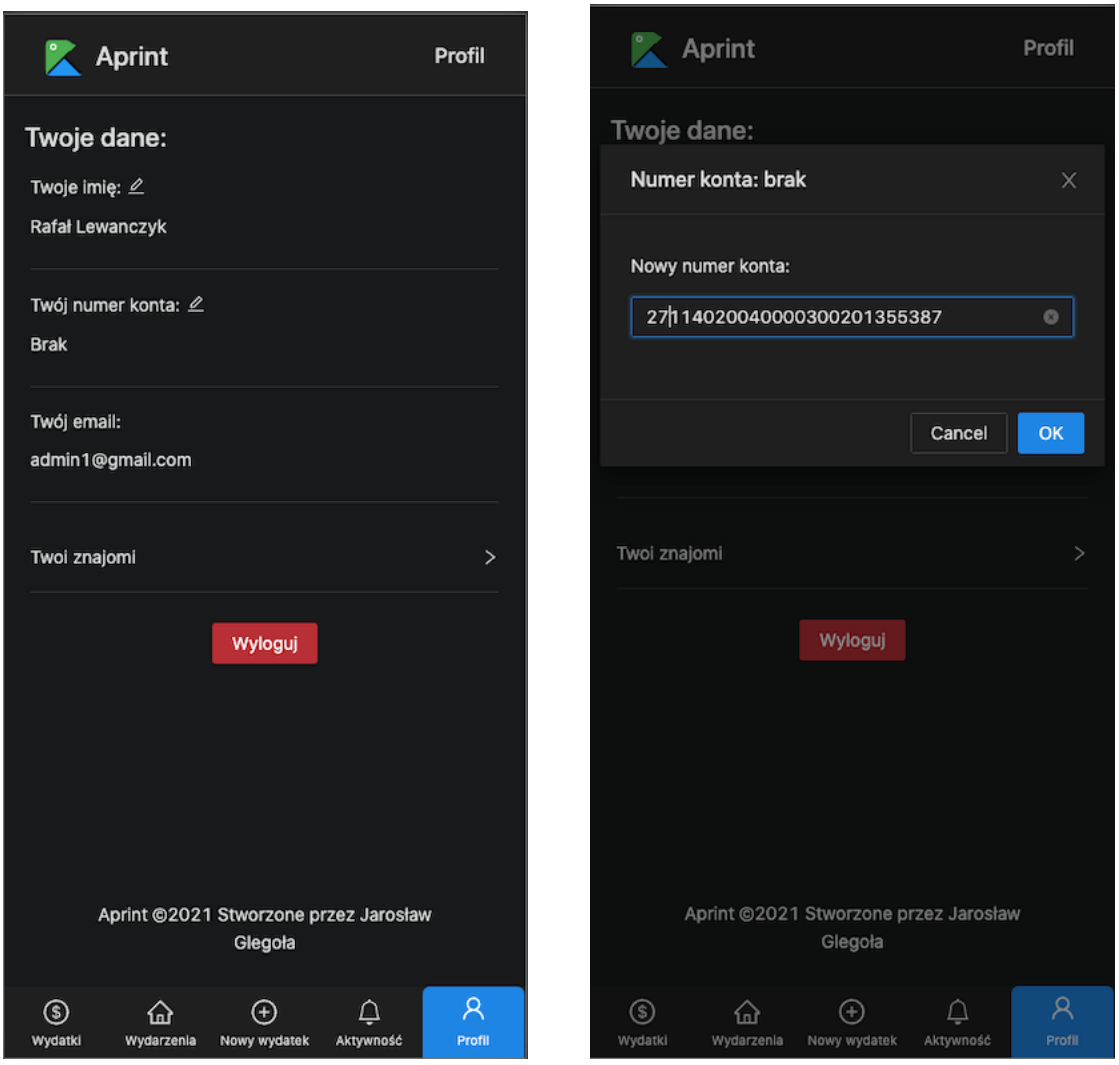
(a) Lista aktywności **(b)** Lista aktywności z nowym powiadomieniem

Rysunek 5.10. Ekran listy powiadomień

5.6. Ekrany użytkownika

5.6.1. Ekran informacji o użytkowniku

Na ekranie użytkownika są wyświetlone dane użytkownika tj. imię, numer konta, e-mail. Z poziomu tego ekranu użytkownik może edytować te informacje oraz wylogować się z konta.

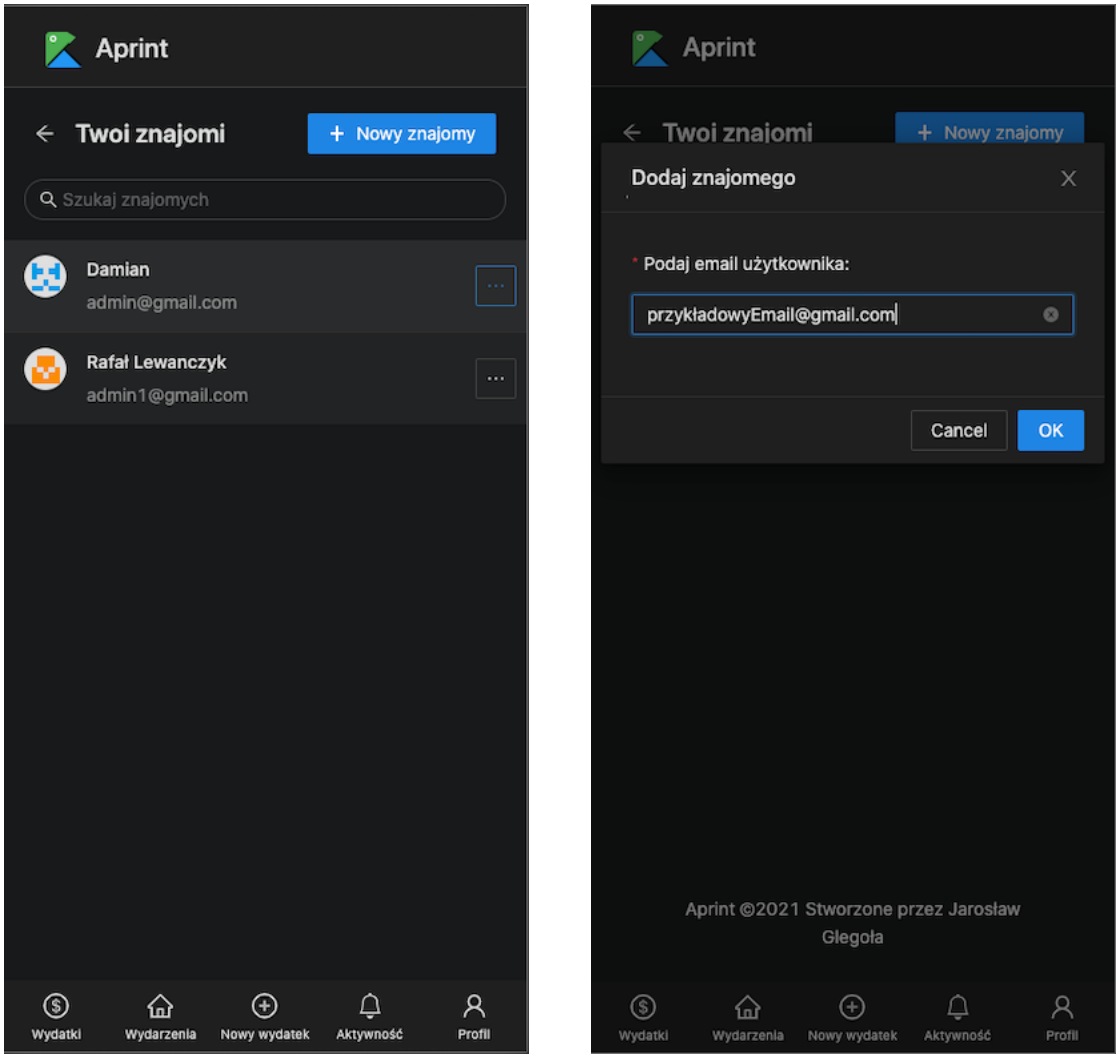


(a) Informacje użytkownika **(b)** Edycja numeru konta

Rysunek 5.11. Ekran użytkownika

5.6.2. Ekran znajomych

Na ekranie znajomych znajduje się lista wszystkich znajomych użytkownika oraz mechanizm dodawania nowych użytkowników. Dodatkowo z tego poziomu użytkownik może też usuwać innych użytkowników z listy swoich znajomych. Ekran spełnia wymaganie WF3.

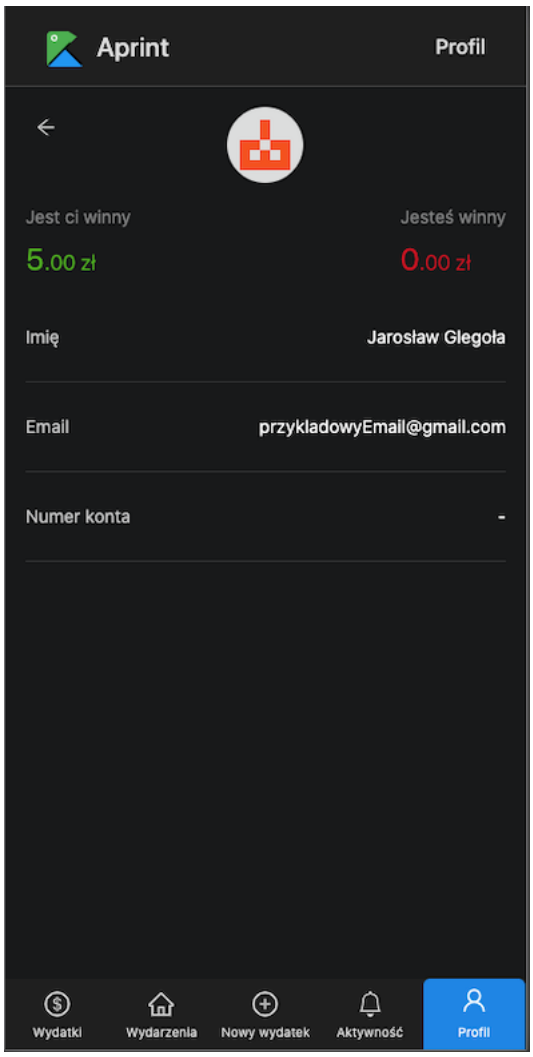


(a) Lista znajomych **(b)** Dialog dodawania nowych znajomych

Rysunek 5.12. Ekran znajomych

5.6.3. Ekran innego użytkownika

Użytkownik może też zobaczyć informacje o innym użytkowniku: jego e-mail, imie oraz numer konta, na który może przelać pieniądze za płatność. Na tym ekranie dostępne są też informacje o balansie pieniężnym pomiędzy użytkownikami. Ekran spełnia wymaganie WF11.



(a) Lista znajomych

6. Podsumowanie

Celem mojej pracy było napisanie aplikacji internetowej wspomagającej rozliczanie się z wydatków pomiędzy użytkownikami. Do utworzenia interfejsu użytkownika wykorzystałem podejście jednostronicowej aplikacji i biblioteki React. Dzięki tej bibliotece i mechanizmie reaktywności w aplikacji rzadko pojawiały się błędy związane z niespójnością stanu z widokiem, co znacznie przyspieszyło pisanie części klienckiej.

Aplikacja została stworzona zgodnie ze specyfikacją Ant Design w trybie ciemnym. Dzięki użyciu komponentów z biblioteki AntD zgodnych z tą specyfikacją aplikacja wygląda bardzo profesjonalnie. Strona kliencka komunikowała się z serwerem z użyciem GraphQL, co było dużym uproszczeniem w tworzeniu aplikacji klienckiej. Dużym atutem takiego podejścia jest to, że mogła być użyta biblioteka, która automatycznie generowała typy schematu GraphQL.

Sam serwer został stworzony przy pomocy Spring oraz Spring Boot wykorzystując Architekturę Heksagonalną. Na końcu przetestowałem moją aplikację na trzy różne sposoby: jednostkowo, integracyjnie oraz z użyciem wizualnej regresji.

Bibliografia

[1] A. Cockburn, "Hexagonal architecture", Dostęp zdalny (16.01.2021): https://alistair.cockburn.us/hexagonal-architecture/, 2005.

[2] *Kotlin*, Dostęp zdalny (16.01.2021): https://kotlinlang.org/.

[3] *Spring*, Dostęp zdalny (16.01.2021): https://spring.io/.

[4] *PostgreSQL*, Dostęp zdalny (16.01.2021): https://www.postgresql.org/.

[5] *Graphql-Kotlin*, Dostęp zdalny (16.01.2021): https://expediagroup.github.io/graphql-kotlin/docs/getting-started.html.

[6] *Graphql*, Dostęp zdalny (16.01.2021): https://graphql.org/.

[7] *React*, Dostęp zdalny (16.01.2021): https://reactjs.org/.

[8] *Rekoncyliacja w bibliotece React*, Dostęp zdalny (16.01.2021): https://pl.reactjs.org/docs/reconciliation.html.

[9] *Typescript*, Dostęp zdalny (16.01.2021): https://www.typescriptlang.org/.

[10] *Eslint*, Dostęp zdalny (16.01.2021): https://eslint.org/.

[11] *Create React App*, Dostęp zdalny (16.01.2021): https://github.com/facebook/create-react-app.

[12] *Apollo Client*, Dostęp zdalny (16.01.2021): https://pl.reactjs.org/docs/reconciliation.html.

[13] *React Testing Library*, Dostęp zdalny (16.01.2021): https://testing-library.com/docs/react-testing-library/intro/.

[14] *Jest*, Dostęp zdalny (16.01.2021): https://jestjs.io/.

[15] *Cypress*, Dostęp zdalny (16.01.2021): https://www.cypress.io/.

Spis rysunków

Lista Wydruków